Bucket-like space partitioning data structures with applications to ray-tracing

Frédéric Cazals, Claude Puech

▶ To cite this version:

Frédéric Cazals, Claude Puech. Bucket-like space partitioning data structures with applications to ray-tracing. 13th ACM Symposium on Computational Geometry, 1997, Nice, France. inria-00510092

HAL Id: inria-00510092
https://inria.hal.science/inria-00510092
Submitted on 17 Aug 2010

Bucket-like Space Partitioning Data Structures with Applications to Ray-Tracing

F. Cazals and C. Puech

*i*MAGIS/GRAVIR-IMAG
BP 53 - 38041 Grenoble cedex 09 - FRANCE
email: Frederic.Cazals@imag.fr

Abstract

Data structures based on uniform subdivisions of the space —also known as bucketing— have the nice properties that they can be walked through very easily and can provide neighborhood relations at low cost. For data sets which are uniformly scattered in $2D$ or $3D$ space, this makes the implementation of algorithms such as ray tracing, nearest neighbors computation or Delaunay triangulation almost trivial. But should the processed data set admit dense clusters, the spatial partitioning does not result in data partitioning so that the performances are collapsing.

Although it has been known for a long time in dimension one that recursive bucket-sort admits a linear complexity for a wide range of probability densities, recursive bucket-like data structures have not received any attention in the computational geometry community. It has been observed in computer graphics that these were the fastest to ray-trace, but the question of understanding why they are not just another space partitioning data structure but rather the only data structure that succeeds in capturing the probabilistic properties of data distribution remains open.

This paper is a first step in this direction and investigates hierarchical recursive and non recursive data structures for ray-tracing. First, we show that precisely analyzing an optimized ray-tracer is a difficult task due to the context sensitivity of the calls costs of the functions called most often. Second, we exhibit statistics showing that if uniform grids are definitely not the right data structure to use for non-uniform distributions, recursive grids are very good at handling such distributions. Third, we present several improvements of the Hierarchy of Uniform Grids data structure, which result for the best cases in running times improved by up to a factor three with reference to the previously best known solution.

1 Introduction

1.1 Computational model, data structures and complexity

Although relatively new areas of computer science such as computer graphics or computational geometry focused primarily on the design and analysis of new algorithms — rendering, pixel based methods, animation, it is fair to say that the algorithms themselves received more attention that the underlying data structures. More precisely, after the pioneering work by Don Knuth in the early seventies, most of the research was carried out in the so called algebraic decision trees model defining a theoretical computer providing only algebraic operations (such as comparison, polynomial expressions evaluation) on the objects manipulated.

In dimension one, prototypes of such algorithms or data structures are quick-sort or balanced binary search trees. And in dimension two or three these are binary space partitioning trees or $k-d$-trees (see [dB$^+$96]). Their common feature is to recursively split the input data set around one of its elements, so that choosing this element as the median results in a balanced data structure whose height is logarithmic in the input size. When the splitting element is chosen at random, the worst case performance relies on the order of the items, and the analysis is generally done considering the $n!$ permutations equally likely.

Almost by then, another family of algorithms and data structures got some attention. Since the previous model is intrinsically binary, that is at any time a decision is made between a constant number of items (two for quick-sort, four for quad-trees, etc), it was noticed that using non constant *branching factors* could result in data structures of much reduced height. Splitting the data set was replaced by splitting the space containing it, which we illustrate with the example of bucket-sort. To sort the keys $\{x_1, \ldots, x_n\}$, the algorithm (1)subdivides the interval $[inf\ x_i, sup\ x_i]$ into sub-intervals called *buckets* of equal length $\delta = (sup\ x_i - inf\ x_i)/n$ (2)assigns each x_i to its sub-interval by computing $\lfloor (x_i - inf\ x_i)/\delta \rfloor$ (3)recurses in any bucket containing more than a constant b (e.g. 5) number of points (4)sorts by insertion the buckets containing less than b points. It is easy to see that if the probability density the x_i are drawn from is uniform, each *bucket* contains $O(1)$ points after step (2) so that the sorting time is linear.

Two major differences with the algorithms/data structures of the former type are that the floor function is needed and that the running time of the algorithm does not depend on the order of the input data set but on the underlying probability density. If the first concern is not restrictive, being dependent on some possibly highly non uniform density is scary. Indeed, if at each step of recursion in bucket-sort all the points but one fall in the same bucket, the complexity becomes quadratic. Fortunately, such data sets must be built as follows $\{x_1 = 1!, x_2 = 2!, \ldots, x_i = i!, \ldots, x_n = n!\}$ and it is easily checked that the domain spanned by such inputs gets huge even for very small samples: for $n = 70$ we already get 10^{100}. (Remember that the diameter of the earth is just 6400 km so that a tiny object of say 1mm with reference to the whole earth would just result in a 10^{10} change of scale!) And more generally, it has been shown ([Dev86]) that forcing a recursive bucket-sort to recurse more than two levels requires fancy densities. The intuitive interpretation is that clusters of points that look very dense at some level of recursion appear to be almost uniform a few levels farther. Apart from bucket-sort, examples of such algorithms and data structures are interpolation-search in dimension one, grid-files, uniform and recursive grids in higher dimension [NH84, Dev86, CDP95]. Very interesting pieces of work are [SD95b, A^+94] where space partitioning based algorithms are tested against divide-and-conquer and sweep-line algorithms for computing the Delaunay triangulation and reporting red/blue line segments intersections. In the two cases the former scheme appears to be the best for evenly distributed data sets and performs poorly on 'bumped' inputs. But as we shall see in this paper, recursive grids and hierarchies of uniform grids (HUG in the sequel) encompass uniform grids and provide a far better solution for such inputs.

More precisely, this paper addresses the question of building and evaluating (in the scope of a ray-tracer) data structures which are able to capture the objects repartition properties of complex real-world scenes. An example of such scenes is the kitchen from figure 5(a): whereas many tiny polygons are found in the neighborhood of the bowl teapot or stack of plates, the middle of the room is just empty! Before reviewing briefly the previous work and giving the paper overview, we want to emphasize the fact that we do not view bucket-like data structures as opposed to the classic ones. Indeed, since we are aiming at reproducing the efficiency of uniform grids for non uniform distributions, it seems natural to use algorithms whose complexity does not depend on the objects' spatial repartition. The construction of a recursive grid does not require such a step, but the one of a hierarchy of uniform grids does so for the clustering step. For applications such as the ones from [SD95b, A^+94] where the data structure is built and used once, the cost of this pre-processing might be worth of consideration. But for ray tracing where the same data structure can be used to compute different views of say a walk-through within a complex environment, it does not matter. This is even more the case since as observed in [CDP95] the recursive grid and the HUG construction times are negligible with reference to the rendering one.

1.2 About ray tracing

Ray tracing primarily consists in finding for a given ray — that is a point and a direction in $3D$, the first object hit. From its early days it became clear that for complex scenes all the objects could not be tested for intersections which led to the development of spatial subdivision data structures. Still, if one can say that from a practical point of view a consensus considers the recursive grid as the best data structure to ray trace ([JdL92, CDP95, Hai96]), very little is known on the reasons why it is so and in particular the question of performing some average case analysis of ray tracing is open. This is a difficult concern especially since some notion of random scene is needed.

An attempt to find spatial partitioning taking into account the non uniformity of objects distribution was made in [Dev88] using integral geometry tools. Unfortunately, it is not clear how the kinematic density based formulas used to estimate the grids' optimal size are compatible with the non uniform distribution of objects. Another interesting piece of work is [MMS94] and focuses on characterizing the complexity of a particular directed segment shoot —which is like ray shooting excepted that the ray is replaced by a segment. More precisely, define a *strongly simple cover* of a segment S as a set of balls (1)which individually do not intersect more than a constant number of obstacles edges (2 on figure 11(a)) even when grown of a factor $\varepsilon > 0$ (2)whose union covers S. Also, let the *simple cover complexity* of S denoted $scc(S)$ be the cardinality of the smallest strongly simple cover of S. Then it is shown that it is possible to build a spatial subdivision of size $O(n)$ in $O(n \log n)$ such that segment shooting queries can be answered in $O(\log n + scc(s))$. Unfortunately, no experimental fact is reported about this data structure even if one can expect a quad-tree like behavior given the way it is built.

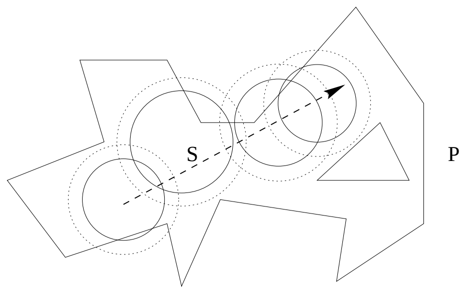


Figure 1: Strongly simple cover of a directed line segment S

1.3 Uniform grid, recursive grid and hierarchy of uniform grids

We briefly review the notions of uniform grid, recursive grid and hierarchy of uniform grids depicted on figures 2 and 3. More details on bucket-like space partitioning data structures can be found in [CDP95, A^+94, AEII85, lBWY80, NH84]. We suppose we are given a set $\mathcal{O}$ of n objects which can be polygons, implicit/algebraic patches, whatever. We just require each object to be bounded and by its length d_i we refer to its axis aligned bounding box diameter length. The reason why we need so few hypothesis on the data manipulated is that we are interest in statistical properties under a huge numbers of objects and not in local properties attached to a particular object.

A *uniform grid* over the set $\mathcal{O}$ is a partition of its bounding box into (n_x, n_y, n_z) subdivisions of equal length along the x, y and z axis. To get a memory requirement linear in the number of objects, the n_i are usually taken so that $n_i = \alpha_i \sqrt[3]{n}$ which we call the $\sqrt[3]{n}$ criterion in the sequel. The α_i are positive constants and the simplest choice is $\alpha_x = \alpha_y = \alpha_z = 1$. Heterogeneous values based on the ratios of the dimensions of the box are also presented in [KS97].

Once a uniform grid is built of $\mathcal{O}$, one may find that some voxels are too populated. If $MAXP$ — standing for MAXimum number of Polygons — is some positive integer (50 e.g.) a *recursive grid* on $\mathcal{O}$ is a hierarchical data structure based on uniform grids such that whenever a voxel contains $N > MAXP$ items (the initial voxel being the scene bounding box) it is recursively split into a uniform grid of N voxels.

Instead of letting the recursive partitioning figuring out which are the most populated areas of the scene, a *hier-*

archy of uniform grids tries to do it more cleverly first by splitting $\mathcal{O}$ into subsets of objects of the 'same' size — the size d_i of the object o_i being for instance the length of its axis aligned bounding box diagonal, and second for each of these by looking for sub-subsets of neighbors. These two steps, respectively called the filtering and clustering steps, are defined as follows:

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 $d_{inf} = inf_i d_i \in [f_1, f_2)$ and $d_{sup} = \sup_i d_i \in [f_{m-1}, f_m)$. 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 $\in [f_k, f_{k+1})$.*

Definition 2 ***δ-connectivity - δ-cSet*** *The distance between two objects is defined as the minimum distance between their bounding boxes. Two objects o_i and o_j are called* δ-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 δ_k be the δ associated to level k. We call* connectivity coefficient *α a strictly positive real number, and we set $\delta_k = \alpha \bar{f}_k$. A set $\mathcal{O}$ is said to be δ-connected or a δ-cSet if $\forall i \in 1..n$ there exists a $j \neq i$ such that $d(o_i, o_j) < \delta$.*

Definition 3 ***Property π_{xyz}, potential cluster and cluster*** *A set $\mathcal{O}$ is said to be π_x or to have property π_x if the objects' projections along the x axis form a δ-cSet. The opposite will be noted $\pi_{\bar{x}}$. Thus a set $\pi_{x\bar{y}z}$ will have the property π_x, $\pi_{\bar{y}}$ and π_z. $\mathcal{O}$ is said to be a potential cluster if it has property π along one or two axis, and a cluster if it is π_{xyz}.*

The whole point is therefore to group objects of the same size and neighbors into clusters. The HUG construction algorithm does not assume anything about the clusters excepted that the highest filter level consists of a single cluster referred to as the *world* cluster in the sequel. Once a cluster has been isolated, its N objects are stored into a uniform grid containing $\alpha^3 N$ voxels obtained by subdividing the cluster bounding box into $\alpha \sqrt[3]{N}$ subdivisions along each axis — as for the recursive grid, this is called the $\sqrt[3]{N}$ criterion in the sequel. The constant α is taken equal to one for every voxel but the world one. Indeed, as observed in [CDP95] taking a small value such as $\sqrt{2}$ for that particular cluster results in a substantial improvement of the rendering time with no significant memory requirement overhead. This observation which we call the $\sqrt{2}$ *criterion* is explained in section 6. The last step of the construction consists in building a hierarchy of clusters in a top-down fashion with respect to the filter levels: for a given grid, all the voxels intersected by a grid of lower level receive a pointer to that grid. The experiments of [CDP95] show that the value of the connectivity coefficient is not very important and that the practical running time of the hug construction is $O(n \log n)$ using $O(n)$ space.

Since most of this paper deals with experimental statistics measured on various test scenes under several data structures, we adopted the acronym

$$dataSetName.dataStructureName$$

to refer to a particular configuration. For example, $kit.r50$ refers to the kitchen model stored in a recursive grid with termination condition $MAXP = 50$; $kit.g30$ refers to the same model stored in a uniform grid with 30^3 voxels; and $kit.u.f3\sqrt{2}$ refers to a three levels HUG with the $\sqrt{2}$ criterion for the world cluster. Also we may use DS and bbox as a short cut for data structure and axis aligned bounding box.

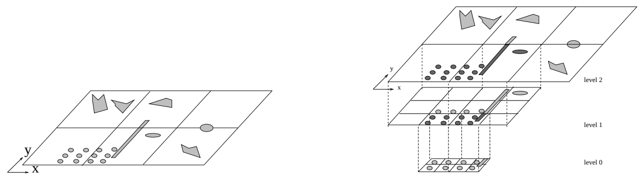


Figure 2: (a) Uniform grid (b) Recursive grid

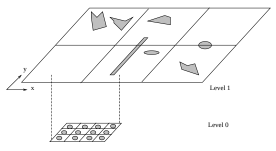


Figure 3: $\mathcal{H}$ierarchy of $\mathcal{U}$niform $\mathcal{G}$rids

1.4 Paper overview

In section 2 we present the test scenes; in section 3 we discuss some statistics of interest to characterize space partitioning data structures and the performances of a ray tracer; in section 4 we raise an important point about the context sensitivity of some functions calls costs; in section 5 we compare the ability of the uniform grid, recursive grid and HUG to partition non-uniform data sets; in section 6 we analyze the rendering statistics of section 3; in section 7 we present several improvements of the HUG data structure and in section 8 we conclude and list unsolved problems.

2 Scenes presentation

We performed tests on four real-world scenes: a kitchen model provided by Don Greenberg at the Cornell University program for Computer Graphics (USA); the Berkeley computer science building model provided by Seth Teller from MIT (USA); random cities generated by Phil Dench's computer program developed at the University of Curtin (Australia); and plants models generated by the AMAP computer program jointly developed by the CIRAD and the University of Montpellier (France). Pictures of these scenes are displayed on figures 5, 6, 7, 8 and 9.

The characteristics of these scenes are the following. The kitchen and the Cal CS building provide accurate models of data sets usually handled in architecture. The random cities are interesting because they are more $2,5D$ models. As for the plants models arranged to form a garden, they provide a more uniform distribution in $3D$ space since most of the polygons are leaves scattered all over the place.

The complexity of these scenes in terms of number of polygons can be made arbitrarily high. For the kitchen scene, just add dishes, forks, etc, and discretize them up to an arbitrarily precision. The full model of Berkeley CS building is already over 1,000,000 polygons ! The random cities can be parameterized to be as large as desired. As for the plants models, it is easy to generate plants of hundreds of thousands of polygons.

For each of the four scenes, we picked a configuration which we believe is representative of this kind of scene and does not require prohibitive computer resources to be interactively displayed and ray-traced. Indeed, all the test were performed on an Indigo2 extreme SGI workstation with 256

Mega-bytes of RAM and a MIPS R4400 processor at 200 MHZ. For example we did not work with the full Berkeley CS building model but either with the walls only (that is no furniture) or the 3rd floor. Considering two or more floors would not have changed the spatial distribution of the polygons in terms of size as well as relative position. We also avoided excessive data clusters in the sense that putting a 20,000 polygon teapot in a 20,000 polygon kitchen is artificial ! The following statistics for the scenes used are displayed on figures 4, 5, 6, 7, 8 and 9:

- the number of polygons
- the ratios min/max and $min/diag$, with min (max) the smallest (largest) object length and diag the scene bounding box diagonal length
- the percentage of objects whose length is less than 20% of the largest size, denoted %.2
- the histogram of the objects lengths and the associated cumulated distribution function. These two functions are depicted on the same chart and are distinguished as follows: the leftmost scale corresponds to the histogram percentages while the rightmost one gives the values of the cumulated distribution function. The x axis scale stops at .2 because most of the objects' sizes are smaller than the size of the largest objects over 5.

Of course these statistics do not provide any information on the relative position of the objects and this issue is discussed in a companion paper [CS97].

	#polys	min/max	min/diag	%.2
Kitchen	20947	9,0e-4	6,5e-4	99,29
CSB walls	32811	5,7e-4	3,6e-4	99,49
CSB 3^{rd} floor	117605	1,0e-5	8,1e-6	99,99
City	52136	2,9e-3	1,0e-3	99,85
Garden	45293	4,3e-3	1,1e-3	99,72

Figure 4: Test scenes statistics

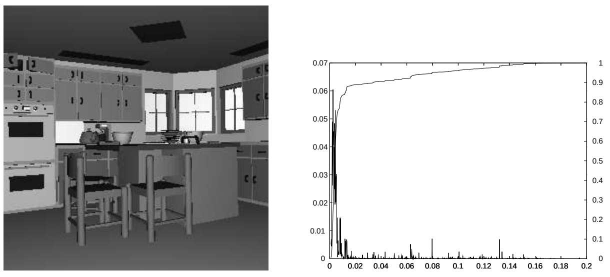


Figure 5: Kitchen scene

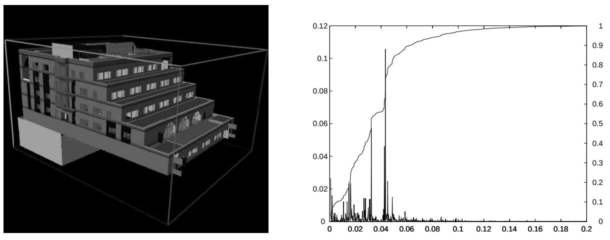


Figure 6: Walls of the Berkeley CS building

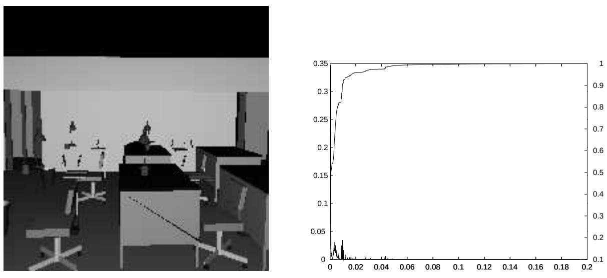


Figure 7: An office in the Berkeley CS building 3^{rd} floor

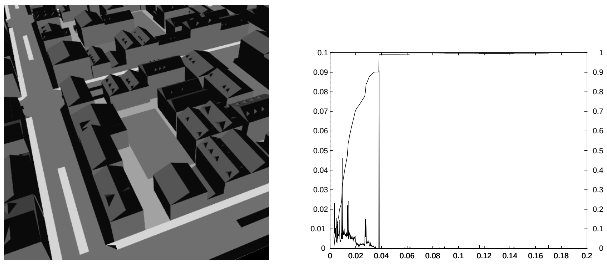


Figure 8: City model

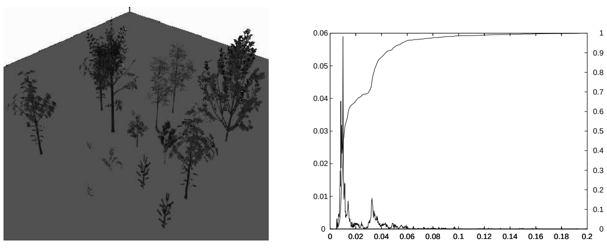


Figure 9: Garden scene

3 Ray tracing: statistics of interest

Pretty much all the information describing the ability of a data structure to ray-trace is captured by the number of ray-polygon and ray-bounding box intersection tests, the number of voxels traversed, the memory requirements expressed in mega-bytes, and the rendering time in seconds, which we respectively denote by #pit, #bbi, #vt, #mbs, and t. But to these high-level statistics correspond more precise metrics, some of which characterize the way the DS stores the scene primitives and are thus viewpoint independent, and some of which depend on the particular image rendered. More precisely, for each hierarchy level of our data structures we define the following statistics:

•**resources repartition statistics:** By %obj, %grids, %voxels, and %ptrs we respectively refer to the percentages of polygons stored at that level, grids voxels and pointers (to objects and sub-grids) allocated to that level with reference to the total number of such entities in the DS. By $\%vol$ we denote the sum of the volumes of the grids of that level with reference to the volume of the biggest grid. Note that as soon as there is more than one non empty level, $\sum_{all\ levels} \%vol > 100$. As we shall see, this is a good criterion to evaluate a HUG.

•**space occupancy statistics:** By μo_g we refer to the average number of objects by grid. By $\%o$ we refer to the percentage of non empty voxels —that is containing objects. By μo_v, σo_v, $\mu o_v^{'}$, $\sigma o_v^{'}$ we refer to the average and standard deviation of the number of objects by voxel, and to the average and standard deviation of the number of objects by *non-empty* voxel. The quantities μg_v, σg_v, $\mu g_v^{'}$, $\sigma g_v^{'}$, have the same meaning but are related to the number of grids referenced by each voxel

•**rendering statistics:** By $\%pit$, $\%pi$, $\%bbi$, $\%vt$ we refer to the percentages of ray-polygon intersection tests, polygons intersected, bounding boxes intersection tests and voxels traversed with reference to the total number of such entities for the whole data structure. The ratio $\%pi/\%pit$ is equal to the number of polygons intersected over the number of tests. These viewpoint dependent statistics were computed while rendering 250x250 pixels images and averaged over a sequence of ten runs.

4 Heavy duties and functions calls costs

The rendering statistics depend on which functions are most often called and on how much each of these calls costs. An important point we want to raise here is that the running time of a function with more than one exit point can be context-sensitive. To see why, consider the following ray-polygon intersection algorithm:

```
          Intersect(ray, polygon)
• [$test_1$]  if the ray and the polygon are coplanar, return
• compute the intersection point $p$ between
          the ray and the polygon plane
• [$test_2$]  if $p$ does not belong to the polygon
          $2D$ bounding box, return
• [$test_3$]  compute the ray-crossing or winding number
```

Writing this routine this way is important since the two first tests are constant time whereas the third one depends linearly on the polygon number of vertices. But if one can expect $test_1$ to be passed most of the time, successing to $test_2$ obviously depends on the context from which the routine is called. Indeed, the ray-polygon intersection test is called whenever a ray is crossing a voxel referencing polygons. But the probability for a ray to intersect a polygon *given that* it intersects the voxel is proportional to the surfaces areas ratio ([S76, CS97]). Which shows that the test will be more expensive for polygons whose size matches the referencing voxel surface area.

The best one can look for is therefore average running times. Candidate tools to get these are the profiling programs such as `prof` and `pixie` on SGI. Unfortunately, the number of processor cycles spent within each function is computed by sampling the program counter every milli-second which is too sparse for functions whose running time order of magnitude is the micro-second. The only valid information one can get from a profile is therefore a rough idea of the functions called most often. In our case:

ray polygon intersection: see above

ray bounding box intersection and grid walk through configuration: intersection between the ray and a $3D$ bounding box (either the world bbox, or a cluster bbox for the HUG, or a sub-grid bbox for the recursive grid), and in the case of intersection, configuration of the parameters of the walk through across that grid

next voxel computation: given a location in the grid, computation of the next voxel traversed by the ray

To compute the costs t_{pi}, $t_{bi} + t_{rg}$ and t_{nc} of these functions, we therefore timed each of them with two calls to the system `times()` function when stepping in and out. The average times are reported on the top part of figure 10 and confirm the context sensitivity of the calls costs. For example the average cost of the ray-polygon test varies from $1.19e-6$ seconds for the close view on the third floor of the Berkeley CSB with a $r50$ down-to $2.97e - 7$ seconds for a recursive grid with recursion termination condition $MAXP = 500$. This drop corresponds to a lower percentage of tests reaching the winding number computation as shown by the $\%s_1$ and $\%s_2$ values expressing the percentages of successes to $test_1$ and $test_2$.

This variability makes the prediction of the expected performances difficult since as shown on the bottom part of figure 10 the numbers of calls of the ray-polygon, ray-bounding box and voxel crossing algorithms respectively expressed in million, thousand and million varies a lot from one data structure to another. In other words reducing the number of calls by the appropriate spatial subdivision results in more expensive calls and it is not clear where the optimum is.

DS	t_{pi}	$t_{rg} + t_{bi}$	t_{nc}	$\%s_1$	$\%s_2$
$csbc.u.f2.\sqrt{2}$	1.6e-6	3.2e-5	8.4e-6	72.3	9.35
$csbc.u.f3.1$	8.6e-7	2.9e-5	9.3e-6	70.3	4.91
$csbc.r50$	1.1e-6	4.6e-5	6.1e-6	84.4	6.63
$csbc.r500$	2.9e-7	5.2e-5	7.1e-6	87.7	1.69
$csbc.g80$	1.0e-6	5.0e-5	4.1e-6	90.6	6.29

DS	$\#pit$	$\#bbi$	$\#vt$	$\#mbs$	t
kit.u.f3.$\sqrt{2}$	5,3	509	2	4	57
kit.u.f2.$\sqrt{2}$	5,3	367	2,3	3.63	47
kit.r50	3,9	231	2,9	7.87	45
kit.r500	18	138	2,4	4.38	86

Figure 10: Call costs, number of calls and overall performances

5 Voxels occupancy and spatial partitioning

To figure out how good at handling non uniform distributions the uniform grid, recursive grid and HUG are, we computed the statistics described in section 3 for the kitchen scene and the three configurations $g28$, $u.f3.\sqrt{2}$ and $r50$. In addition, we plotted for each hierarchy level of each data structure the histogram and the cumulated distribution function of the number of objects per voxel. The results are displayed below.

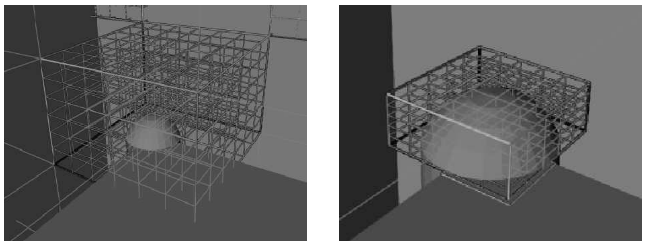


Figure 11: (a,b)Densely populated areas: chair foot top stored by a recursive grid and a HUG

level	%obj	%grids	%voxels	%ptrs	%vol	#mbs
0	100	100	100	100	100	
%	100	100	100	100	100	
Σ	20891	1	21952	58387		2.11

level	μo_g	$\%o$	μo_v	σo_v	$\mu o_v'$	$\sigma o_v'$
0	1	28.4	0.952	23	3.35	42.8

level	$\%g$	μg_v	σg_v	$\mu g_v'$	$\sigma g_v'$
0	0	0	0	0	0

Figure 12: Voxels occupancy for $kit.g28$

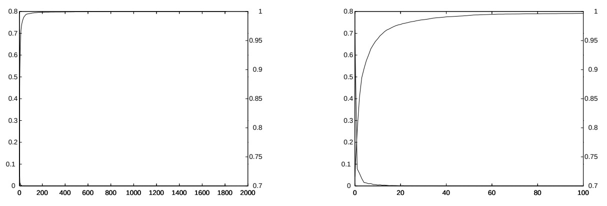


Figure 13: Distribution of the number of objects by voxel for $kit.g28$: (a)Full histogram (b)Zoom on the left part

level	%obj	%grids	%voxels	%ptrs	%vol	#mbs
0	82.6	70	65.3	66.1	0.152	
1	5.84	29.7	5.22	10.9	0.0218	
2	11.5	0.286	29.5	23	100	
%	100	100	100	100	100	
Σ	20891	350	31443	103150		4.00

level	μo_g	$\%o$	μo_v	σo_v	$\mu o_v'$	$\sigma o_v'$
0	1.18	43.6	3.32	6.87	7.62	8.7
1	0.197	81.4	6.01	6.1	7.38	5.96
2	40.4	34.2	2.49	7.08	7.28	10.6

level	$\%g$	μg_v	σg_v	$\mu g_v'$	$\sigma g_v'$
0	0	0	0	0	0
1	84.6	0.87	0.403	1.03	0.17
2	3.37	0.0669	0.449	1.99	1.48

Figure 14: Resources repartition and Voxels occupancy for $kit.u.f3.sqrt2$

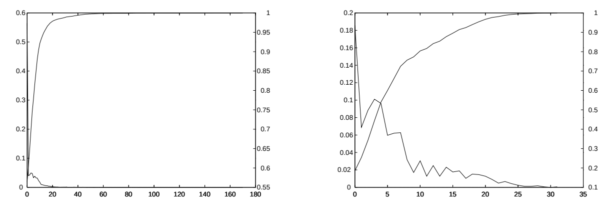


Figure 15: Distribution of the number of objects by voxel for $kit.u.f3.sqrt2$. levels 0 and 1

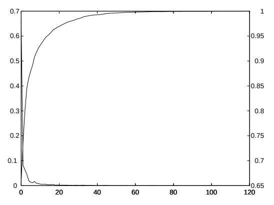


Figure 16: Distribution of the number of objects by voxel for $kit.u.f3.sqrt2$. level 2

level	%obj	%grids	%voxels	%ptrs	%vol	#mbs
0	0.11	0.813	0.21	0.713	3.37	
1	11.5	39	13.8	22	0.00144	
2	70.4	59.8	50.1	59.8	0.67	
3	18	0.407	36	17.4	100	
%	100	100	100	100	104	
Σ	20891	246	61030	196232		7.87

level	μo_g	$\%o$	μo_v	σo_v	$\mu o_v'$	$\sigma o_v'$
0	0.135	89.1	10.9	9.22	12.3	8.88
1	0.296	58.9	5.15	6.49	8.75	6.33
2	1.18	48.4	3.84	6.81	7.93	7.95
3	44.2	27.7	1.55	4.75	5.59	7.67

level	$\%g$	μg_v	σg_v	$\mu g_v'$	$\sigma g_v'$
0	0	0	0	0	0
1	0.0238	0.000238	0.0154	1	0
2	0.314	0.00314	0.056	1	0
3	0.67	0.0067	0.0816	1	0

Figure 17: Resources repartition and voxels occupancy for $kit.r50$

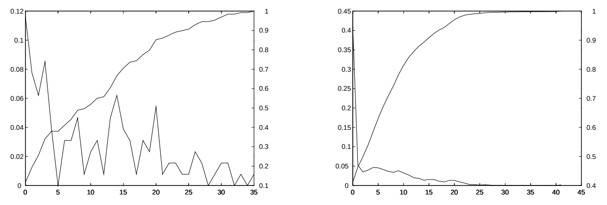


Figure 18: Distribution of the number of objects by voxel for $kit.r50$. levels 0 and 1

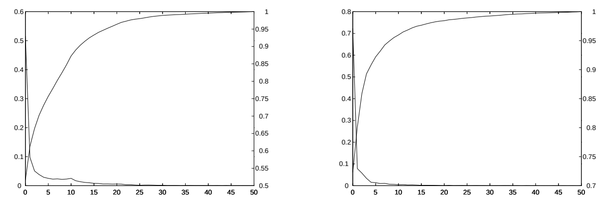


Figure 19: Distribution of the number of objects by voxel for $kit.r50$. levels 2 and 3

For the uniform grid the hierarchy has a single level but the way the polygons are referenced is disastrous: the average number of polygons per voxel is less than 1 and the standard deviation is 23. The cumulated distribution function actually shows that about 98% of the voxels contain a reasonable (less than 20) number of polygons. But the remaining ones admit a huge variability and are causing overall bad performances.

For the HUG, we first observe that the most resources consuming level is the level 0, which corresponds to the fact that most polygons of the scene are small polygons. This observation shows that the $\sqrt{2}$ criterion mentioned in section 1.3 is not too memory consuming since the big objects represent 11.5% of the scene. The voxels occupancy statistics are far better than for the uniform grid since $\mu o_g'$ and $\sigma o_g'$ are within a factor two or so. Also, the distribution of clusters is good since $\mu g_v'$ is bounded by 1 with a small standard deviation.

As for the recursive grid, we first note that two levels of recursivity are sufficient to capture about 88% of polygons within voxels containing less than $MAXP$ references. Also, the voxels occupancy statistics for these two levels are good, especially for the third. Indeed, in spite of a relatively low percentage of non empty voxels (27.7%), μo_g, σo_g $\mu o_g^{'}$, and $\sigma o_g^{'}$ are slightly better than for the HUG.

The HUG therefore performs pretty well at referencing the polygons. But the important fact is that the recursive grid which is built in a totally blind manner performs even better if one forgets the memory requirements. The point is that when a given volume containing n objects is subdivided into n voxels, this results in relatively small voxels so that most of the big polygons are referenced from scratch in voxels containing less than $MAXP$ polygons, which is also the case for the majority of the small polygons. And one level of recursion later, this is even more true for most of the small polygons, that is most of the polygons. Of course, one drawback is a relatively high percentage of empty voxels which affects the memory requirements. Another interesting fact is that the deeper in the hierarchy, the worse the statistics. This is a consequence of the above remark: at level 3, all the polygons take advantage of the thin subdivision induced by the small polygons scattered all over the place. But this property disappears when the grid recurses to handle dense spots. For example on figure 11, the 96 polygons modeling the top of the chair foot induce the subdivision of a voxel to eventually occupy just a corner. The HUG on the opposite figured out this dense spot through the filtering and clustering steps.

6 Rendering statistics: a detailed analysis

We now consider the viewpoint dependent statistics described in section 3 and averaged over a sequence of ten runs for 250x250 pixels pictures of the kitchen. We skip the uniform grid case whose running time is about 5 times slower than the two other data structures.

We start with the analysis of the performances for the viewpoint of figure 5 depicted on figure 20. First and for the two data structures, the percentage of intersected polygons with reference to the total number on intersection tests is pretty low (below 5%) which is not very satisfactory for the whole space partitioning paradigm ! Second, the majority of the ray-polygon intersection tests comes from the highest level of the hierarchy: 85.6% and 79.5% respectively for the recursive grid and the HUG. That's the reason of the efficiency of the $\sqrt{2}$ criterion mentioned in section 1.3, and that's also the reason why the recursive grid performs better than the HUG (see the discussion at the end of the previous section). As for the repartition of the number of bounding boxes intersected and the number of voxels traversed, the difference lies in the fact that the HUG inherently has more grids than the recursive grid (350 vs. 246, see section 5), and that these resources are allocated differently as already observed.

The discussion for the viewpoint of figure 21 is about the same. We observe that the rendering times are increased in a factor 1.37 and 1.29 for the recursive grid and HUG respectively, and that the number of ray-polygon tests is higher for the HUG while the opposite holds for the number of voxels traversed. For the HUG, the most consuming level in term of ray-polygon is now level 0, which is fine too since the voxels occupancy statistics are as good as the one of level 2 — fig. 14. And for the recursive grid, the most consuming level is now the second which is not very good since as shown on fig. 17, the deeper in the hierarchy, the higher the expected number of polygons per voxel.

level	%pit	%pi	%(pi / pit)	%bbi	%vt	t
0	10.9	7.61	2.16	53.6	49.7	
1	3.51	3.74	3.29	21.8	22.2	
2	85.6	88.7	3.2	24.5	28.1	
%	100	100		100	100	
Σ	5.3e6	1.6e5		5.0e5	2.1e6	57

level	%pit	%pi	%(pi / pit)	%bbi	%vt	t
0	6e-3	1e-3	0.735	9e-3	9e-3	
1	0.367	0.507	5.05	1.03	0.864	
2	20.2	15.9	2.87	44.9	42.2	
3	79.5	83.6	3.84	54.1	56.9	
%	100	100		100	100	
Σ	3.9e6	1.4e5		2.3e5	2.9e6	45

Figure 20: Rendering statistics for $kit.u.f3.sqrt2$ and $kit.r50$

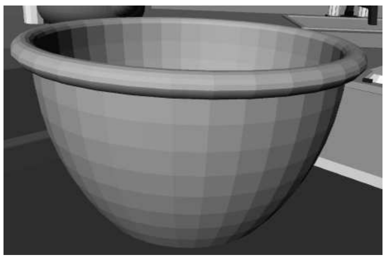


Figure 21: Zoom on tiny polygons

level	%pit	%pi	%(pi / pit)	%bbi	%vt	t
0	57.9	30.9	1.94	35.6	63.5	
1	1.15	2.03	6.37	46.6	31.4	
2	41	67	5.93	17.8	5.11	
%	100	100		100	100	
Σ	10.1e6	3.6e5		7e5	2.4e6	74.27

Figure 22: Rendering statistics for $kitchenBowlView.u.f3.sqrt2$

level	%pit	%pi	%(pi / pit)	%bbi	%vt	t
0	0	0	$\emptyset$	0	0	
1	3.86	3.79	3.43	7.45	14.4	
2	79.4	59.3	2.61	67.1	81.8	
3	16.7	36.9	7.69	25.5	3.77	
%	100	100		100	100	
Σ	5.9e6	2e5		4.9e5	3.3e6	62.73

Figure 23: Rendering statistics for $kitchenBowlView.r50$

7 Improvements of the HUG data structure

7.1 Some observations on filtering and clustering

Since the clustering step is intended to find closely fitting bounding boxes over sets of objects, an attractive idea is to use non axis aligned bounding boxes once the clusters have been isolated. This is even more so that, as already observed in section 4, the probability for a ray to cross a convex volume depends on the surface area delimiting this volume. So that a natural way to reduce the number of grids traversed per ray could consist in packing these objects into bounding boxes of minimal surface area. It should be emphasized that in doing so one can only expect to reduce the number of ray-box intersection tests, but not the number of ray-polygon tests. Indeed it is not clear that the orientation that minimizes the box surface area is also the one that makes the distribution of the number of polygons per voxel the best. This latter issue is actually related to the way a given box is partitioned into voxels and these two issues are discussed in section 7.2 and 7.3.

Another important question is the goodness of the original clustering algorithm. Applied to the kitchen scene, this algorithm does not result in surprising clusters: we indeed get separately the doors knobs, the chairs feet tops, etc. But some clusters as the one of figure 27(a) where the bowl, the knife handle and a part of the breadboard have been put together are more confusing since the configuration of figure 27(b) would be expected. In the same vein, this clustering algorithm on the smallest level of a three levels filter for the garden scene results in huge clusters containing leaves sparsely scattered in the cluster volume as depicted on figure 28, while a greater number of smaller clusters might appear as more natural as shown on figure 29. The problem here is that specifying a cluster as a set of objects whose projection on the three axis is (almost) connected is not sufficient. A symptom of this kind of problem is the $\sum_{all\ levels} \%vol$ statistic described in section 3. Getting a too big value for it —say over 120%, means that the number of voxels traversed during the rendering process is significantly increased and so is the rendering time. Section 7.4 presents a new clustering algorithm getting around this problem.

At last, a key issue is the synergy between the filtering and the clustering steps. Due to the intuition provided by the bucket-sort behavior, the original work of [CDP95] focussed on three levels hierarchies. But as seen in section 6, most of the intersection tests are performed at the highest level so that having a thinner grid for that one is better —remember the $\sqrt{2}$ criterion and the comment about the recursive grid. One way to achieve this goal is to merge the two first levels of the filter which corresponds to a binary partition of the objects' lengths histogram. But as mentioned above, a two levels filter should try to keep apart the big objects that would bridge the gaps from the small ones thus contributing to clusters of volume of the same order of magnitude as the whole scene. Since the filtering and clustering are performed in sequence, finding the threshold value inducing too high values the $\sum_{all\ levels} \%vol$ statistic is difficult. And that's even more the case that scenes as the city are not really prone to clustering — see fig. 8. In practice and due to the observation made in section 2, a two levels filter is just computed by putting together all the objects whose size is below a small percentage of the biggest object size — say 5%. As to decide whether or not a scene is amenable to clustering, some integral geometry based tools are proposed in [CS97].

7.2 Clusters orientation

For a given cluster whose polygons are defined by n vertices $\{P_i\}_{i=1..n}$, the problem reduces to the computation of a box of minimal surface area containing the points P_i. Although this question has not received a specific attention in the computational geometry community, the related topic of computing bounding boxes of minimal volume has been shown to require sophisticated data structures such as the Voronoi diagram to be performed in optimal $O(n \log n)$ time. This led computer graphics programmers to look for approximate and more tractable solutions, and it was pointed out that the so-called Principal Components Analysis (PCA) used by statisticians could be useful ([Wu92, GLM96]). The goal of this method is to find a new frame such that the inertia of the points projection on the new axis is maximum. The calculation, which runs in linear time, is the following.

Let $X_{n,3}$ be the matrix containing the $3D$ coordinates of the n points $\{P_i\}_{i=1..n}$, the coordinates of P_i being noted $(v_{i,1}, v_{i,2}, v_{i,3})$, and let $C_{n,3}$ be the covariance matrix defined as follows:

$$C_{n,3} = \begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

with

$$C_{i,j} = \frac{\sum_{k=1}^{n} v_{k,i} v_{k,j}}{n} - \mu_i \mu_j \text{ and } \mu_i = \frac{\sum_{k=1}^{n} v_{k,i}}{n}$$

The theory shows that the axis sought are the eigenvectors (u, v, w) of $C_{n,3}$, which are orthogonal to one another since the matrix is symmetric. Once these new vectors have been computed, the coordinates of the projected points on say the axis defined by the vector u are given by $X_{n,3}\ .\ u$. Or equivalently, if we note T the matrix $T = [u\ v\ w]$, the coordinates of the points in the new coordinate system are given by $T^{-1} X^t_{n,3}$, with $X^t_{n,3}$ the transposed matrix of $X_{n,3}$.

We implemented this algorithm and were expecting results as the one of figure 24(a), where the red box is obviously tighter than the yellow one. Unfortunately, configurations as the one of figure 24(b) are pretty frequent. And plotting the ratios of the volumes and surface areas for PCA computed bounding boxes over axis aligned ones for the bounding boxes of the clusters of the kitchen gives the graphics of figure 25 — as usual, the left scale refers to the histogram classes percentages, and the right one to the cumulated percentages.

Even if these diagrams are not presented in [GLM96], the reason why the PCA method does not work that well is mentioned: points in the interior of a cluster which ought not influence the selection of a bounding box, can have an arbitrary impact on the eigenvectors. The authors therefore propose a refined method based on the convex hull points. This works fine for their collision detection purposes, but the situation here is more involved as observed at the beginning of this section.

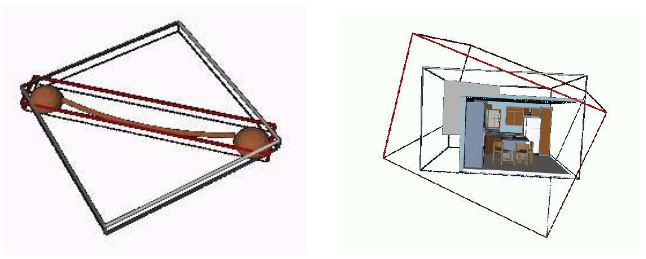


Figure 24: (a,b)Principal Components Analysis and bounding boxes: favorable and less favorable cases

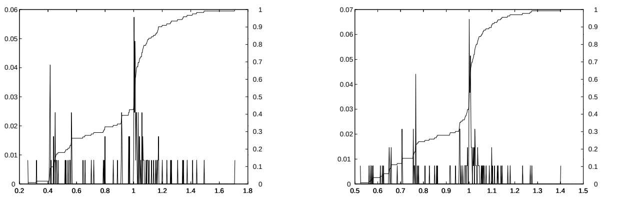


Figure 25: Volume and surface area ratios for the kitchen scene

7.3 Heterogeneous gridings

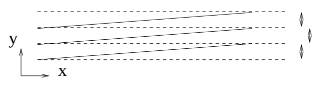
The second "natural" method to improve the way a cluster is stored is to make the distribution of the number of polygons referenced by the voxels as even as possible. And of course, choosing the same value for n_x, n_y, n_z — see section 1.3 — is very unlikely to meet this need. In [KS97], an heterogeneous griding scheme based on the box dimensions is proposed, and the improvements reported are pretty good. We tried a different method based on the notion of discriminant axis. A discriminant axis is an axis which succeeds in separating the objects when subdivided into slabs of the same width. For example on fig. 26(a), the y axis is more discriminant than x axis since along that latter direction the line-segments projections overlap one-another all the way along. A measure of this overlapping introduced in [Caz97] is the following:

Definition 4 *Let a one-dimensional arrangement of* n *line segments be a set of* n *line segments with distinct extremities on some line* L. *We define the* overlap number *of this arrangement as the sum over all the endpoints, of the number of line segments interiors these endpoints are contained in*

An example of arrangement of 3 line segments

$$\{(a_1, a_2), (b_1, b_2), (c_1, c_2)\}$$

is depicted on fig. 26(b), and the corresponding overlap number is $0 + 1 + 2 + 2 + 1 + 0 = 6$. Call o_x, o_y and o_z the overlap numbers of the three axis. The higher the overlap number o_i, the smaller the corresponding number of subdivisions n_i, with $i = x, y$ or z. A way to proceed is to set $n_i = n^{\beta_i}$ with $\beta_i = g(o_i)$, g a decreasing function from $I\!N \rightarrow [0, 1]$, and $\beta_x + \beta_y + \beta_z = 1$. This ensures exactly n voxels for a cluster of n objects and takes into account the ability of the i axis to separate the objects. But so far, we did not find any expression of β_i as a function of the o_i leading to satisfactory improvements of the rendering time.



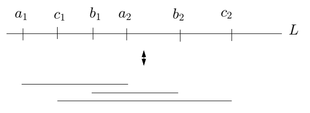


Figure 26: (a)Discriminant axis (b)Arrangement of line segments

7.4 Clustering by $2d$ sweeping

To remedy the problem mentioned in 7.1, we restrict the definition of a cluster as follows:

Definition 5 ***Cluster*** *Let* $\{b_1, \ldots, b_n\}$ *be the axis aligned bounding boxes of the objects of the scene processed. We define the bounding boxes intersection graph* IG *as the non directed graph whose nodes are the* b_i *and two nodes* b_i *and* b_j *are linked by an arc if the two bounding boxes intersect one another. A cluster is then defined as a connected component of this graph.*

The use of this definition is to provide a straightforward two step algorithm to compute the clusters by first looking for all the pairs of intersecting bounding boxes and second running a depth first search on the resulting IG graph. As far as we know, no equivalent data structure to the the interval tree (see [PS85, dB+96]) is known in dimension two, so that the first step can be done in sub-optimal time $O(n \log n + k')$ with k' the number of intersecting bounding boxes projections in the xy plane. Practically, it actually turned out that 3/4 of these intersections in the plane were denied along the z axis. The second step just requires $O(n + k)$ with $k \leq k'$ the number of intersecting bounding boxes in $3D$. As shown on figure 27(b) and 29, the resulting clusters are more tight.

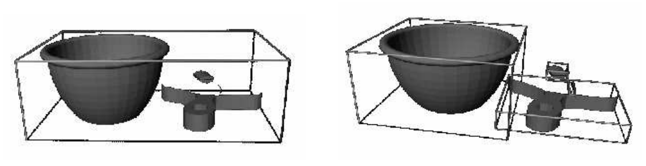


Figure 27: (a)(b)Kitchen stuff

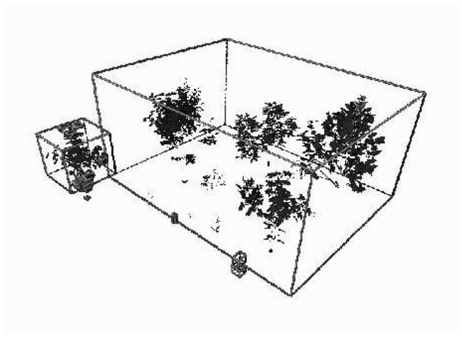


Figure 28: Huge tree leaves clusters

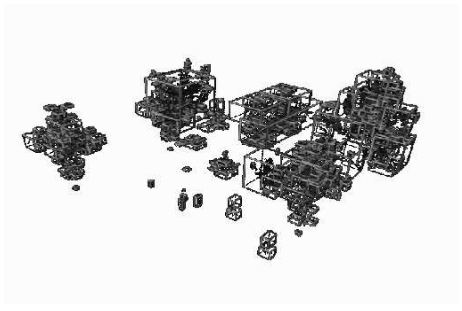


Figure 29: Tighter tree leaves clusters

7.5 Selected experimental results

The first improvement we made over the version of [CDP95] was to tune the subdivision parameter of the world cluster for a two-levels filter. For example for the kitchen scene it is easy to get a rendering time as good at the recursive grid that is around 45 seconds while using 3.99 mega-bytes instead of 7.87. On the opposite, restricting this resource to 4 mega-bytes for that data structure results in a performance of 103 seconds which is about 2.3 times slower than the HUG. The conclusions are the same for the Berkeley CS building, that is with the same memory requirement the HUG is between two and three times faster than the recursive grid. It should also be pointed out that for these scenes which are prone to clustering, using either clustering algorithm does not really matter.

On the opposite, for the garden and the city scenes where it was noticed that the original clustering algorithm was not performing that well, the situation is less favorable. The new clustering algorithm isolates houses in the city, sets of leaves in the trees, etc. But is a cluster of say 5 polygons still a cluster? The answer is probably not and due to the observation raised at the end of section 7.1 one could be tempted to pop up these polygons into the World grid ... which ends up being a uniform grid if there is no cluster left in the filter lower levels! This issue deserves more investigation.

Another problem is that as opposed to the initial algorithm that was performing an *in situ* clustering by just swapping the scene objects in an array, the new one needs more dynamic memory allocations for the line sweep and the graph depth-first search. And the resulting memory fragmentation turns out to require as much memory as the recursive grid.

8 Conclusion

The results achieved in this paper are threefold. First, we show that coming up with a precise average case analysis of the performances of a ray-tracer is a difficult question since the time consumed by the functions called most often is context-sensitive and depends within a factor five or more on the data structure these functions are called from. Second, we present experimental facts showing that the uniform grid which is known to perform pretty well for a wide set of problems such as the Delaunay triangulation computation, reporting red-blue line segments intersections as well as ray tracing or nearest neighbors computation, is outperformed by the recursive grid and the HUG in terms of data partitioning. Applied to ray-tracing this results in performances one order of magnitude better. Third, we exhibit an improved version of the hierarchy of uniform grids data structure which performs three times faster than the recursive grid for architectural scenes. This solution is particularly suitable for rendering models of several hundreds of megabytes and for which the recursive grid is too much memory consuming.

There are still important issues remaining. First, it would be important to have tools characterizing the ability of a particular model to be clustered and a first step in that direction is made in [CS97]. Second, the benefit of non axis-aligned bounding boxes as in [Wu92, GLM96] should be investigated further. Third it would be interesting to develop and analyze heterogeneous griding schemes. Fourth, our clustering algorithms may find interesting applications in the scope of hierarchical radiosity (see [SD95a]) or Geographic Information Systems.

Acknowledgment: The authors wish to thank the models providers as well as Mathieu Desbrun, Eric Haines and Stéphane Rivière for rereading this paper.

References

[A+94] D.S. Andrews et al. Further comparison algorithms for geometric intersection problems. In 6th *International Symposium on Spatial Data Handling*, 1994.

[AEII85] T. Asano, M. Edahiro, H. Imai, and M. Iri. Practical use of bucketing techniques in computational geometry. In G.T. Toussaint, editor, *Computational Geometry*. Elsevier, 1985.

[Caz97] F. Cazals. Combinatorial properties of one-dimensional arrangements. *Experimental Mathematics*, 6(1), 1997.

[CDP95] F. Cazals, G. Drettakis, and C. Puech. Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. In *EG'95*, Maastricht, 1995.

[CS97] F. Cazals and M. Sbert. Some integral geometry tools to estimate the complexity of 3*d* scenes. *In preparation*, 1997.

[dB+96] M. de Berg et al. *Computational geometry by example.* Stanford University Press, draft version, 1996.

[Dev86] L. Devroye. *Lectures notes on bucket algorithms.* Birkhauser, 1986.

[Dev88] O. Devillers. *Méthodes d'optimisation du tracé de rayon.* PhD Thesis - Université Paris Sud, 1988.

[GLM96] S. Gottschalk, M.C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *ACM Siggraph 96*, New Orleans, 1996.

[Hai96] E. Haines. Personal communication. 1996.

[JdL92] Erik Jansen and Win de Leeuw. Recursive ray traversal. *Ray Tracing News (available at nic.funet.fi:/pub/graphics/misc/RTNews)*, 5(1), 1992.

[KS97] K. Klimaszewski and T. Sederberg. Faster ray-tracing using adaptative grids. *IEEE Computer Graphics and Applications*, Jan-Feb:42–51, 1997.

[lBWY80] J l. Bentley, B. Weide, and A. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6, 1980.

[MMS94] J. Mitchell, D. Mount, and S. Suri. Query-sensitive ray shooting. In 10th *ACM CG*, pages 359–368, Stony Brook, 1994.

[NH84] J. Nievergelt and H. Hinterberger. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[Ś76] L. Sántalo. *Integral geometry and geometric probability.* Addison-Wesley, 1976.

[Sbe93] M. Sbert. An integral geometry based method for fast form-factor computation. *Computer Graphics Forum - Eurographics'93*, 12(3), 1993.

[SD95a] François X. Sillion and George Drettakis. Feature-based control of visibility error: A multi-resolution clustering algorithm for global illumination. In *SIGGRAPH 95*, Los Angeles, 1995.

[SD95b] P. Su and S. Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proc. of the* 11th *ACM Symp. on Computational Geometry*, pages 61–70, 1995.

[Wu92] X. Wu. A linear-time simple bounding volume algorithm. *Graphics Gem*, III, 1992.