Fast generation of pointerless octree duals

THOMAS LEWINER[1], VINÍCIUS MELLO[2], ADELAILSON PEIXOTO[3], SINÉSIO PESCO[1] AND HÉLIO LOPES[1]

[1] Department of Matematics — Pontifícia Universidade Católica — Rio de Janeiro — Brazil
[2] Institute of Mathematics — Universidade Federal da Bahia — Salvador — Brazil
[3] Institute of Mathematics — Universidade Federal de Alagoas — Maceió — Brazil

Abstract. Geometry processing applications frequently rely on octree structures, since they provide simple and efficient hierarchies for discrete data. However, octrees do not guarantee direct continuous interpolation of this data inside its nodes. This motivates the use of the octree's dual structure, which is one of the simplest continuous hierarchical structures. With the emergence of pointerless representations, with their ability to reduce memory footprint and adapt to parallel architectures, the generation of duals of pointerless octrees becomes a natural challenge. This work proposes strategies for dual generation of static or dynamic pointerless octrees. Experimentally, those methods enjoy the memory reduction of pointerless representations and speed up the execution by several factors compared to the usual recursive generation.
Keywords: ***Octree. Dual. Hash Table.***

1 Introduction

Hierarchical structures are widely used to store discrete geometrical data. In particular, regular hierarchies like octrees [13] are fundamental ingredients in several geometry processing applications. However, octrees are *discontinuous* structures, in the sense that information in a refined node may not be directly accessible to a neighbor node. Defining data interpolation inside each node is thus not always continuous. This discontinuity makes the use of octree very delicate for certain applications [20, 17], such as adaptive extraction of isosurface [11]. A simple solution that preserves the simplicity and regularity of octrees but remains continuous is the *octree dual*, which is already widely used [7, 15, 12, 16]. This work introduces fast algorithms for the generation of dual octrees.

The regularity of the octree, subdividing in the middle independently of the data, reduces the adaptability of the structure, but it provides several advantages in terms of ease of use and performance. Classical representations of octrees use pointers, which leads to large memory footprint and random memory access during traversal. This is particularly inefficient, but can be avoided if using pointerless representations [4, 13]. Such representations associate to each node of the octree a unique key [10, 19], and the traversal operations resume to key manipulations that can be performed in local memory [14, 5, 2]. This work proposes to generate the dual of an octree using such key manipulations, enjoying the reduced memory footprint of pointerless representations and improving the execution time by several factors.

Preprint MAT. 02/10, communicated on April 25^{th}, 2010 to the Department of Mathematics, Pontifícia Universidade Católica — Rio de Janeiro, Brazil. The corresponding work was published in the proceedings of SGP 2010: Computer Graphics Forum, volume 29, number 5, Blackwell 2009.

Related works. Dual octrees have been first used in geometry processing for isosurface extraction [7, 15, 12, 11]. Since it is a continuous structure, usual trilinear interpolation [15], higher order interpolation [7] or finite element method [16] fit nicely, while preserving the adaptability of the octree [12]. All those works use a recursive generation of the dual, which is recalled in Section 3. Recently, León *et al.* [9] proposed a data structure, which preprocesses the octree for dual generation, storing an extra marker per leaf (see Section 3). This allows enumerating the interior vertices and generating the dual from those. Our proposal also builds the dual cells from the octree vertices, but reduces the execution time using key manipulations and is also able to avoid preprocessing and extra memory usage.

Pointerless representations of octrees [4] have become popular for sparing memory [3] and for their ability to work on parallel [21] and GPU architectures [1]. The use of Morton codes [10, 19] for indexing the octree (see Section 2) is widely used, since they allow for efficient manipulation as dilated integers [14, 18] and optimized search [5, 2].

Contributions. This paper introduces algorithms to efficiently generate dual of pointerless octrees. Our method enjoys the possibility offered by such representations to access the octree nodes directly, instead of following the subdivision hierarchy. More precisely, we propose two strategies (see Section 4). The first one stores the interior vertices of the octree during preprocessing and then generates the dual volumes from those vertices. This is efficient for static octree since it factors the vertex generation, but require an extra memory to store the vertices. The second strategy decides during the octree leaves' traversal which vertices to process. This removes the preprocessing and avoids storing the vertices and fits nicely for dynamically adapted octrees. We also

propose simple key manipulations on dilated integers [18] to efficiently represent the vertices of the octree, and an optimized search to retrieve the dual volumes from those vertices. Those methods accelerate the dual generation by an average factor above 3, as shown in the experiments of Section 5.

For the sake of clarity, the figures of this work represent quadtrees, while the text and the algorithms refer to the 3D case. All the results naturally generalize to any dimension. The notation $\overline{abc}$ means the cyclic repetition $abcabcabc\ldots$.

2 Octrees and their Representations

An *octree* [13] is a hierarchical data structure based on recursive decomposition of an initial cube in 3D, where each node of the hierarchy represents a part of the initial cube as follows. The root node of an octree represents the whole cube. Each node may be subdivided, generating eight children, each of which represents one octant (see Figure 1). Usually, a piece of data is associated only to the un–subdivided nodes, called leaves. The *depth* of a node n is the number of divisions between the root and n.

Classical octree structures. The two most common representations of octrees use pointers to represent the subdivisions hierarchy. The first one relies on an exhaustive tree representation (see Figure 2(top)): each node has eight pointers, one for each of its children, and a reference to the associated data. The second one, called sibling/child representation, stores for each node a pointer to its first child and to the next child of its parent, and a reference to the data.

Besides, some implementations add pointers to the parent to accelerate bottom-up traversals. Observe that the second option uses a fourth of the memory (2 pointers per node, instead of 8), but requires in average 4.5 more pointer dereferencing to access a given node.

Hashed octree. Another type of octree representation, more compact, replaces pointers by index manipulation. The nodes are then stored in a hash table, which allows direct access to any node (see Figure 2(bottom)). This representation assigns to each node a *key*, which is used to identify it and to compute its address in the hash table. This key may represent the position of the node in the subdivision hierarchy or the geometry of the associated cube. In efficient schemes, the key cumulates both significations (see Figure 3). This allows at the same 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 [2].

Morton codes. There are several efficient definitions of such keys [19], the most usual being Morton codes [10]. The Morton key k_n of a node n can be generated either recursively from the octree hierarchy or from the geometry of the associated cube.

Following the first approach, the key of the root is 1, and the key of a child of n is the concatenation of k_n with the 3 bits coding the octant of the child (see Figure 3). The depth of n is then $l = \lfloor \frac{1}{3} \log_2 (k_n) \rfloor$ and the key of the parent of n is obtained by removing the 3 last bits of k_n: $k_n \gg 3$.

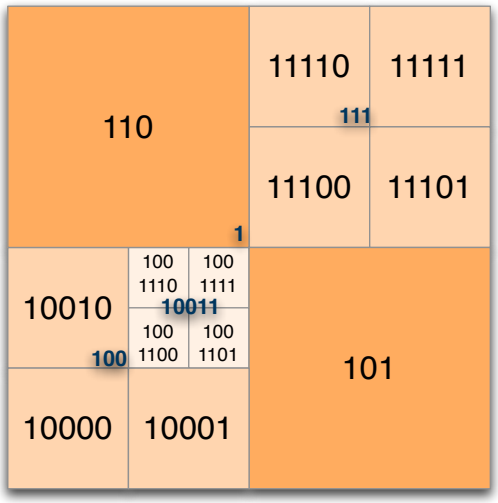

Figure 1: *Quadtree with the Morton keys of each node.*

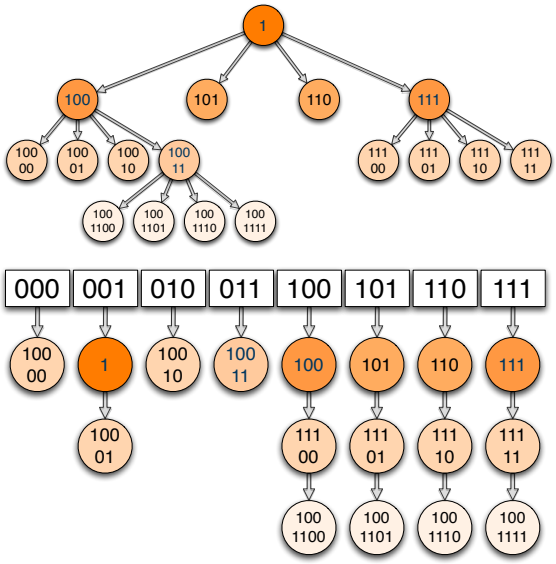

Figure 2: *Two representations for the quadtree of Figure 1, using pointers (top) and hash table (bottom). The hash table uses the three last bits of the key: k & $\bar{0}111$.*

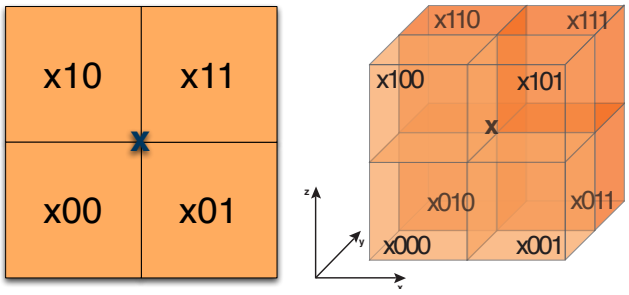

Figure 3: *Suffixes to append to the Morton code x of a node to obtain the codes of its children, in 2d (left) and 3d (right).*

The corresponding work was published in the proceedings of SGP 2010: Computer Graphics Forum, volume 29, number 5, Blackwell 2009.

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 side of the cube associated to n is 2^{-l} and the key k_n is computed by interleaving the bits of x, y and z: if $x = 0.x_1x_2 \dots x_M$, $y = 0.y_1y_2 \dots y_M$ and $z = 0.z_1z_2 \dots z_M$, then $k_n = 1z_1y_1x_1z_2y_2x_2 \dots z_ly_lx_l$ (see Figures 1 and 4). This interleaving can be accelerated using dilated integer operations [14, 18].

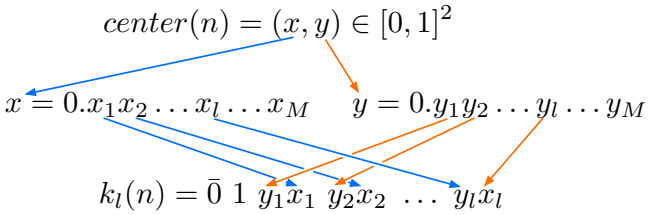

Figure 4: *Bit interleaving for the Morton key of n at depth l.*

Morton hashing. The usual hash function for Morton keys assigns to a node n the b last bits of its key: $k_n \bmod 2^b = k_n \ \& \ \bar{0}11 \dots 1$. This leads to a hash table of size 2^b. Two nodes would collide (i.e. have the same Morton key) if they have the same path from their $\frac{b}{3}$-ancestry. This scheme intends to equalize the hash table (see Figure 2), in the sense that its entries are regularly distributed, especially when the octree is unbalanced. To obtain good hashing performances, b must be big enough to avoid hash collisions, which may require large amount of continuous memory, although the amount of memory actually used is always less than for pointer representation.

3 Octree Duals

The definition of the dual of an octree follows the notion of Poincaré duality for cell complexes [6]. In the 2D case, the dual can be obtained informally by creating dual vertices at the center of each leaf of the quadtree, and drawing a dual edge between vertices of adjacent leaves, i.e. leaves of the quadtree sharing an edge (see Figure 5).

More formally, the cell complex is associated to the last level of the octree, where the 3-dimensional cells are the cubes associated to the leaves of the octree. The cells of dimension 2, 1 and 0 are respectively the faces, edges and vertices of those leaf cubes. The dual is the cell complex whose cells of dimension c are in bijection to cells of dimension $3 - c$ of the octree. In particular, leaf cubes of the octree are identified to vertices of the dual, and vertices of those cubes in the interior of the octree are identified to volumes (3-cells) of the dual.

The adjacency in the dual is defined as follow. If a c-cell e of the octree has (primal) cells v_1 and v_2 as faces, then the dual cells identified with v_1 and v_2 are adjacent in the dual complex, sharing the dual cell identified with e. In particular, the dual vertices identified with adjacent leaves of the octree are linked by a dual edge.

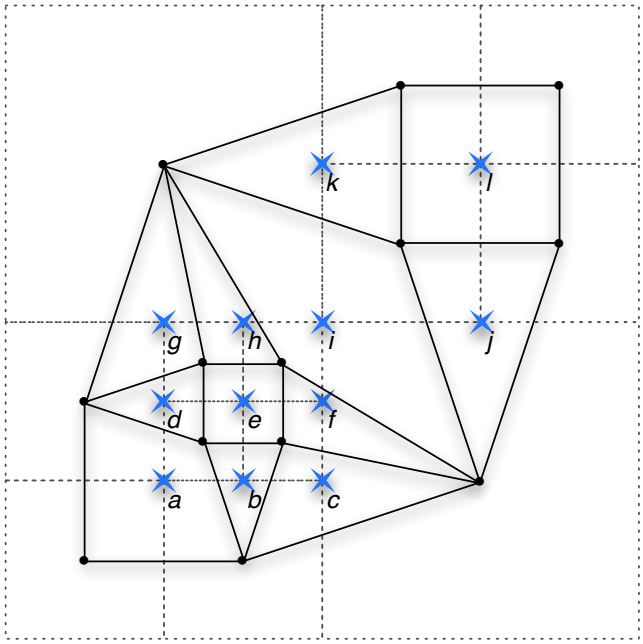

Figure 5: *Dual of the quadtree of Figure 1, with the leaf vertices corresponding to dual volumes marked.*

Frequently, the volumes associated to vertices on the boundary of the octree are discarded [12, 16, 9]. The dual volumes are frequently represented by eight octree leaves (i.e. dual vertices), although some of those leaves may be repeated. This redundancy is useful in applications such as Dual Marching Cubes [15] since it allows handling dual volumes as combinatorial cubes, although their geometry may differ.

Recursive generation. The usual computation of dual octrees requires a recursive implementation [7, 15], returning each dual cell by its dual vertices, or equivalently the associated octree leaves. The recursion starts with the root of the octree, which corresponds to a single dual vertex. A recursive function is implemented for each kind of dual cell, namely dual vertex / primal cube `cubeProc`(n_0), dual edge / primal face `faceProc`(n_0, n_1), dual face / primal edge `edgeProc`(n_0, n_1, n_2, n_3) and dual volume / primal vertex `vertProc`$(n_0, \dots, n_7)$. Those functions stop when all of their arguments are leaf nodes. They recurse when some of its arguments are not leaves, and call the functions corresponding to all the dual cells created by subdividing those nodes. The dual volumes are returned only from `vertProc`. In the 2D case, the creation of the quadtree dual use three functions represented in Figure 7, and their use is illustrated in Figure 6.

Observe that this algorithm calls one of the recursive function once for each cell it traverses, i.e. once for all the cells of all the levels of the octree. Its complexity is therefore linear. Our proposal is also linear, but its complexity is proportional only to the number of vertices of the leaves of the octree, which is around nine times less.

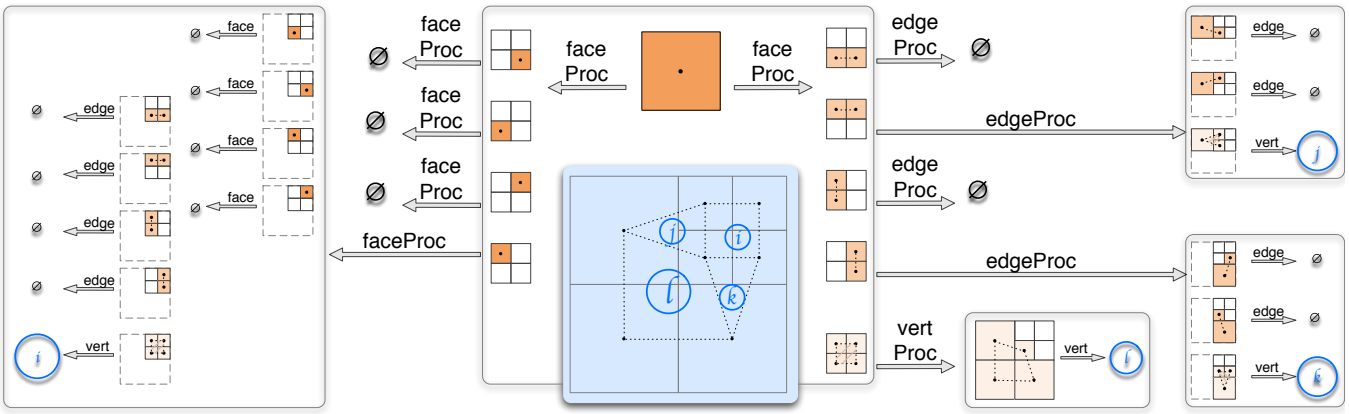

Figure 6: *Illustration of the recursive generation of the dual quadtree (middle bottom), made of four faces (i,j,k,l). The sequence of calls of the recursive procedures of Figure 7 starts from the root node (middle top).*

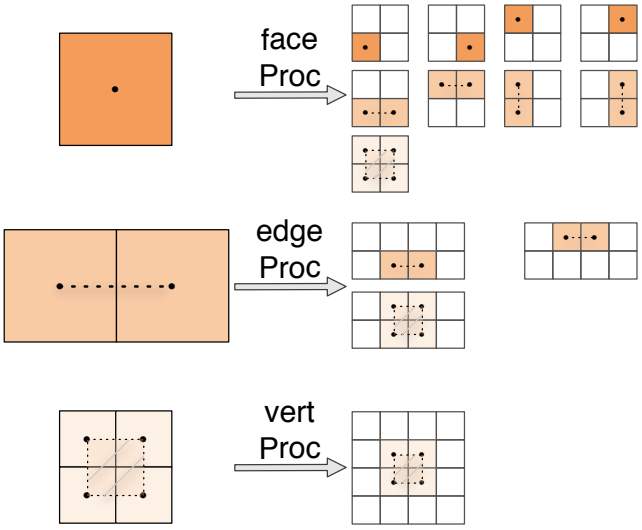

Figure 7: *Recursive procedures to generate the dual of a quadtree (image inspired from [7]).*

Volume octree. Recently, León *et al.* [9] proposed to store extra information within the octree leaves to accelerate the dual generation. More precisely, they assign each dual cell to one of its dual vertex, i.e. a leaf node. Each octree node has eight extra bits to mark if it is responsible for each of the eight possible primal vertices (see Figure 8). This assignment is done in a two-passes traversal of the octree. Then, the dual extraction traverses the octree and for each leaf, processes the leaf vertices assigned to it by searching for the eight possible adjacent leaves. It returns the dual cell as the eight search results (see Figure 9).

Our proposal also builds the dual cells from the octree vertices. However, it optimizes the final search, simplifies the assignment and is able to avoid the preprocessing and the extra memory used for it.

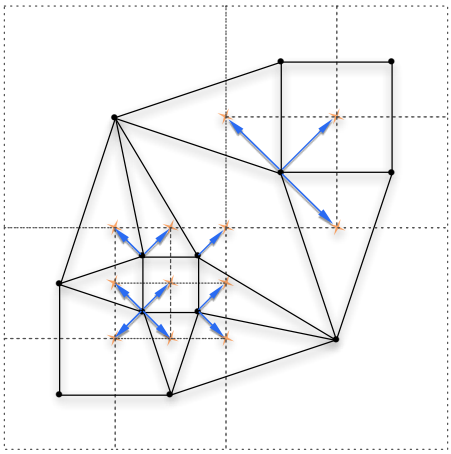

Figure 8: *Illustration of the volume quadtree data structure: each vertex is assigned to an adjacent leaf.*

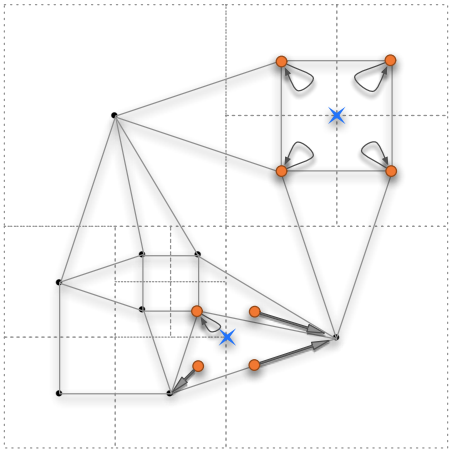

Figure 9: *Computing the dual volume from a vertex by searching for leaves at the four (eight for octree) adjacent positions, marked as an orange dot for two of the vertices. For the upper right vertex, the four searches returns immediately. For the lower left one, one search returns immediately, and the three others must look one level up in the quadtree, two of them eventually leading to the same leaf.*

The corresponding work was published in the proceedings of SGP 2010: Computer Graphics Forum, volume 29, number 5, Blackwell 2009.

Algorithm 1: Preprocessing step: vertex generation

in : The octree
out Auxiliary hash table *aux* with the vertices
:
1 **foreach** *key* k *of the octree's leaves* **do**
  // codes for k*'s vertices, see Algorithm leaf2vert*$_{opt}$
2 $level, v\_codes[8] \leftarrow$ leaf2vert(k) ;
3 **for** $i \in [0 \cdots 7]$ **do**
4 **if** $v\_codes[i] = \emptyset$ **then next** vert i ;
  // get current vertex/level data of aux
5 $v, lv \leftarrow$ aux[$v\_codes[i]$] ;
6 **if** $v = \emptyset$ **then** aux[$v\_codes[i]$] $\leftarrow level$;
7 **else** $lv \leftarrow \min(level, lv)$;

Algorithm 2: Traversal step: dual generation

in : The octree and the auxiliary hash table
out The dual volumes
:
1 **foreach** *vertex code / level* v, lv *in aux* **do**
  // node keys at level lv, see Algorithm vert2leaf$_{opt}$
2 $keys[8] \leftarrow$ vert2leaf(v,lv) ;
3 **for** $j \in [0 \cdots 7]$ **do**
  // optimized search for leaf: up from level lv
4 **while** $keys[j] \neq \emptyset \ \& \ \neg node\_exists(keys[j])$ **do**
5 $keys[j] \gg= 3$;
6 **output** $keys$;

4 Optimized Dual Generation

The use of keys and hash table to represent octrees allows bypassing the hierarchic traversal. We propose here a scheme to enjoy this aspect in the dual generation with two different strategies. The first one uses Morton-like codes to represent the vertices of the octree, permitting to store all the octree vertices at preprocessing and efficiently search for the dual vertices from the Morton-like codes. It suits for applications where the octree is static, factoring on the preprocessing time. The second strategy avoids the extra storage of the octree vertices, but with a small execution overhead. It suits for dynamic octrees. Both strategies support parallel implementations.

(a) Static strategy

This strategy consists in a preprocessing step, required only when the octree structure is modified, which generates a code for each leaf's vertex; and a dual traversal step which generates the dual volumes by fast local searches (see Algorithms 1 and 2).

The preprocessing step traverses all the octree leaves stored in the hash table, and generates Morton-like codes for all the vertices of those leaves that are in the interior of the octree. Those codes are stored in an auxiliary hash table aux, together with the depth of the leaf. When two leaves share a vertex, the deepest depth is retained. The structure of those Morton-like codes is described in the next paragraph. Those codes can be computed efficiently from the Morton key of the leaf, as described in the subsequent paragraph. The dual traversal step then reads the auxiliary hash table and, for each dual vertex / leaf depth pair, searches for the eight octree leaves associated to that vertex. Since the depth of the deepest leaf is known, the search can be optimized further than generic optimized searches [2].

Table 1: *Morton codes for the vertices of Figure 5.*

a: $10011100\bar{0}$	e: $10011110\bar{0}$	i: $11100000\bar{0}$
b: $10011010\bar{0}$	f: $10110100\bar{0}$	j: $11101000\bar{0}$
c: $10110000\bar{0}$	g: $11001000\bar{0}$	k: $11110000\bar{0}$
d: $10011100\bar{0}$	h: $11001010\bar{0}$	l: $11111000\bar{0}$

Morton-like codes for vertices. The Morton key of an octree node corresponds to the geometric position of its center. The centers of all the possible nodes of depth l form a regular grid of 2^l units per side. The interior vertices of those nodes actually form a similar grid, obtained from the previous one by a translation of vector $(2^{-l-1}, 2^{-l-1}, 2^{-l-1})$ and removing the vertices on the boundary. Since those are binary positions, they can be represented directly by Morton codes, using the geometric key generation (see Algorithm leaf2vert). On the contrary to octree nodes, which have a depth limiting the key size, the leaves' vertices have no specific depth associated to them. Therefore, their codes must be generated at the maximal depth instead of $\lfloor \frac{1}{3} \log_2 (k_n) \rfloor$. This requires adjusting the hashing of the auxiliary hash table, as detailed at the end of this subsection. The codes for the vertices of Figure 5 are given in Table 1.

Algorithm leaf2vert: Morton codes for vertices (slow)

in : The Morton key k of the leaf
out The eight codes for its vertices
:
1 $c \leftarrow key2cube(k)$; *// see Figure 4*
2 **for** $i \in [0 \cdots 7]$ **do**
  // get the vertex coordinates at maximal level
3 $vert \leftarrow cube(c.[xyz] \pm c.side, MAX\_LEVEL)$;
  // overflow test
4 **if** $vert.[xyz] \notin [0,1]^3$ **then output** $\emptyset$;
5 **else output** $cube2key(vert)$;

Fast code translations. A direct implementation of the previous key generation requires translating Morton codes to coordinates, check the validity of those integers and translate back (see Algorithm leaf2vert). However, one advantage of Morton codes is that they can be efficiently manipulated using dilated integers [14]. We therefore adapt usual dilated integer addition and propose overflow test for converting between Morton-like codes of vertices and Morton keys of adjacent leaves avoiding coordinate representations in Algorithms leaf2vert$_{opt}$ and vert2leaf$_{opt}$. No overflow test is required for Algorithm vert2leaf$_{opt}$ since all the input keys come from Algorithm leaf2vert$_{opt}$.

Algorithm leaf2vert$_{opt}$: Morton codes for vertices

in : The Morton key k of the leaf
out The eight codes for its vertices
:
1 $dil_x \leftarrow \overline{001}001$; $dil_y \leftarrow \overline{010}010$; $dil_z \leftarrow \overline{100}100$;
2 $lv \leftarrow key2level(k)$; $lv_k \leftarrow 1 \ll 3 \cdot lv$;
3 **for** $i \in [0 \cdots 7]$ **do**
  // dilated integer addition $k + i$
4 $vk \leftarrow \{ [(k \mid \neg dil_x) + (i \,\&\, dil_x)] \,\&\, dil_x \} \mid$
  $\{ [(k \mid \neg dil_y) + (i \,\&\, dil_y)] \,\&\, dil_y \} \mid$
  $\{ [(k \mid \neg dil_z) + (i \,\&\, dil_z)] \,\&\, dil_z \}$;
  // overflow test (repeat **or** *for* $[xyz]$*)*
5 **if** $(vk \geq (lv_k \ll 1))$ **or** $\neg((vk - lv_k) \,\&\, dil_{[xyz]})$
  then output $\emptyset$;
6 **else output** vk ;
return lv ;

Algorithm vert2leaf$_{opt}$: Leaves' keys from vertex code

in : The Morton-like code c of a vertex and its level lv
out The eight codes of the adjacent leaves
:
1 $dil_x \leftarrow \overline{001}001$; $dil_y \leftarrow \overline{010}010$; $dil_z \leftarrow \overline{100}100$;
// removes trailing 0's
2 $dc \leftarrow c \gg 3 \cdot (MAX\_LEVEL - lv)$;
3 **for** $i \in [0 \cdots 7]$ **do**
  // dilated integer substraction $dc - i$
4 **output** $\{ [(dc \,\&\, dil_x) - (i \,\&\, dil_x)] \,\&\, dil_x \} \mid$
  $\{ [(dc \,\&\, dil_y) - (i \,\&\, dil_y)] \,\&\, dil_y \} \mid$
  $\{ [(dc \,\&\, dil_z) - (i \,\&\, dil_z)] \,\&\, dil_z \}$;

Optimized search for dual volume. The auxiliary hash table stores a code of a vertex v together with the depth l of the deepest adjacent leaf. Using Algorithm vert2leaf$_{opt}$, the Morton keys k_i of the eight adjacent nodes of depth l are computed. The dual vertices (i.e. octree leaves) of the dual volume associated to v is then retrieved performing a search in the hash octree with those codes (see Figure 9). Observe that dual vertices may be repeated, which is the desired representation of dual volumes as combinatorial cubes.

The search in hash octrees from a Morton key k_i looks for leaves at, below and above the depth of k_i until the hash table lookup returns a leaf. However, since we know the depth l of the deepest adjacent leaf, we do not need to search deeper than l (see Algorithm 2, lines 4,5).

Moreover, we guarantee that at least one of the hash table lookup search will return a leaf at the first try, and 2.37 searches at least return immediatly (8 for the central vertex of the father of a leaf, at least 4 for its 6 faces centers, at least 2 for its 12 edges centers and at least one for its 8 vertices). The other searches have an (improbable) worst-case complexity of l, but constant in practice (see Section 5). This leads to a total complexity of the dual generation of less than 200 bit operations per leaf and one auxiliary hash table access for the preprocessing, and less than 100 bit operations *per dual volume* plus the search accesses for the traversal.

Combinatorial cube retrieval. From the outputs of Algorithm 2, the combinatorial cube representing the dual volume has vertices $keys[0] \dots keys[7]$, where the indexes written in binary are the unit cubes' coordinates. The dual edges are then $\big[keys[a]\ keys[b]\big]$ when a and b differ from exactly one bit. Finally, the dual faces are $\big[keys[a]\ keys[b]\ keys[c]\ keys[d]\big]$ where a, b, c, d have only one bit in common. The degenerated edges and faces can be removed by testing if all their vertices have the same key.

Hash function for the leaf vertices. The auxiliary hash table access may thus be crucial for good performance of the preprocessing. As we mentioned earlier, the least significant bits of Morton keys for octree nodes are good hash functions. However, since the vertex codes must always be generated at the maximal depth, most of the vertex codes end with a sequence of 0. Using the least significant bits for hashing would induce a huge collision in the auxiliary hash table. Therefore, we use here bits starting from the median depth of the octree. This maintains the spirit of Morton hashing by using the least significant bits, but avoid incorporating the final sequence of 0 for at least half of the leaf vertices. Since we know the octree statistics before the preprocessing step, this is easily implemented.

(b) Dynamic strategy

The above strategy processes each leaf vertex exactly once, but this requires storing all the vertices, leading to more memory operations. This extra preprocessing cost is amortized if building several times the dual without modifying the octree. However, some applications such as view-dependent isosurface generation constantly adapt the octree before the dual generation. Moreover, the extra memory cost of the auxiliary hash table may be prohibitive for very large data.

We propose here a dynamic dual generation that avoids the vertex generation as preprocessing (see Algorithm 3 and Figure 10). Since an interior vertex is always shared by several leaves, the main difficulty is to guarantee that each vertex is processed only once. We thus define a one-to-one mapping from the vertices to the leaves. This would correspond

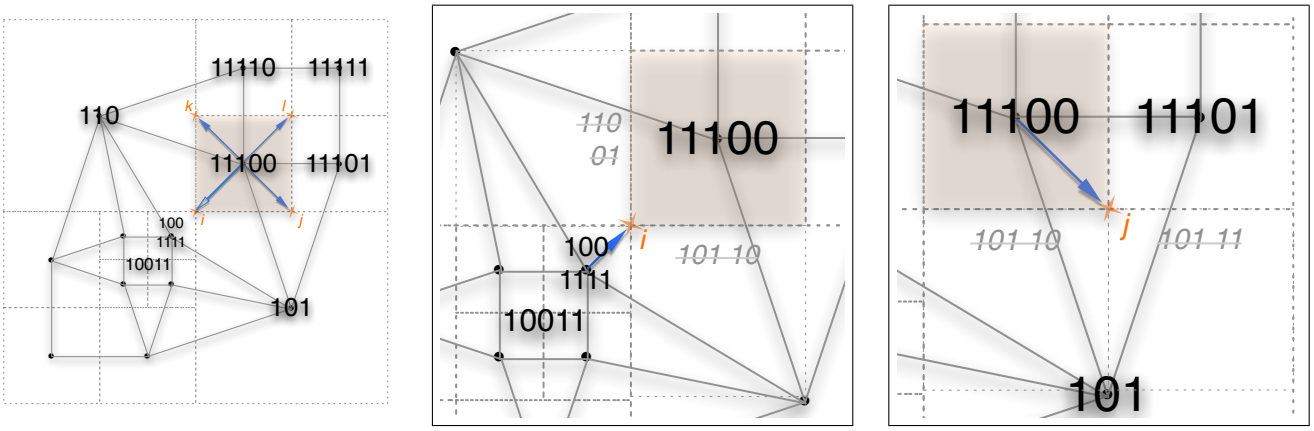

Figure 10: *Illustration of the dynamic strategy, processing leaf $k_n = 11100$ of the quadtree of Figure 1, with the notation of Algorithm 3. (left) The codes of the vertices of k_n are computed: $lv = 2, v\_codes = \{i, j, k, l\}$ (see Table 1). (center) Processing vertex i: the Morton keys of the nodes of level $lv = 2$ adjacent to vertex i are computed: $keys = \{11100 = k_n, 11001, 10110, 10011\}$. The first key 11100 is k_n itself, and is thus skipped (line 6). The second key 11001 is not a leaf (test of line 8), so vertex i will not be processed from leaf k_n: indeed it has been processed from leaf 1001111. (right) Processing vertex j: the Morton keys of the nodes of level $lv = 2$ adjacent to vertex j are computed: $keys = \{11101, 11100 = k_n, 10111, 10110\}$. The first key has the same level as k_n, leading to a tie with k_n. Since k_n appears before in the order around j (test of line 9), vertex j will be processed from k_n. The second key is skipped since it is k_n itself (line 6). The third and last keys do not correspond to existing nodes (they do not appear in the hashtable of Figure 2), so the algorithm skips those keys (test of line 7). The two other vertices are similar to j.*

to the assignment defined in the volume octree structure of León *et al.* [9], with the difference that it is defined here systematically and online, allowing for a one-pass dual generation without preprocessing.

The mapping associates a vertex to the deepest adjacent leaf. More precisely, for each leaf k, the eight codes v_i of its vertices are generated. For each vertex, the eight keys of the adjacent nodes n_j, i.e. neighbors of k in the direction of v_i, are computed. If k is deeper than all the adjacent nodes n_j, then it is associated to v_i (lines 7,8 of Algorithm 3). In case of tie, i.e. if k and n_j have same depth, we choose the first one in Morton order around the vertex. We can observe from Algorithms leaf2vert$_{opt}$ and vert2leaf$_{opt}$ that k is always generated as the i-th node adjacent to v_i, since the first algorithm adds i while the seconds subtracts i. This observation leads to a simple test to avoid checking k against itself (line 6 of Algorithm 3), and to resolve ties (line 9).

The resulting association is the one actually illustrated in Figure 8, although the volume octree structure assignment may be different. Observe that this strategy does not use any extra memory, and performs only a few more memory accesses per leaf than the static approach (tests of lines 7 and 8 of Algorithm 3 can be done with the same access).

Algorithm 3: Dynamic dual generation

in : The octree
out The dual volumes
:

1 **foreach** *key k of a leaf of the octree* **do**
  // get the vertex codes of the leaf
2 $lv, v\_codes[8] \leftarrow$ leaf2vert(k) ;
3 **for** $i \in [0 \cdots 7]$ **do**
4 **if** $v\_codes[i] = \emptyset$ **then next** vert i;
  // get the nodes of level lv adjacent to vertex $v\_codes[i]$, i.e. a neighbor node of leaf k
5 $keys[8] \leftarrow$ vert2leaf(v_codes[i],lv) ;
6 **for** $j \in [0 \cdots 7] \setminus \{i\}$ **do**
  // leaf k is deeper than neighbor$keys[j]$: OK, check next neighbor key
7 **if** $\neg node\_exists(keys[j])$ **then next** key j;
  // neighbor is deeper than leaf: skip vertex since it will be processed by that neighbor
8 **if** $\neg is\_leaf(keys[j])$ **then next** vertex i;
  // neighbor has same level: tie, it will process the vertex if $j < i$
9 **if** $j < i$ **then next** vertex i;
  // the vertex is processed as in Algorithm 2
10 **for** $j \in [0 \cdots 7]$ **do**
  // optimized search for leaf: up from level lv
11 **while** $keys[j] \neq 0$ & $\neg node\_exists(keys[j])$ **do** $keys[j] \gg= 3$;
12 **output** $keys$;

Table 2: *Execution time and memory consumption for the dual generation of randomly generated octrees, as average on 200 runs, including the preprocessing of the static strategy. The gain for each strategy is given for comparable memory consumption.*

number of nodes (millions)		0.03	0.07	0.12	0.19	0.65	1.01	3.19	13.04	24.81
number of vertices (millions)		0.05	0.14	0.22	0.33	0.22	1.87	5.69	2.23	3.18
octree maximal level M		8	10	8	8	12	10	10	12	12
subdivision probability p		30%	30%	40%	45%	30%	40%	45%	40%	45%
bits for node hashing		21	21	21	21	21	24	24	24	24
median bits used for vertex hashing		40	34	40	40	34	34	34	34	34
time (ms)	recursive with pointer	82	223	349	528	2 033	2 990	9 151	38 449	70 528
	recursive pointerless	35	97	151	227	901	1 278	3 978	17 675	34 093
	static	33	73	101	148	166	893	2 737	2 357	3 878
	dynamic	32	79	117	176	150	1 012	3 091	1 975	3 341
	pointer / static	*2.5x*	*3.1x*	*3.4x*	*3.6x*	*12.2x*	*3.3x*	*3.3x*	*16.3x*	*18.2x*
	hash / dynamic	*1.1x*	*1.2x*	*1.3x*	*1.3x*	*6.0x*	*1.3x*	*1.3x*	*8.9x*	*10.2x*
memory (MB)	recursive with pointer	0.64	1.67	2.79	4.31	14.93	23.07	73.09	298.47	567.81
	recursive pointerless	0.21	0.56	0.93	1.44	4.98	7.69	24.36	99.49	189.27
	static	0.62	1.65	2.61	3.96	6.63	21.95	67.80	116.50	213.50
	dynamic	0.21	0.55	0.93	1.43	4.97	7.68	24.36	99.48	189.26

5 Experiments

We experimented on random octrees and octrees adapted to isosurfaces, on a $3GHz$ MacPro with $18GB$ of RAM.

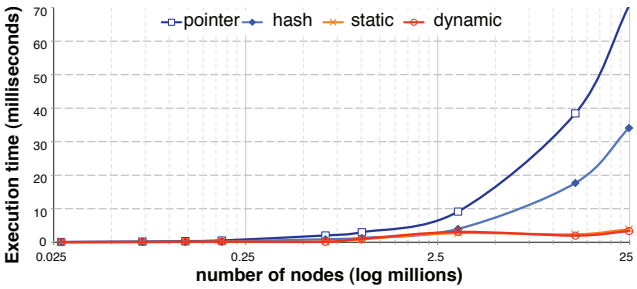

Figure 11: *Execution time, in milliseconds, versus the octree size, in millions of nodes / logarithmic scale (see Table 2).*

Random octrees. We first tested on random octrees, with different maximal levels M and Bernouilli probabilities p for a node to be subdivided. We compared the execution time and memory consumption of our static and dynamic strategies with the usual recursive implementation on pointer and pointerless octrees (see Table 2 and Figure 11). The average gain in memory consumption of the pointerless representations is a factor $3x$, which is preserved in the dynamic strategy. For the static strategy, the extra memory of the auxiliary hashtable reduces this average memory gain to a factor $1.5x$. Both the static and dynamic strategies speeds up the execution by an average factor above $3.3x$ on the recursive implementation with hashtable, and above $7.3x$ over the recursive algorithm with the octree representation with eight pointers per node. Note that this includes the preprocessing time for the static strategy, which represents 53% of the total execution time. This means that, for the second and further runs on the same octree, the gain of the static strategy is doubled.

Octrees adapted to isosurface. We compared the gain of our dynamic dual generation over the recursive generation on the total time of an isosurface extraction application. We experimented on Dual Marching Cubes [15] using robust adaptation [12]. We generated results from 24 different implicit functions in the unit cube, refined to maximal depth 9 and with curvature threshold 0.6 (see Table 3 and Figure 12). Since the timings include the octree adaptation and Marching Cubes calls on the dual volumes, the total gain is in average 30%, and 64% if we weight by the number of nodes. The two methods compared use hashtables with the same parameters, leading to the same memory consumption.

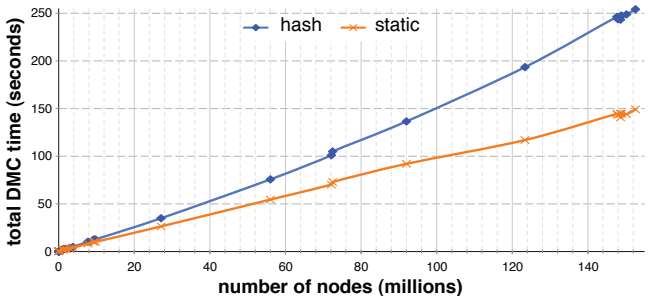

Figure 12: *Total execution time of robust DMC, in seconds, versus the number of octree nodes, in millions (see Table 3).*

Limitation. We can observe on Table 2 that the gain obtained by the proposed algorithms varies brutally when the number of bits b used for the hashing function is changed to

Table 3: *Total execution time for the robust generation of implicit surfaces using a robust Dual Marching Cubes.*

Implicit function	nodes $x10^6$	verts $x10^6$	hash *sec*	dyn *sec*	gain %
Torus	0.00	1	0.1	0.3	-67
Blob	0.03	4	0.1	0.4	-65
Cross cap	0.05	10	0.2	0.4	-53
Spheres in	1.0	85	1.8	1.8	2
Cylinders	1.3	159	2.5	2.3	6
2 Spheres	2.0	105	2.9	2.6	11
Glob tear	2.9	24	3.8	3.2	17
Weird cube	3.8	3	4.8	4.0	21
Lemniscate	7.8	140	10.4	8.3	25
Clebsch cubic	9.2	225	12.6	10.0	25
Cayley cubic	9.7	119	12.8	10.2	26
Steiner relative	27.1	39	34.9	26.4	33
Mitre	56.0	158	75.7	54.5	39
Bifolia	72.1	310	101.0	70.2	44
Chair	72.5	966	105.0	72.9	44
Gumdrop torus	92.0	1159	136.5	92.0	48
Bretzel 123.4	15	193.5	117.0	65	
Klein bottle	147.7	1195	246.3	144.6	70
Smile 147.9	727	243.8	143.3	70	
Heart 148.6	998	246.7	144.7	71	
2 Torii	148.7	67	243.3	140.7	73
Hunt's surface	148.8	1128	247.6	144.8	71
Barth sextic	150.2	561	248.8	144.1	73
Spheres dif	152.6	1165	254.2	149.2	70

cope with the size of the data. Actually, the speed of hashtable manipulation is a crucial ingredient in pointerless representations. In particular, increasing b size may be delicate in the static strategy, since it would require two large blocks (of size 2^b) of data for the hashtable. A solution to optimize the hashing is to use perfect hashing techniques, which are already used for pointerless octrees [8, 1, 3].

6 Conclusions

In this work, we introduced efficient algorithms for dual generation of pointerless octrees. We proposed two strategies, one using a preprocessing, which requires an extra hashtable, doubling the memory, but achieving, after preprocessing, and average speedup of factor $7x$ compared to pointerless representation and $15x$ compared to the usual pointer representation. The second strategy does not require preprocessing nor extra memory, and achieves an average speedup of a factor above $3x$ compared to pointerless representation, and almost $8x$ compared to pointer octrees.

Acknowledegments

The authors would like to thank FAPERJ, FAPEAL, PUC-Rio and CNPq for their help in financing this research.

References

[1] T. Bastos and W. Celes. GPU-accelerated adaptively sampled distance fields. In *Shape Modeling and Applications*, pages 171 –178, 2008.

[2] R. Castro, T. Lewiner, H. Lopes, G. Tavares and A. Bordignon. Statistical optimization of octree searches. *Computer Graphics Forum*, 27(6):1557–1566, 2008.

[3] M. G. Choi, E. Ju, J.-W. Chang, J. Lee and Y. J. Kim. Linkless octree using multi-level perfect hashing. *Computer Graphics Forum*, 28(7):1773–1780, 2009.

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

[5] 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.

[6] A. Hatcher. *Algebraic topology*. Cambridge University Press, 2002.

[7] T. Ju, F. Losasso, S. Schaefer and J. Warren. Dual contouring of Hermite data. In *Siggraph*, pages 339–346. ACM, 2002.

[8] S. Lefebvre and H. Hoppe. Perfect spatial hashing. In *Siggraph*, pages 579–588. ACM, 2006.

[9] A. León, J. C. Torres and F. Velasco. Volume octree with an implicitly defined dual grid. *Computers & Graphics*, 32(4):393–401, 2008.

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

[11] T. S. Newman and H. Yi. A survey of the Marching Cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006.

[12] A. Paiva, H. Lopes, T. Lewiner and L. H. de Figueiredo. Robust adaptive meshes for implicit surfaces. In *Sibgrapi*, pages 205–212. IEEE, 2006.

[13] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.

[14] 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.

[15] S. Schaefer and J. Warren. Dual Marching Cubes: primal contouring of dual grids. In *Pacific Graphics*, pages 70–76, Washington, 2004. IEEE.

[16] A. Sharf, T. Lewiner, G. Shklarski, S. Toledo and D. Cohen-Or. Interactive topology-aware surface reconstruction. In *Siggraph*, pages 43.1–43.9. ACM, 2007.

[17] R. Sivan and H. Samet. Algorithms for constructing quadtree surface maps. In *Symposium on Spatial Data Handling*, pages 361–370, 1992.

[18] L. J. Stocco and G. Schrack. Integer dilation and contraction for quadtrees and octrees. In *Communications, Computers and Signal Processing*, pages 426–428. IEEE, 1995.

[19] L. J. Stocco and G. Schrack. On spatial orders and location codes. *Transactions on Computers*, 58(3):424–432, 2009.

[20] B. Von Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *Siggraph*, pages 103–110. ACM, 1987.

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