Statistical optimization of octree searches

RENER CASTRO, THOMAS LEWINER, HÉLIO LOPES, GEOVAN TAVARES AND ALEX BORDIGNON

Department of Mathematics — Pontifícia Universidade Católica — Rio de Janeiro — Brazil
{rener, tomlew, lopes, tavares, alexlaier}@mat.puc--rio.br.

Abstract. This work emerged from the following observation: usual search procedures for octrees start from the root to retrieve the data stored at the leaves. But since the leaves are the farthest nodes to the root, why start from the root? With usual octree representations, there is no other way to access a leaf. However, hashed octrees allow direct access to any node, given its position in space and its depth in the octree. Search procedures take the position as an input, but the depth remains unknown. This work proposes to estimate the depth of an arbitrary node through a statistical optimization of the average cost of search procedures. Since the highest costs of these algorithms are obtained when starting from the root, this method improves on both the memory footprint by the use of hashed octrees, and execution time through the proposed optimization.
Keywords: ***Octree. Hashing. Quadtree. Geometric Modelling. Data Structures.***

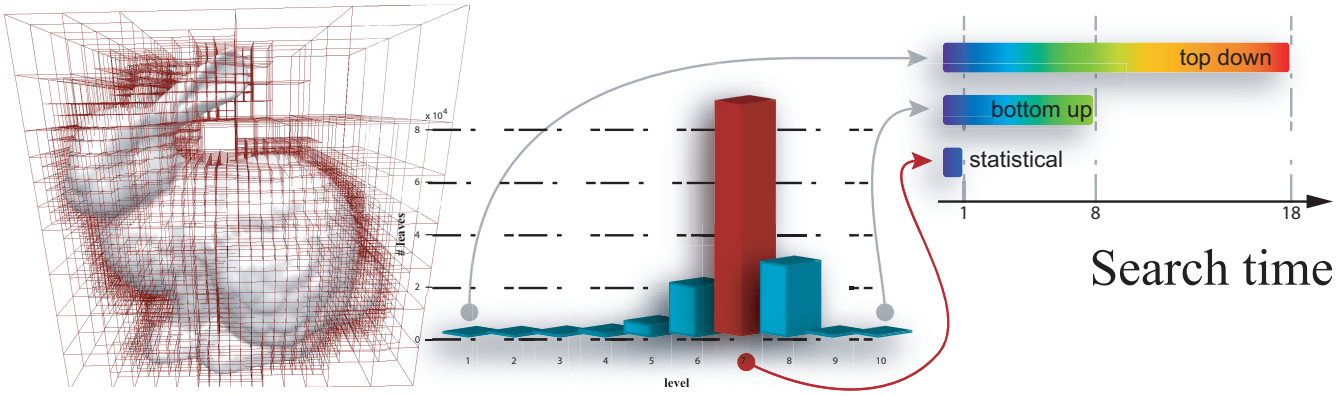


Figure 1: *Since the leaves are the farthest nodes to the root, why start from the root to look for the leaves? An optimal initial level greatly reduces the search costs: (left) an octree refined to separate each vertex of the Stanford bunny; (middle) the histogram of the number of octree leaves per depth; (right) cost of searching for a leaf starting from different depths.*

1 Introduction

The local geometry of discrete objects is generally based on groups of nearby elements. Therefore, Geometry Processing relies heavily on search procedures such as retrieving objects at or close to a given position. Some representations of discrete objects, such as grids or meshes, contain explicitly some of the results of these procedures. In that sense, they provide one extreme of the execution time/memory space trade off. Unstructured representations such as point sets stay on the other extreme.

In both situations, independent localization structures usually complement the object proper data structure. Those allow, for example, computing local properties of points sets such as normals or curvatures; triangulating sampled surfaces through local algorithms such as moving least-squares techniques, or global algorithms using Delaunay triangulations; testing collision in animation; representing local physical interaction in simulation and generating multi–resolution representations of discrete objects. Recently, many techniques use octrees to cluster point sets [10, 2] or render them [1, 3].

The classical search data structures are mostly based on hierarchical representations, since they reduce locations to a logarithmic complexity. They include octrees, kd–trees, multigrids, triangulation hierarchies and more general binary space partitions (BSP). This paper proposes an improvement of search procedures for octrees based on statistical modeling of nodes distribution. It works on previously known space–optimized representations, and improves the execution time of these procedures.

Preprint MAT. 09/06, communicated on May 14^{th}, 2006 to the Department of Mathematics, Pontifícia Universidade Católica — Rio de Janeiro, Brazil. The corresponding work was published in Computer Graphics Forum, volume 27, number 6, pp. 1557-1566. Blackwell, 2008.

Related works. We will focus here on efficient implementations of search methods in octrees. [8] provides a general overview of octrees and their applications. Of particular interest are memory–efficient representations of octrees. Among those, [4] introduced *linear octrees* which use the classical vector representation of fixed valence trees together with efficient codes for retrieving the position of a node from its index. Her representation is very compact but represents only full octrees, i.e. octrees with all the leaves at maximal depth. More recently, [12] extended the work of [5] with *hashed octrees* which represent an octree as a hash table of nodes, indexed by their Morton codes [7], as detailed in section 2 *Octree Representations*. This representation is both compact and allows direct access of an octree node from its Morton code.

Search procedures can benefit from the direct access facility provided by Morton codes. For full octrees, [9] provides a simple algorithm to generate the Morton codes of the adjacent neighbors. [11] improved the efficiency of the code computation using integer dilation. This optimized search has been further extended to other neighbors in [6, chap. III], starting from the same depth and computing the Morton codes of the parent or the children of a given node to complete the traversal. Our work further improves their approach by extending it to direct search and using a statistical estimation of the neighbors' depths.

Contributions. We propose improved search procedures for hashed octrees to benefit from the direct access offered by that representation. Compared to classical octree representations, it reduces both execution time and memory consumption. Compared to previous hashed octree operations, it maintains the same memory usage and saves execution time. Although the method relies on a simple statistical modeling, it improves on previous methods even in the worst case. Moreover, the method is not specific to dimension 3, and can be directly applied to quadtrees or higher dimensional 2^d-trees.

For the hashed octree introduced in section 2 *Octree Representations*, the retrieval of a node by its spatial position and its depth has an average constant execution time. The main idea for retrieving the leaf in a given position is to estimate its depth (see section 3 *Optimized Search*). This involves a simple statistical modeling (presented in section 4 *Statistical Modeling*), which also applies to and improves neighbor retrievals. The method improves searches by an average factor above 15, as shown by the experiments reported in section 5 *Experiments*.

2 Octree Representations

An *octree* is a hierarchical data structure based on the recursive decomposition of a 3D region (see Figure 2). A node of an octree represents a cube in that region. Each of the eight children of a node represents one octant of its parent, and the data is usually stored at the leaves. This hierarchy of subdivisions is commonly represented by a directed tree where each node is either a leaf or has eight children (this work actually includes octrees where a node can have less than eight children). We will consider here general octrees, sometimes called *variable resolution* octrees since they have leaves at different depths. This section recalls the basic implementations of octrees and search procedure. The reader will find a more exhaustive description in [8].

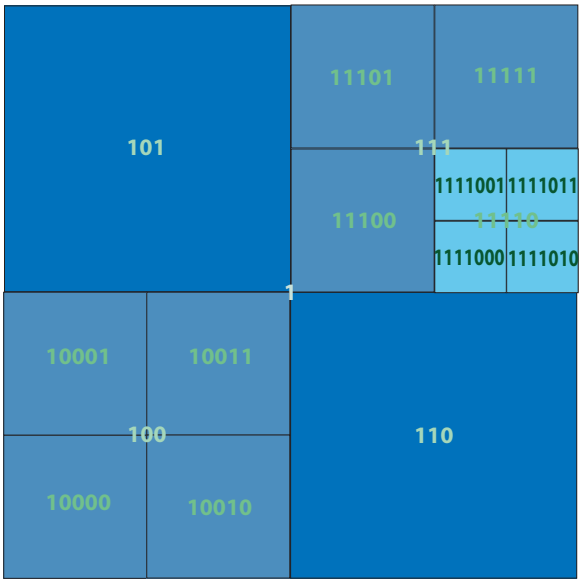
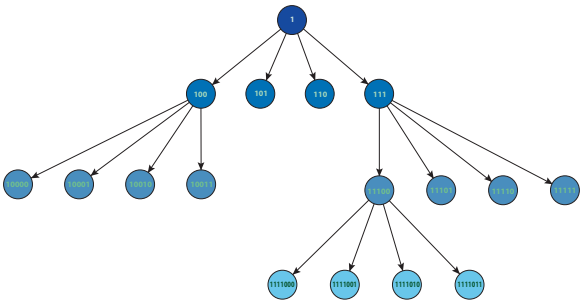


Figure 2: *Quadtree with the node keys, using the following orientation for the nodes* bottom-left, top-left, bottom-right, *and* top-right.

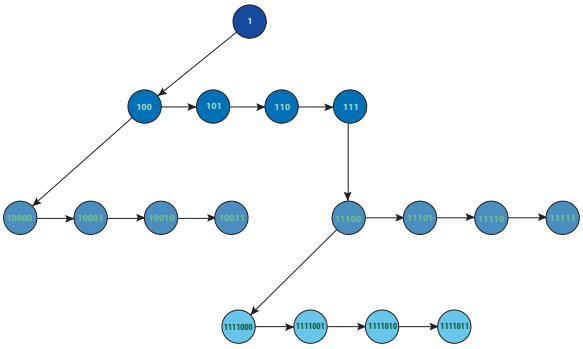
Classical octree structures. The two most common representations of octrees use pointers to allow variable resolutions of a tree. The first one relies on an exhaustive tree representation (see Figure 3(a)): each node has eight pointers, one for each of its eight children, and a pointer to the data. The children pointers are null for leaves, and the data pointer is null for intermediate nodes. The second one relies on the brother/child tree representation (see Figure 3(b)): some nodes have a pointer to its first child and to the next child of its parent, and a pointer for the stored data.

In addition, some implementations add pointers to the parent to accelerate bottom-up traversals. The second option uses less memory, but requires more execution time to search for a given leaf.

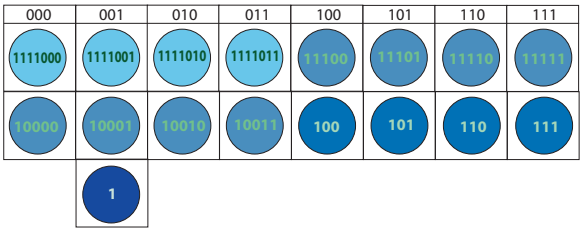
Hashed octrees. Another type of octree representation, more compact, stores the nodes in a hash table, which allows direct access to any node while avoiding explicit pointers for the octree hierarchy (see Figure 3(c)). This representation assigns to each node a *key*, which identifies it and serves for computing its address in the hash table. The key can be computed either from the position of the node inside the octree hierarchy through a systematic orientation of the octants of a node (see Figure 4), or from the coordinates of the node in space. In efficient schemes, the key can be equally computed



(a) Hierarchical representation.



(b) Brother/child representation.



(c) Hashed representation

Figure 3: *Three representations for the quadtree of Figure 2. The hash table uses the three least significant bits of the key:* $k \mod 2^3$. *The octree case is identical.*

by both methods. At the same, this allows time to identify the children of a node by the octant orientation for traversal algorithms, and to access a node directly from its position for search procedures.

Morton keys. To our knowledge, the most efficient key generation mechanisms use the codes of Morton [7]. The key $k(n)$ of a node n can be generated recursively from the octree hierarchy: the key of the root is 1, and the key of the child of n is the concatenation of $k(n)$ with the 3 bits of the child octant (see Figure 4). With that convention, the depth of n is $\lfloor \frac{1}{3} \log_2 (k(n)) \rfloor$ and the key of the parent of n is obtained by truncating the 3 least significant bits of $k(n)$: $k(n) \gg 3$ (shift right by 3 bits).

The key $k(n)$ can also be generated from the depth l of n and the position (x, y, z) of its center: assuming that the root is the unit cube $[0, 1]^3$, the key $k(n)$ is computed by interleaving the bits x_i, y_j, z_k, of x, y and z: $k(n) =$

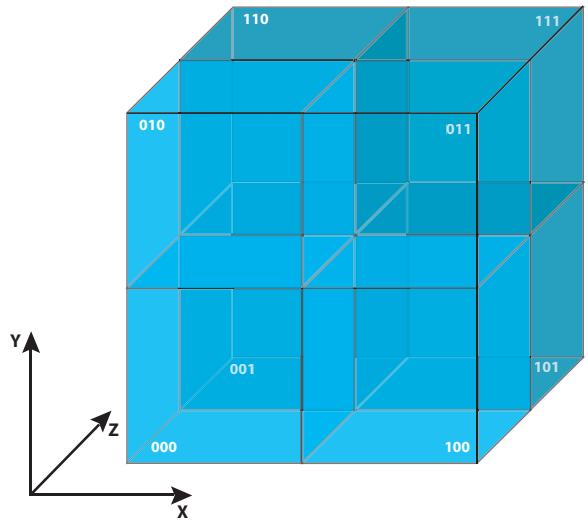


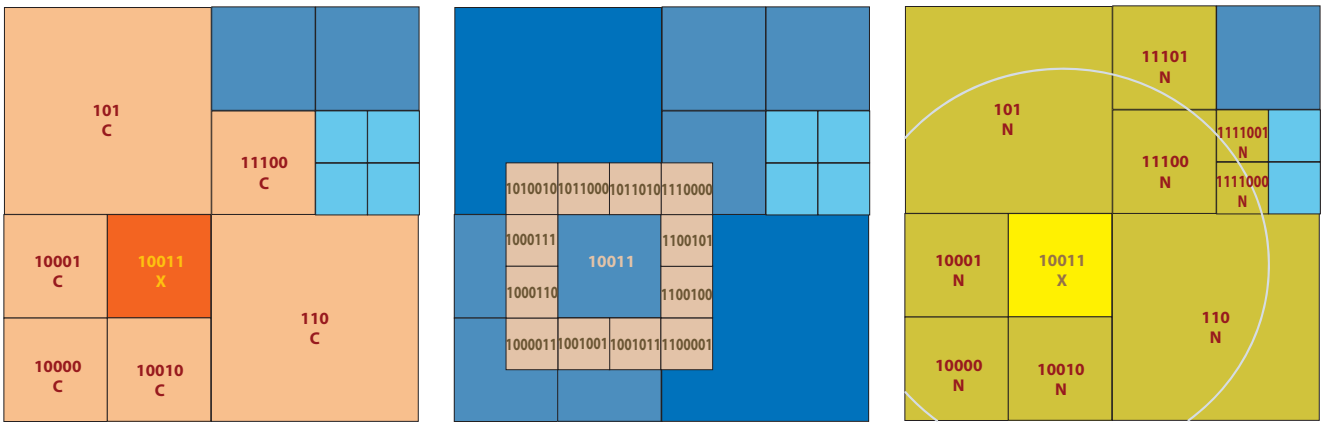
Figure 4: *Suffixes to append to the parent key to obtain the child keys.*

$1 x_l y_l z_l x_{l-1} y_{l-1} z_{l-1} \ldots x_1 y_1 z_1$ (the codes in the 2D case are drawn on Figure 2). Interleaving can be accelerated by integer dilation [11].

The hash function assigns to a node n the b least significant bits of its key: $k(n) \mod 2^b$. It tends to homogenize the hash table (see Figure 3(c)), in the sense that its entries are regularly distributed, especially when the octree is unbalanced. In this paper, we will assume that the hash table is homogeneous, and thus the access of a node by its key can be performed in constant time.

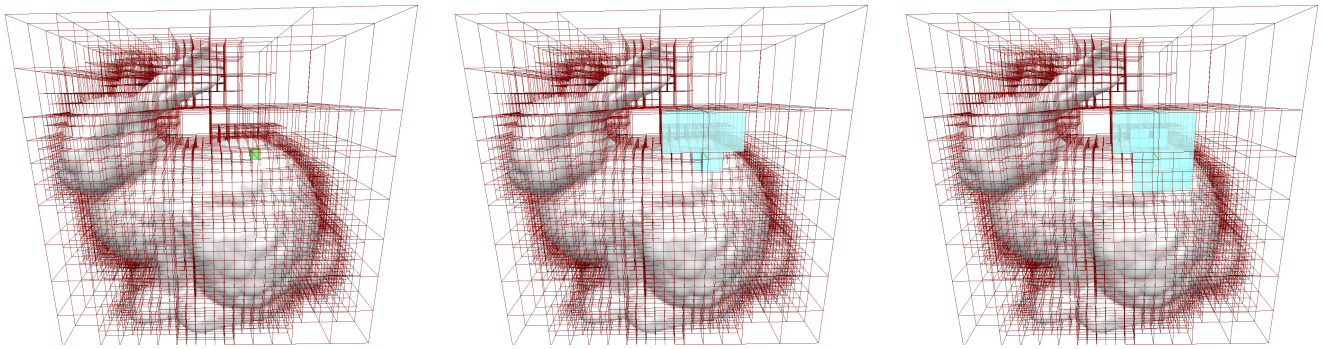
Classical search procedures. The search for the data associated to a spatial position p in an octree consists in finding the leaf containing p and retrieving its data. In pointer octrees, the only way to access a node is to start from the root and recursively choose the child whose octant contains p. This procedure has a complexity proportional to the depth of the found leaf. It is usually estimated as $\log_8(N)$, where N is the number of nodes of the octree, when the tree is balanced, and $O(N)$ in the worst case. This procedure is widely used for pointerless octrees. Observe that we can avoid storing the position and size of a node, since they can be deduced from the traversal.

Some other queries can be performed on an octree: finding the nodes adjacent to a given node n (see Figure 5(a)), which will be called *adjacent neighbors* hereafter, and the nodes which are within a given radius ρ of n (see Figure 5(c)), which will be called *inradius neighbors* hereafter. The usual algorithm also uses a top–down approach. In the first case, it recurses on all the children that intersect n, and in the second case the ones that intersect the ball of radius ρ centered at the center of n. These searches are illustrated on Figure 6.



(a) Adjacent neighbors of node 10011. (b) Adjacent nodes to 10011 of depth 3. (c) Neighbors of node 10011 in radius $\frac{1}{2}$.

Figure 5: *The adjacent neighbors search (a) and inradius neighbors search (c). When searching downward in the adjacency search (case 2), only the keys of the adjacent children are needed (b).*



(a) Direct search. (b) Adjacent neighbors search. (c) Inradius neighbors search.

Figure 6: *Illustration of the search procedures in 3D, on an octree adapted to the 34,834 vertices of the bunny point set.*

Algorithm 1 find(point p): find the leaf containing p.

```
1: compute the key k_max of p at maximal depth
2: compute the key k of p at depth l̂ using k_max
3: access the node n corresponding to k in the hash table
// Case 2: n is not a leaf
4: while n exists in the hash table do
5:     increase by one the depth of k using k_max
6:     access the child of n in the hash table with k
7: end while
8: retrieve the last valid access
// Case 3: n is below the leaf
9: while n does not exist in the hash table do
10:    decrease by one the depth of k
11:    access the parent of n in the hash table with k
12: end while
13: return n
```

3 Optimized Search

This work emerged from the following observation (see Figure 1): since the leaves are the farthest nodes to the root, why start from the root when looking for a leaf? The answer for the usual octree representation reduces to: there is no other way to access a leaf. However, with hashed octrees, a leaf can be directly accessed by its Morton code, which depends only on the leaf position and depth. The position is known when looking for a leaf, but its depth may not. The following algorithms describe how to retrieve a leaf (or more generally a node) from its position and an *estimated depth* $\hat{l}$. The next section will describe how to estimate the depth of a leaf.

(a) Direct Search

The direct search procedure of Algorithm. 1:find is a straightforward application of this idea: In order to find the (unique) leaf containing a point p, the algorithm generates the key $k_l(p)$ of p at the estimated depth $l = \hat{l}$. Looking for the node $n_l(p)$ corresponding to that key, three situations may occur:

1. The node $n_l(p)$ is a leaf: the algorithm thus returns

The corresponding work was published in Computer Graphics Forum, volume 27, number 6, pp. 1557-1566. Blackwell, 2008.

$n_l(p)$.

2. The node $n_l(p)$ is not a leaf: it means that the estimated depth l is too low, and l is incremented until $n_l(p)$ points to a leaf, which is returned
3. There is no node corresponding to $n_l(p)$: it means that the estimated depth l is too high, and l is decremented until $n_l(p)$ points to a node. The first node is then a leaf, and the algorithm returns it.

Observe that if the estimated depth is zero, the search starts from the root and operates with the second case, which corresponds to the usual hierarchical search. Moreover, the key does not need to be recomputed in the third case, since the key of $n_{l-1}(p)$ can be deduced from the key of $n_l(p)$ by truncating the 3 least significant bits. Similarly in the second case, the key at depth $l+1$ can be generated from the key at the maximal depth $k_{max}(p)$ as above : $k_{l+1}(p) = k_{max}(p) \gg 3 \cdot (l_{max} - l - 1)$, which is faster than generating a new key.

(b) Adjacent Neighbors Search

The procedure to find the adjacent neighbors of a node n (see Figure 5(a)) resembles the direct search, although it must return a variable number of leaves. We considered two options to optimize the adjacent node search. First, we generated the keys of the adjacent neighbors at an optimal depth $\hat{l}$ and followed the octree hierarchy toward the leaves. However, this requires many hash table accesses.

Algorithm 2 adjacent(n): find the adjacent neighbors of n.

```
put into set s the adjacent neighbors n_i at depth l(n)
for all nodes n_i in s do
    in the hash table, find the node n_i
  // Case 1: n_i is a leaf
    if n_i exists in the hash table and is a leaf then
        add n_i to the result set r
  // Case 2: n_i is not a leaf
    else if n_i exists in the hash table then
        insert into s the children of n_i adjacent to n
  // Case 3: n_i is below the leaf
    else
        call find(n_i) using depth min{l̂, l(n)}
        add the found node to the result set r
    end if
end for
return the result set r
```

We thus developed a second option for the neighbor search, presented in Algorithm. 2:adjacent: we generate the keys of the 26 nodes adjacent to n, with the same depth $l(n)$. Without statistical optimization, the algorithm would follow the three cases of the direct search, with the following modification of the second case, in the spirit of [6]:

For the second case, several keys $k(n_i)$ may be generated at each increment of the depth: there is only one adjacent $(l-1)$–neighbor if n_i is in the diagonal of n, two if n_i shares an edge with n and four if n_i shares a face with n, as illustrated on Figure 5(b) for the 2D case. The keys of these $(l-1)$–nodes are systematically generated using a small lookup table, and the algorithm recurses on each of these adjacent neighbors of depth $(l-1)$.

Since an error of depth estimation in the second case may result in generating many unused keys, we maintain the above hierarchical search starting from the 26 adjacent nodes in the second case. However, the third case can be optimized in a similar way to the leaf search: if a neighbor of n has a lower depth than n, we search for it directly at the estimated depth $\hat{l}$, if lower than the depth of n. This avoids many intermediate lookups in the hash table. Moreover, as opposed to our first option, it benefits from the fact that, if node n is already at the deepest level of the octree, at least 7 of its adjacent node have the same depth. This optimization is actually effective, as shown in section 5 *Experiments*.

(c) Inradius neighbors Search

The search for nodes within a given radius of n mimics the adjacent neighbor search. The only difference is again in the second case, since there are eight neighbors of depth $(l-1)$ for each l–neighbor (instead of 1, 2 or 4), which involves more key generation and hash table access than the adjacent neighbor search (see Figure 5(c)). Moreover, if the radius is greater than $2^{-\hat{l}}$, some inradius neighbors may not be adjacent to n and the initial set $\{n_i\}$ of inradius neighbors must contain more than 26 nodes. We propose here the same optimization as for the adjacent node, which restricts it to the third case of the algorithm and gets similar results.

4 Statistical Modeling

The above algorithms rely on an estimated depth $\hat{l}$ to look for the leaf. If we set this depth to zero or l_{max}, the algorithms always start from the root: the direct and the neighbor search behave like the classical octree traversal, and the adjacent neighbor search enhances classical procedures only by avoiding geometrical tests. As we mentioned earlier, since leaves are the farthest nodes from the root, $\hat{l} = 0$ corresponds to the worst case (maintaining $\hat{l}$ below the maximal depth).

We will now look for a simple statistical model for $\hat{l}$ in order to optimize the cost of the search procedures. The computation of $\hat{l}$ requires a small time overhead during the octree construction, and can be updated dynamically. Observe that since the worst case $\hat{l} = 0$ is also the most widely used, even without any optimization for $\hat{l}$ our method improves on previous works.

Cost model. The total cost of the search depends on how many times, on average, each leaf n will be looked for. We will denote this number by $f(n)$. For example, if we look for every position in the domain of the octree, $f(n)$ is proportional to the size of n, which is a power of its depth: $f(n) \backsim 2^{3(l_{max}-l(n))}$. If we look for some data related to the octree structure, such as internal collision detection, we may choose $f(n) \backsim 1$.

Point Set	# Nodes	Deepest Level	Memory			Build			Optimization		Number of Searches	
			8_{ptr}	BC_{ptr}	Hash	8_{ptr}	BC_{ptr}	Hash	Time	Level	Point	Leaf
	(x1000)		(MB)	(MB)	(MB)	(msec)	(msec)	(msec)	(msec)		(x1000)	(x1000)
Sphere	4.4	6	0.2	0.1	45	5.6	5.4	11.1	7.8	4	0.9	3.8
Bunny	144	9	6.9	3.4	50	224	215	187	14.2	7	35	126
Maracanã	190	21	9.1	4.6	51	268	257	230	27.1	21	10	167
David head	395	13	19	9.5	51	726	682	548	32.4	9	100	346
Pig	401	11	19	10	51	772	725	583	38.2	10	100	351
CSG	529	19	25	13	52	819	768	642	50.7	8	82	463
Volcano	551	21	26	13	52	1 043	967	795	50.2	9	133	482
Fighter	723	14	35	17	54	1 584	1 509	1 211	58.6	9	257	632
Deltao	1 210	20	58	29	60	2 466	2 119	1 695	116	12	208	1 058
David	1 534	21	74	37	62	3 124	2 819	2 379	135	10	350	1 342
Rnd Sphere	2 883	18	138	69	82	5 756	5 100	4 053	268	9	500	2 523
Dragon	3 060	21	147	73	85	5 634	5 280	4 286	314	10	438	2 678
Happy	3 617	21	174	87	96	7 227	6 627	5 361	377	10	544	3 165
Blade	5 412	16	260	130	134	11 641	10 079	8 384	518	10	883	4 735
Chair	9 255	18	444	222	223	34 471	24 833	20 076	930	11	1 669	8 098
Rnd Cube	15 675	12	752	376	376	45 867	39 380	31 012	1 202	8	4 000	13 715
Thai Statue	22 296	20	1070	535	535	155 803	106 509	81 446	2 229	12	5 000	19 509
	weighted average		**632**	**316**	**321**	**68 436**	**49 213**	**38 084**	**354**			

Table 1: *Building and preprocessing times on different point sets: varying density (Maracanã, David head, Pig, Fighter, Deltao), volumetric (Volcano, Fighter, Deltao, Cube). The building of hashed octree is slightly faster than pointer octrees. The memory consumption is equal for both cases on big point sets. The preprocessing consists in computing the optimal level and lasts less than one percent of the build time. The averages are weighted by the octree size. The number of searches are reported for each point set according to the simulated frequency model (the grid search has a constant number of searches 2097).*

Our main assumption is that $f(n)$ depends only on the depth of n: $f(n) = f_l$. In practice, this means that the octree is used without geometric bias. Observe that this is the case for the two examples mentioned above. The average cost of the search is then the sum on each leaf n of the cost of searching that leaf multiplied by $f(n)$: $cost(\hat{l}) = \sum_{n \in \text{leaves}} f(n) \cdot cost_{\hat{l}}(n)$. Since the cost of searching a leaf depends only on its depth, and denoting by p_l the number of leaves of depth l, we can write it as a sum indexed by the depth:

$$cost(\hat{l}) = \sum_l p_l \cdot f_l \cdot cost_{\hat{l}}(l) \; .$$

Direct Search. As mentioned earlier, we will assume that the hash table is homogeneous, and that accesses to it are made in constant time. We model the cost of the three cases of the direct search as follows: if the depth is correctly estimated (case 1), the node is returned directly with a constant time c. Moreover, since we use a unique key generation for all the cases, the costs of computing the key of a parent or a child are equal (cases 2 and 3). Since there are $|l - \hat{l}|$ generated keys when searching a node at depth d, the cost of that search is the sum of a constant cost c for the key generation plus an overhead proportional to the difference $|l - \hat{l}|$:

$$cost\left(\hat{l}\right) = c + \sum_{l<\hat{l}} p_l \cdot f_l \cdot \left(\hat{l} - l\right) + \sum_{l>\hat{l}} p_l \cdot f_l \cdot \left(l - \hat{l}\right)$$

We look for the value of $\hat{l}$ that minimizes that cost. To do so, we construct a differentiable cost function $\text{cost}(\hat{l})$, and look for a zero of its derivative, as detailed in the appendix:

$$\frac{\mathsf{d}\text{cost}}{\mathsf{d}\hat{l}}\left(\hat{l}\right) = \int_0^{\hat{l}} p(l) f(l) \; \mathsf{d}l \quad - \int_{\hat{l}}^{\infty} p(l) f(l) \; \mathsf{d}l$$

The optimal cost $\hat{l}$ is thus the median of the depths of the octree leaves weighted by the number of times they are looked for: $\hat{l}$ must satisfy $\sum_{l<\hat{l}} p_l \cdot f_l = \sum_{l>\hat{l}} p_l \cdot f_l$. This median can be dynamically computed during the octree creation by maintaining a small histogram of the leaves depth. It can also be approximated by the weighted mean, which reduces the (already small) preprocessing time.

Adjacent and Inradius Neighbors Search. The estimation of the depth for the other searches will differ from the previous one in only one aspect: the statistical model or f_l depends on the node n whose neighbors we are looking for. Whereas p_{l_n} depends only on the depth $l_n = l\,(n)$, the distribution p_l depends on n. The optimum thus depends on each node n. For extensive use of the octree, this optimum can be computed for each node, and requires storing one more integer per node. However, since we use this optimum only in the third cases of the algorithms, where the neighbor has a lower depth, we may consider that the distribution of p_{l_n} and p_l are independent. Therefore, we approximate the optimal depth for all nodes by the optimal depth of the direct search.

The corresponding work was published in Computer Graphics Forum, volume 27, number 6, pp. 1557-1566. Blackwell, 2008.

Point Set	Direct Search				Find Adjacent Neighbors				Find Inradius Neighbors			
	8_{ptr}	BC_{ptr}	Unopt	Ours	8_{ptr}	BC_{ptr}	Unopt	Ours	8_{ptr}	BC_{ptr}	Unopt	Ours
Sphere	2.17	7.24	11.47	0.58	23.10	23.85	41.55	7.63	28.38	25.39	41.49	0.71
Bunny	2.86	11.28	18.67	0.75	38.34	35.93	59.70	8.70	36.70	35.44	59.26	0.73
Maracanã	6.53	31.21	53.47	0.60	58.98	55.79	95.70	9.12	77.75	76.31	136.91	33.72
David head	3.29	12.98	21.27	0.64	43.57	40.88	68.69	9.43	42.61	40.93	70.30	1.39
Pig	3.71	15.40	25.65	0.71	46.46	43.90	74.64	11.73	53.53	51.59	91.88	10.67
CSG	3.30	13.34	21.98	0.79	40.60	37.75	63.22	9.76	38.79	37.18	62.81	4.57
Volcano	3.28	13.42	22.73	0.71	43.04	40.37	68.74	8.83	40.18	39.43	69.16	2.78
Fighter	3.89	14.82	24.53	1.06	54.70	50.38	86.60	12.88	51.81	49.92	87.79	2.18
Deltao	4.47	18.63	31.64	1.65	49.07	45.56	77.67	11.30	180.30	169.13	332.11	129.28
David	3.58	15.04	26.54	0.87	46.13	42.64	73.64	10.39	43.71	42.17	74.79	11.88
Rnd Sphere	4.50	16.42	27.92	1.59	46.51	42.55	73.15	14.45	42.43	41.42	72.71	9.02
Dragon	3.81	15.88	27.43	1.17	45.53	42.80	74.31	12.79	45.88	44.44	79.03	11.67
Happy	3.80	16.17	28.69	1.21	46.30	43.05	75.37	13.18	48.60	46.39	83.23	15.46
Blade	3.65	15.25	27.98	1.27	44.05	40.89	73.09	13.11	43.84	41.54	75.27	11.45
Chair	3.83	16.38	31.41	2.15	46.56	43.43	80.89	19.34	52.92	50.80	96.60	24.48
Rnd Cube	4.18	14.76	33.54	3.79	51.36	46.69	95.21	27.44	45.34	44.04	84.09	0.96
Thai Statue	3.93	17.38	37.51	3.13	49.99	47.58	94.86	22.43	79.50	77.18	163.46	53.56
average	**3.81**	**15.62**	**27.79**	**1.33**	**45.55**	**42.59**	**75.12**	**13.09**	**56.02**	**53.72**	**98.88**	**19.09**
gain	**x 3.8**	**x 15.7**	**x 26.9**	x 1	**x 3.8**	**x 3.6**	**x 6.2**	x 1	**x 14.5**	**x 13.8**	**x 24.2**	x 1

Table 2: *Execution time (in microseconds) of our optimized searches, compared to classical top-down searches on classical 8-pointers (8_{ptr}) brother/child (BC_{ptr}) and hash (Unopt) representations. Our optimized direct search returns* 15 *times faster than brother/child searches, and between* 10% *(random volume) and* 1090% *faster (maracanã) than classical representations. Each time reported is the average of the searching time for each point of the set. The gain corresponds to the ratio of the pointer search times over our search times.*

5 Experiments

We tested our optimization on octrees tracking geometrical objects in two settings: raw searches inside the octree and within a collision detection context. The tests were performed on a $3GHz$ Pentium IV machine.

Raw search. We test each of the three proposed search procedures on octrees adapted to a set of points: the nodes are subdivided until they contain less than one point of the set or if their level is maximal (21 since we work with 64-bits keys). The octree build is slightly faster on hashed octrees, and the optimization time is around one percent of the build time (see Table 1). In our tests, we fix the hash function to return the 21 least significant bits of the key. This generate hash tables of sizes similar to the memory consumption of pointer representation on the big point sets we used. We compare our results with the classical top-down search procedures on classical 8-pointers representation and brother/child representation and hashed representation (see Table 2). Searches on unoptimized hashed representation are slower since the access to the child of a node requires an access to the hash table. Our algorithm performs in average 15 times faster than the pointer representation. For the other searches, the performance of our method remains several times faster.

Moreover, we compare different optimizations depending on the *a priori* access frequency $f(n)$ to a leaf of level n, as described in Section 3. We experiment with three frequency model: searching for each point of the set ($f_{points}(n) = \#points \in n$), for each leaf of the octree ($f_{leaves}(n) = 1$) and for each voxel of a regular grid 127^3 ($f_{grid}(n) = 2^{3(7-l(n))}$). This optimization is experimentally validated on Table 3: the searches return significantly faster when we choose the correct frequency model for the optimization.

	f_{points}	f_{leaves}	f_{grid}
Search all Points	1.30	1.33	2.45
Search all Leaves	0.95	0.93	1.52
Search on a Grid	1.70	1.66	0.89

Table 3: *Average on all the point sets of the execution times (in microsecons) of the direct search, simulating different frequency models (lines). Optimizing according to the correct frequency model (diagonal terms) improves the search performance between* 2% *(points) and* 92% *(grid). The number of searches in each case is reported on Table 1.*

Collision detection. We also tested our optimized search on a real context of collision detection: we simulated a small ball kicking inside a 3D mesh. Here the octree is adapted to the triangles of the mesh, and each leaf is eventually associated with one of the mesh triangles intersecting it. At each iteration, the algorithm searches the octree for the leaf containing the ball. If there is a triangle associated with that leaf, the ball changes direction according to Snell-Descartes reflection. Since these operations use direct searches intensively, our optimized octree outperforms the classical representation (see Table 4).

Surface	# trigs ($\times 10^3$)	# nodes ($\times 10^3$)	Frame rate 8_{ptr}	Frame rate BC_{ptr}	Frame rate Ours
Sphere	2	4 432	11.3	9.0	60.0
Bunny	69	16 671	7.0	5.0	20.4
David	700	11 236	4.6	3.2	7.4
Dragon	871	23 904	5.0	4.1	7.5
Buddha	1 088	19 525	3.9	2.9	4.1
	average		**x 2.5**	**x 3.3**	

Table 4: *In a collision detection environment where a ball kicks into a 3D surface, our optimization improves the frame rate (in Hz), including the whole pipeline from the ball position computation to the rendering.*

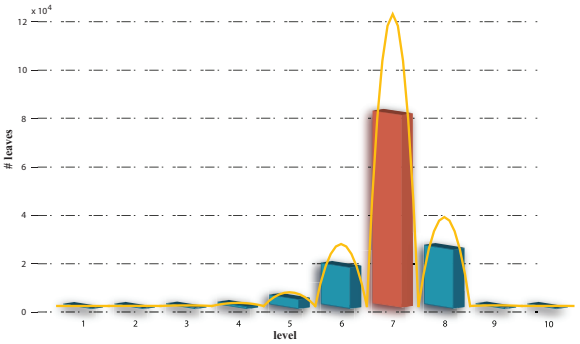


Figure 7: *A probability density $p(l)$ of the number of leaves p_l of depth l of the bunny.*

6 Conclusions

In this work, we introduced a statistical optimization for octree searches. The optimization applies to pointer–less octrees, such as hash table representations. In particular, these representations allow a direct access of a node from its key, and replace geometrical tests by bitwise key manipulations. Within this context, we proposed to search a node directly at its estimated level, instead of starting from the root. The resulting algorithms combine the memory efficiency of the hashed octree with an execution time performance around ten times less than a classical implementation. The proposed strategy works particularly well on geometric data with multiple level of details (like the Maracanã model), but has limited, although positive gain on data with depth distributions with high variance (like random volume models). Mixed statistical models may extend this work to more general geometric data.

References

[1] M. Botsch, A. Wiratanaya and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Eurographics Workshop on Rendering*, pages 53–64, 2002.

[2] T. Boubekeur, W. Heidrich, X. Granier and C. Schlick. Volume–Surface Trees. *Computer Graphics Forum*, 25(3):399–406, 2006.

[3] C. Dachsbacher, C. Vogelgsang and M. Stamminger. Sequential point trees. In *Siggraph*, pages 657–662. ACM, 2003.

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

[5] A. Glassner. Space Subdivision for Fast Ray Tracing. *Computer Graphics & Applications*, 4(10):15–22, 1984.

[6] N. A. Gumerov, R. Duraiswami and E. A. Boroviko. Data Structures, Optimal Choice of Parameters, and Complexity Results for Generalized Multilevel Fast Multipole Methods in d Dimensions. Technical report, University of Maryland, 2003.

[7] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, 1966.

[8] H. Samet. *Applications of spatial data structures*. Adison–Wesley, Reading, 1989.

[9] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *Computer Vision, Graphics and Image Processing*, 55(3):221–230, 1992.

[10] S. Schaefer and J. Warren. Adaptive Vertex Clustering Using Octrees. *Geometric Design and Computing*, pages 491–500, 2003.

[11] L. Stocco and G. Schrack. lnteger Dilation and Contraction for Quadtrees and Octrees. In *Communications, Computers and Signal Processing*, pages 426–428. IEEE, 1995.

[12] M. S. Warren and J. K. Salmon. A parallel hashed octtree N-body algorithm. In *Supercomputing*, pages 12–21. IEEE, 1993.

The corresponding work was published in Computer Graphics Forum, volume 27, number 6, pp. 1557-1566. Blackwell, 2008.

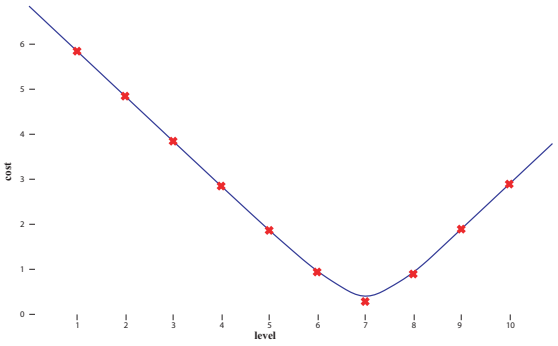


Figure 8: *Discrete cost function* $cost(\hat{l})$ *and differentiable cost function* $\mathrm{cost}(\hat{l})$ *for the bunny: the differentiable cost has a unique minimum, whose integer part is the minimum of the discrete cost.*

A Minimization of the cost

We aim at minimizing the discrete cost function $cost(\hat{l})$ defined in section 4 *Statistical Modeling* (see Figure 8):

$$cost\left(\hat{l}\right) = c + \sum_{l<\hat{l}} p_l \cdot f_l \cdot \left(\hat{l} - l\right) \sum_{l>\hat{l}} p_l \cdot f_l \cdot \left(l - \hat{l}\right) +$$

To do so, we consider a differentiable function cost interpolating the discrete cost function: $\forall \hat{l} \in \mathbb{N}, \mathrm{cost}(\hat{l}) = cost(\hat{l})$. From the intermediate value theorem, the level $\hat{l}_m$ including the lowest cost $cost(\hat{l})$ is at a distance less than 1 to a local minimum $\mathrm{cost}(l_m)$ of the differentiable function cost: $\left|\hat{l}_m - l_m\right| < 1$. We will now construct the differentiable function cost in order to have generically only one minimum l_m, and we will round it to optimize the cost function.

We define $\mathrm{cost}(\hat{l})$ by the following integral :

$$\mathrm{cost}(\hat{l}) = c + \int_0^{\hat{l}} p(l) f(l) \left(\hat{l} - l\right) \mathrm{d}l + \int_{\hat{l}}^{\infty} p(l) f(l) \left(l - \hat{l}\right) \mathrm{d}l$$

Considering $p(l)$ as a probability density for p_l and $f(l)$ as a density for f_l, the differentiable function cost interpolates the discrete cost function. As a canonical construction, we define p_l by parts on each integer interval $[l, l+1]$. Each part is a parabola of area $p(l)$ canceling at the interval bounds (see Figure 7). With a similar construction for f_l, the functions to be integrated in the definition of cost are continuous, and thus cost is a differentiable function (see Figure 8). For generic distributions p_l and $f(l)$, this function has a unique local minimum l_m, which can be computed by differentiation:

$$\begin{aligned}
\frac{\mathrm{d}\mathrm{cost}}{\mathrm{d}\hat{l}}\left(\hat{l}\right) &= \frac{\mathrm{d}}{\mathrm{d}\hat{l}}\left(c + \int_0^{\hat{l}} p(l) f(l) \left(\hat{l} - l\right) \mathrm{d}l + \right. \\
&\qquad \left. \int_{\hat{l}}^{\infty} p(l) f(l) \left(l - \hat{l}\right) \mathrm{d}l\right) \\
&= \frac{\mathrm{d}}{\mathrm{d}\hat{l}}\left(\hat{l} \cdot \int_0^{\hat{l}} p(l) f(l) \,\mathrm{d}l + \right. \\
&\qquad - \int_0^{\hat{l}} p(l) f(l) \cdot l \,\mathrm{d}l + \\
&\qquad \int_{\hat{l}}^{\infty} p(l) f(l) \cdot l \,\mathrm{d}l + \\
&\qquad \left. -\hat{l} \cdot \int_{\hat{l}}^{\infty} p(l) f(l) \,\mathrm{d}l\right) \\
&= \int_0^{\hat{l}} p(l) f(l) \,\mathrm{d}l + \hat{l} \cdot \left(p(\hat{l}) f(\hat{l})\right) - p(\hat{l}) f(\hat{l}) \cdot \hat{l} \\
&\quad - \left(p(\hat{l}) f(\hat{l}) \cdot \hat{l}\right) - \int_{\hat{l}}^{\infty} p(l) f(l) \,\mathrm{d}l - \hat{l} \cdot \left(-p(\hat{l}) f(\hat{l})\right) \\
&= \int_0^{\hat{l}} p(l) f(l) \,\mathrm{d}l - \int_{\hat{l}}^{\infty} p(l) f(l) \,\mathrm{d}l
\end{aligned}$$

Since this minimum is unique, it must be close to the minimum of the discrete cost (see Figure 8).