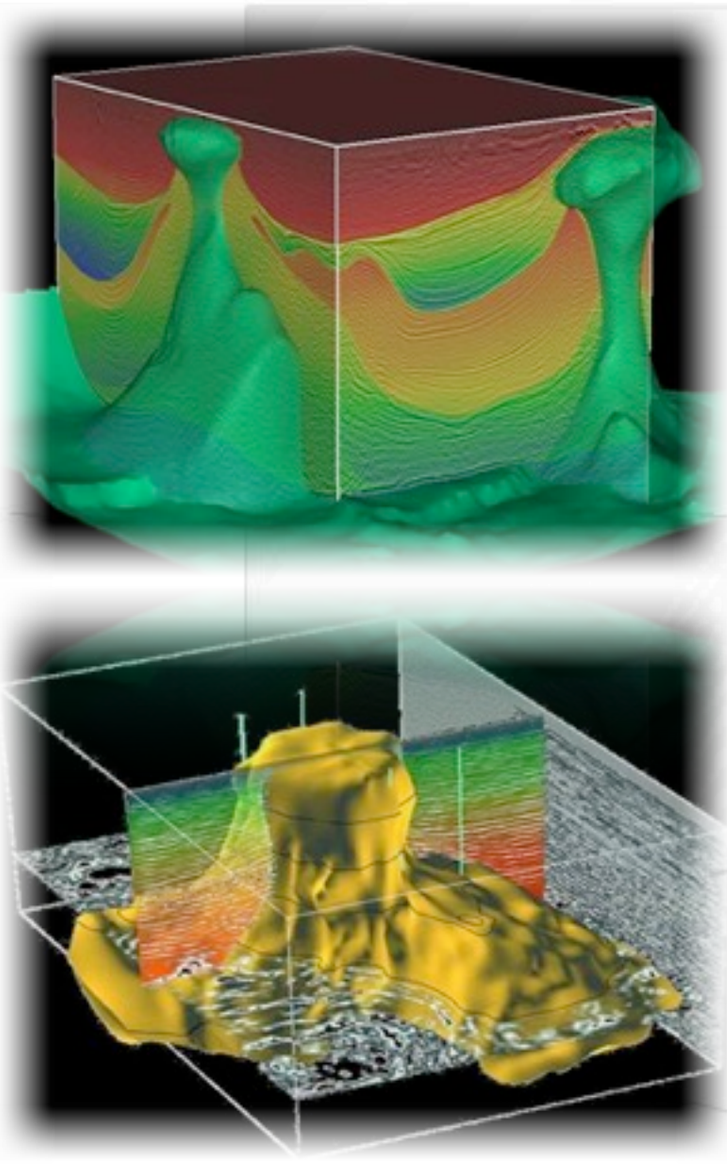
Fast Generation of Pointerless Octree Duals

Thomas Lewiner, Vinícius Mello,
Adelailson Peixoto, Sinésio Pesco, Hélio Lopes

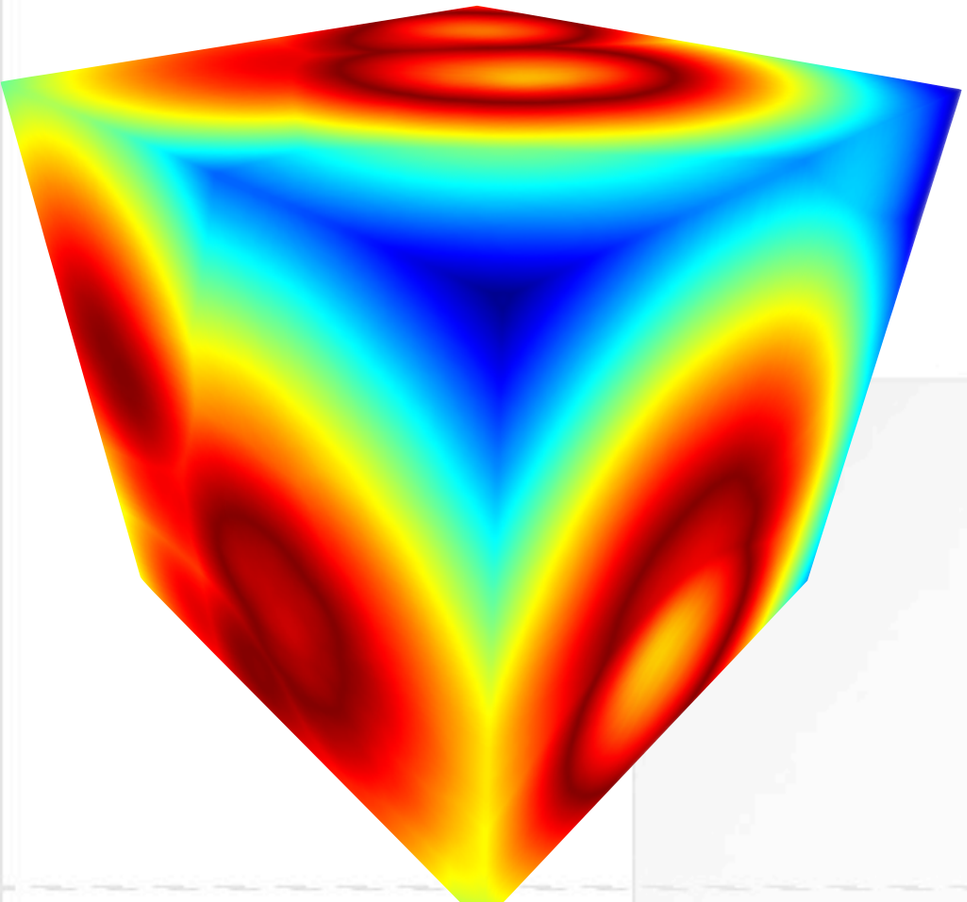
PUC-Rio de Janeiro, UFBA & UFAL - Brazil

Where we started

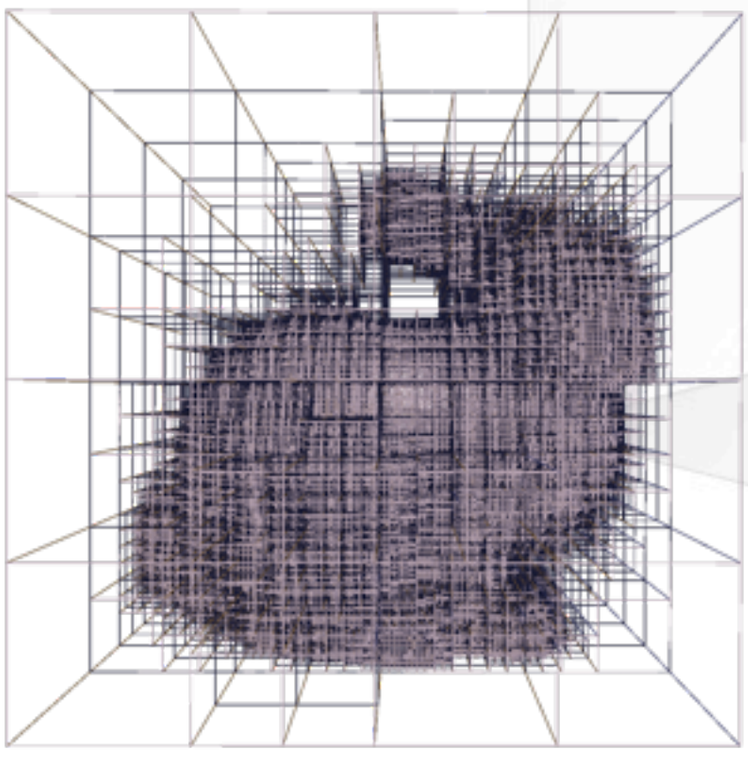


huge data
ex: seismic measures

limited resources
ex: our lab's computers

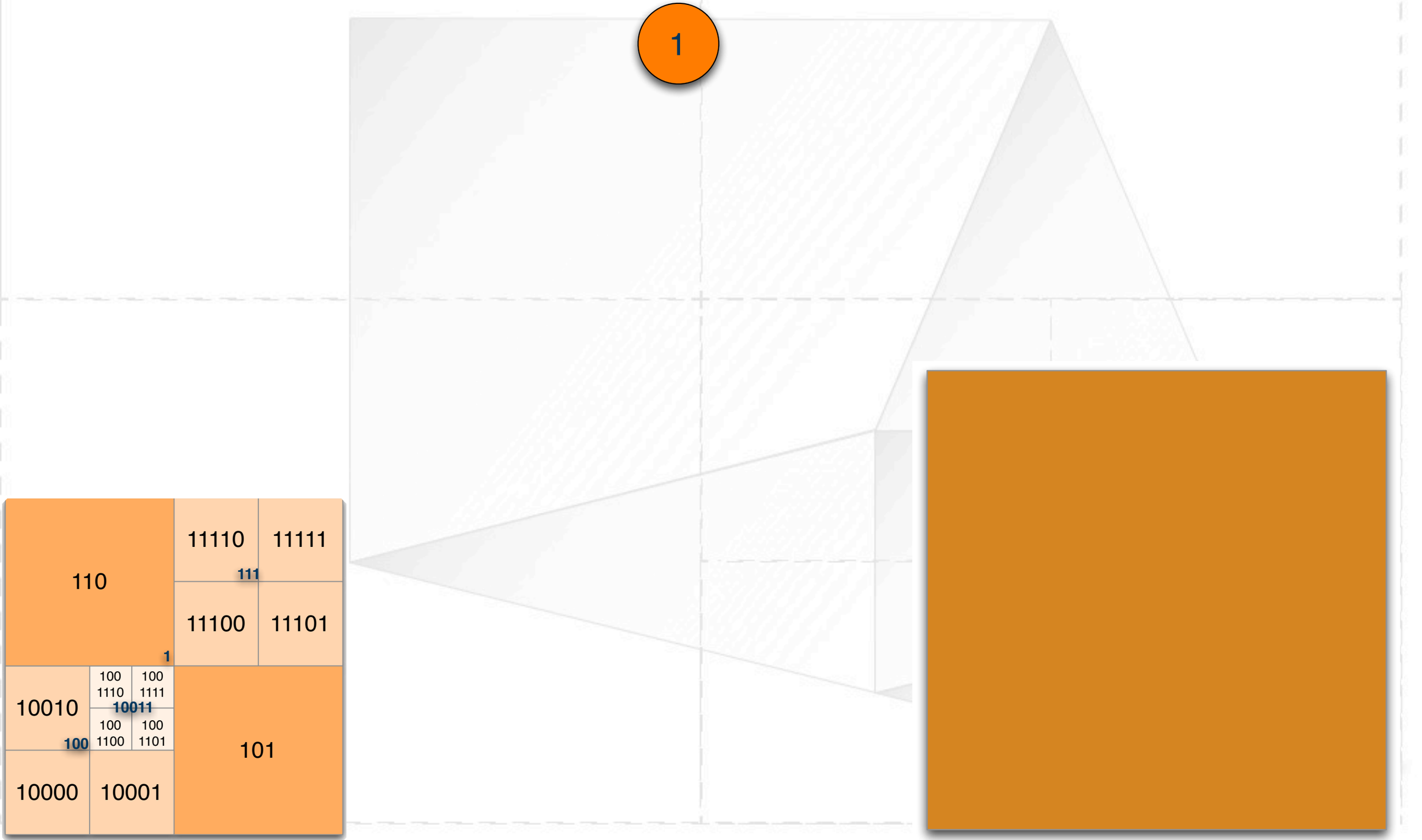


2^n-trees

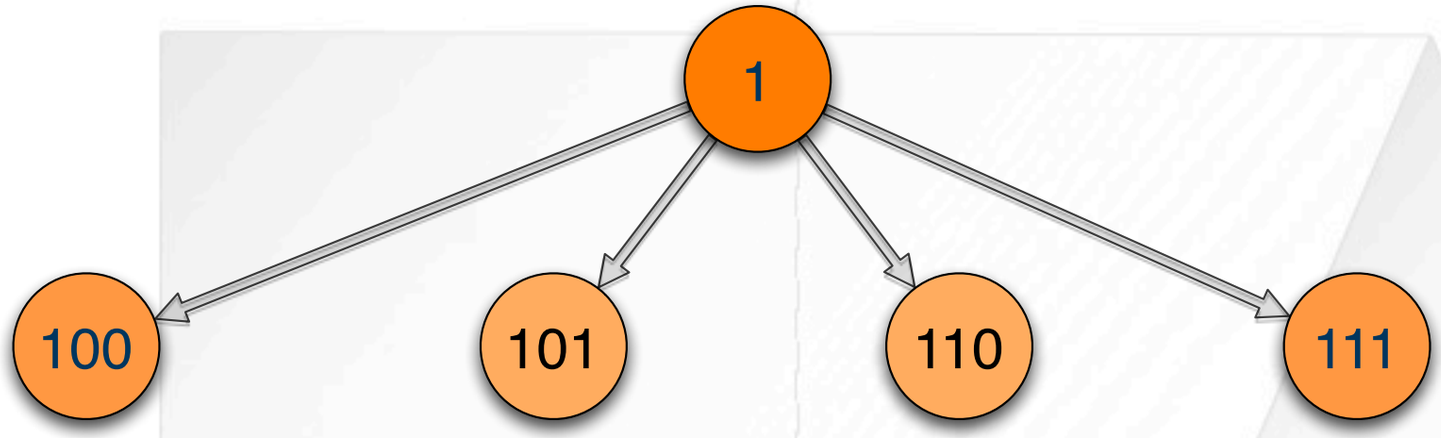


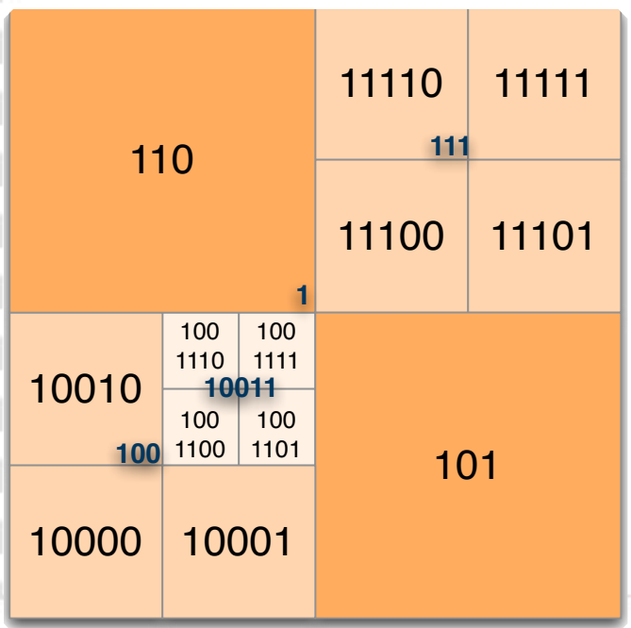
precision only near
the region of interest

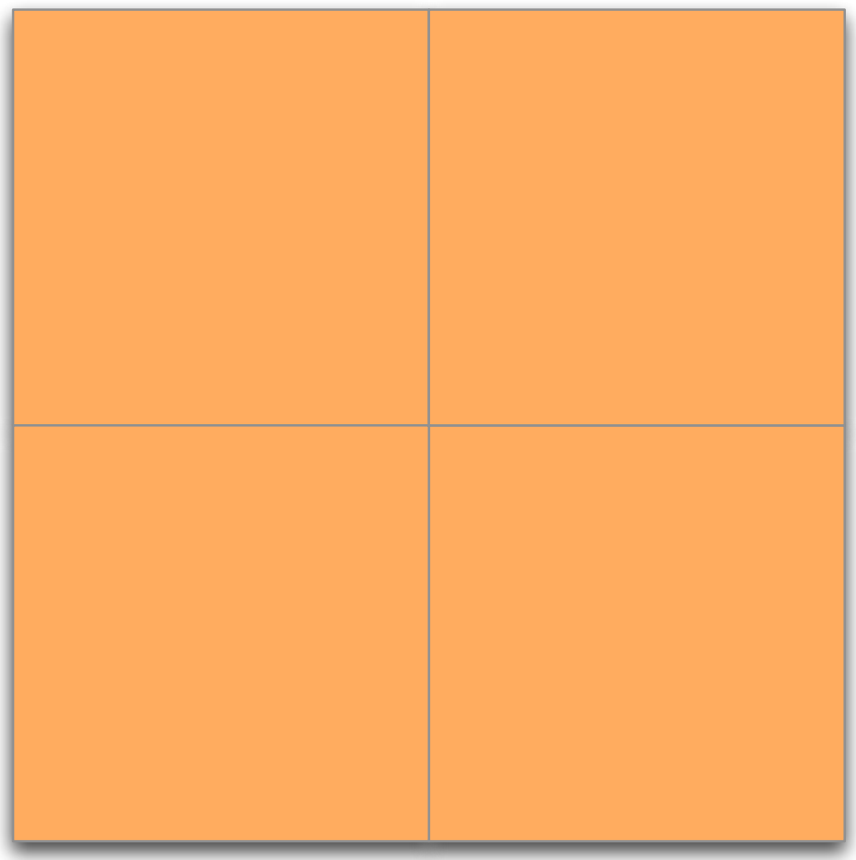
Hierarchic representation



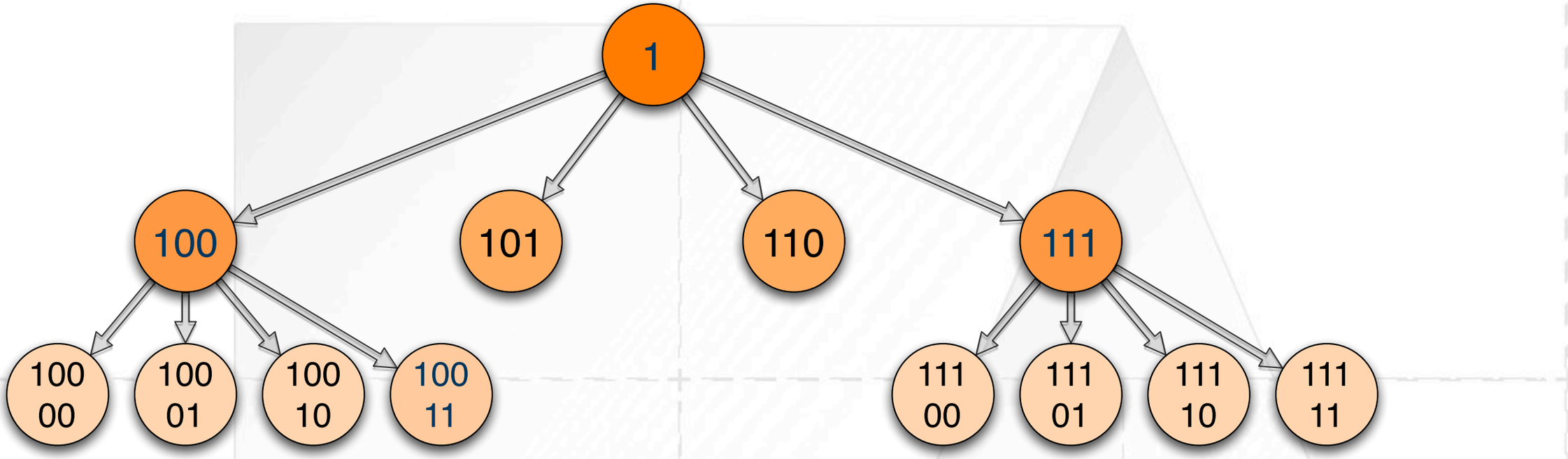
Hierarchic representation

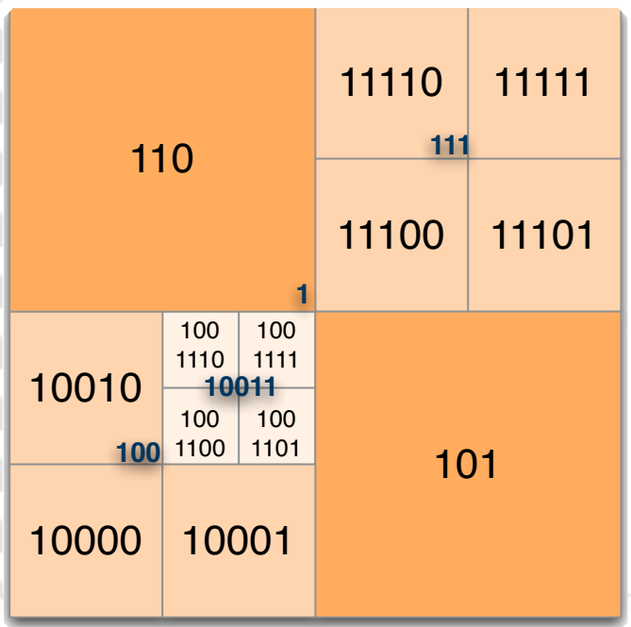


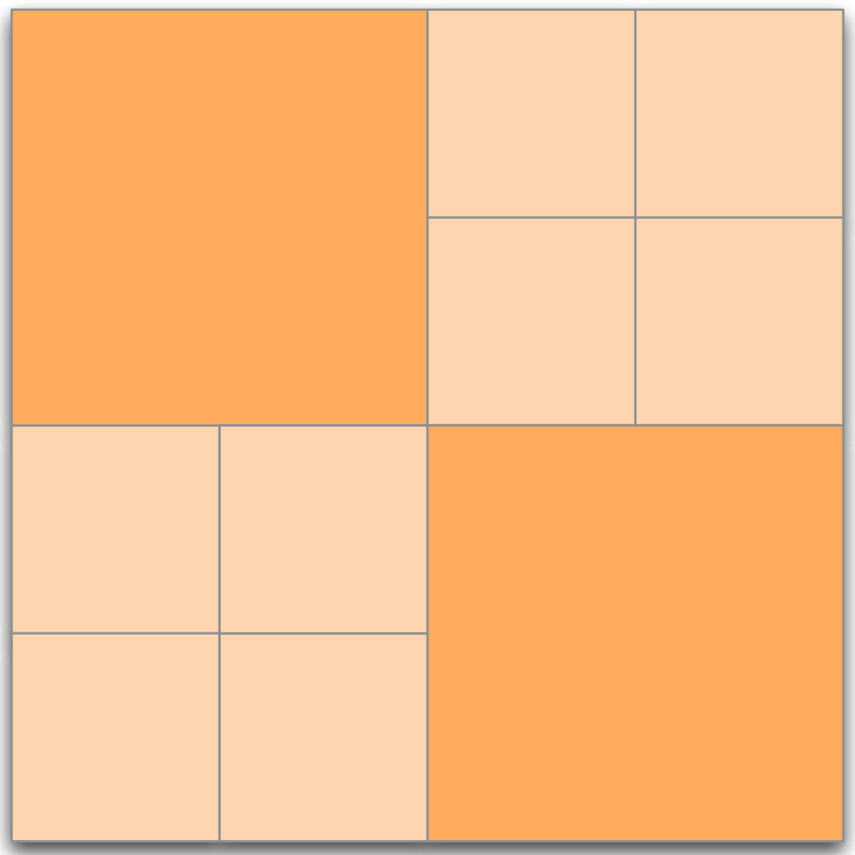




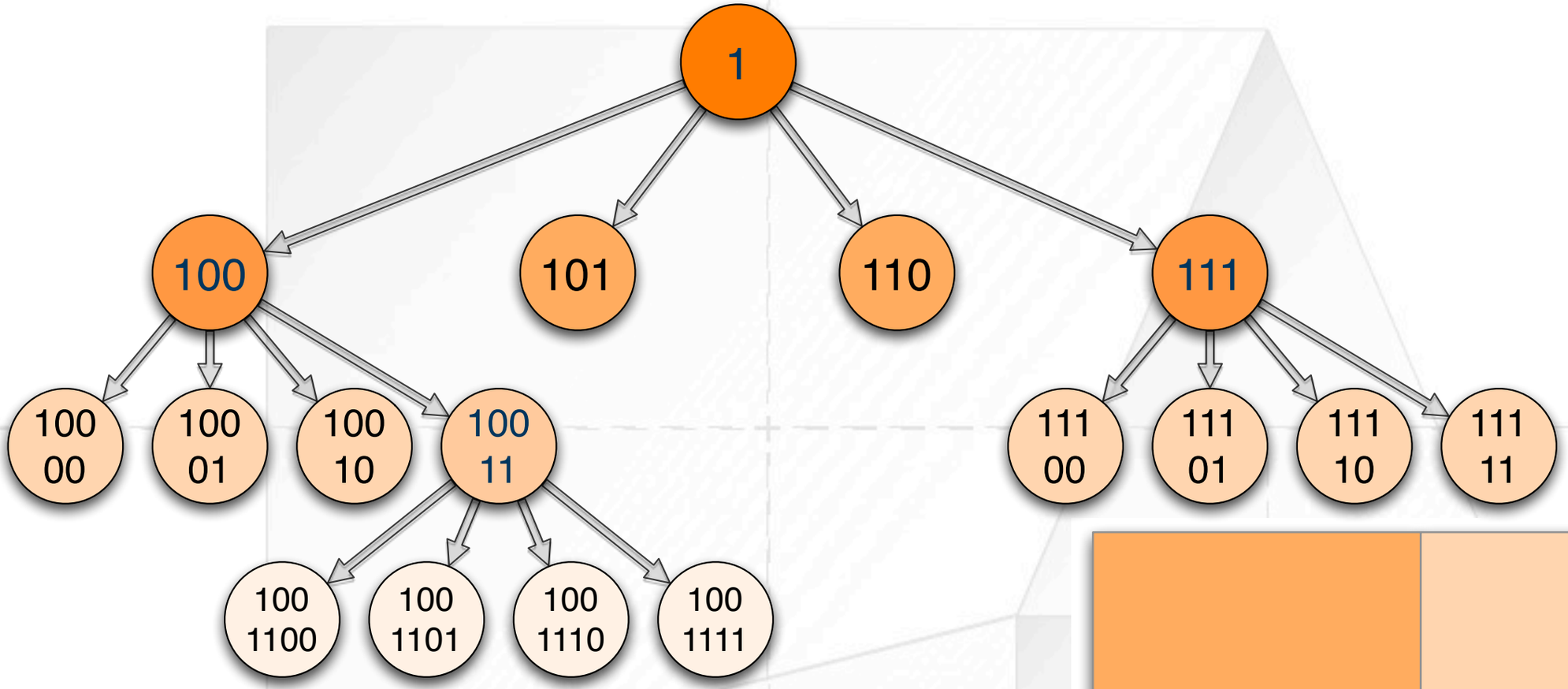
Hierarchic representation

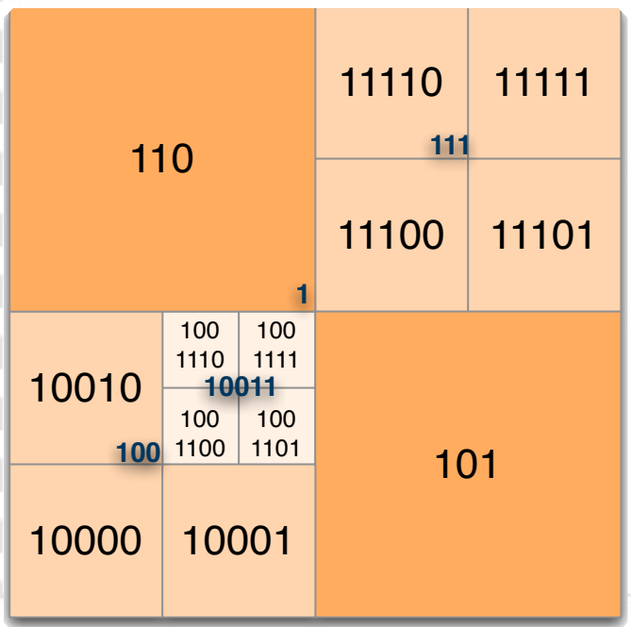


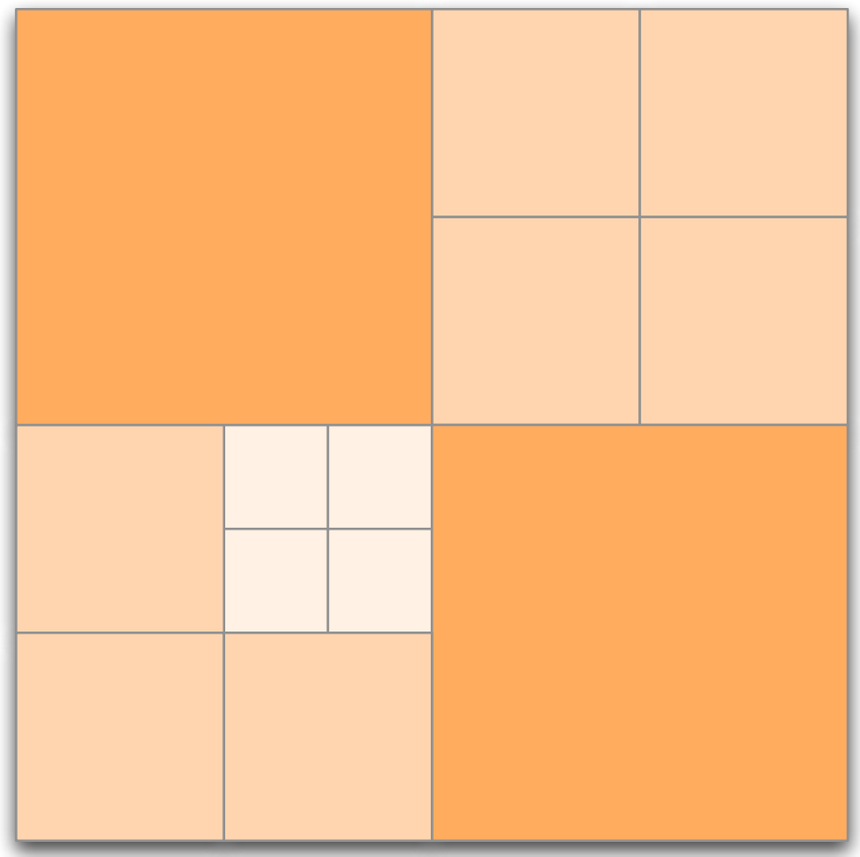




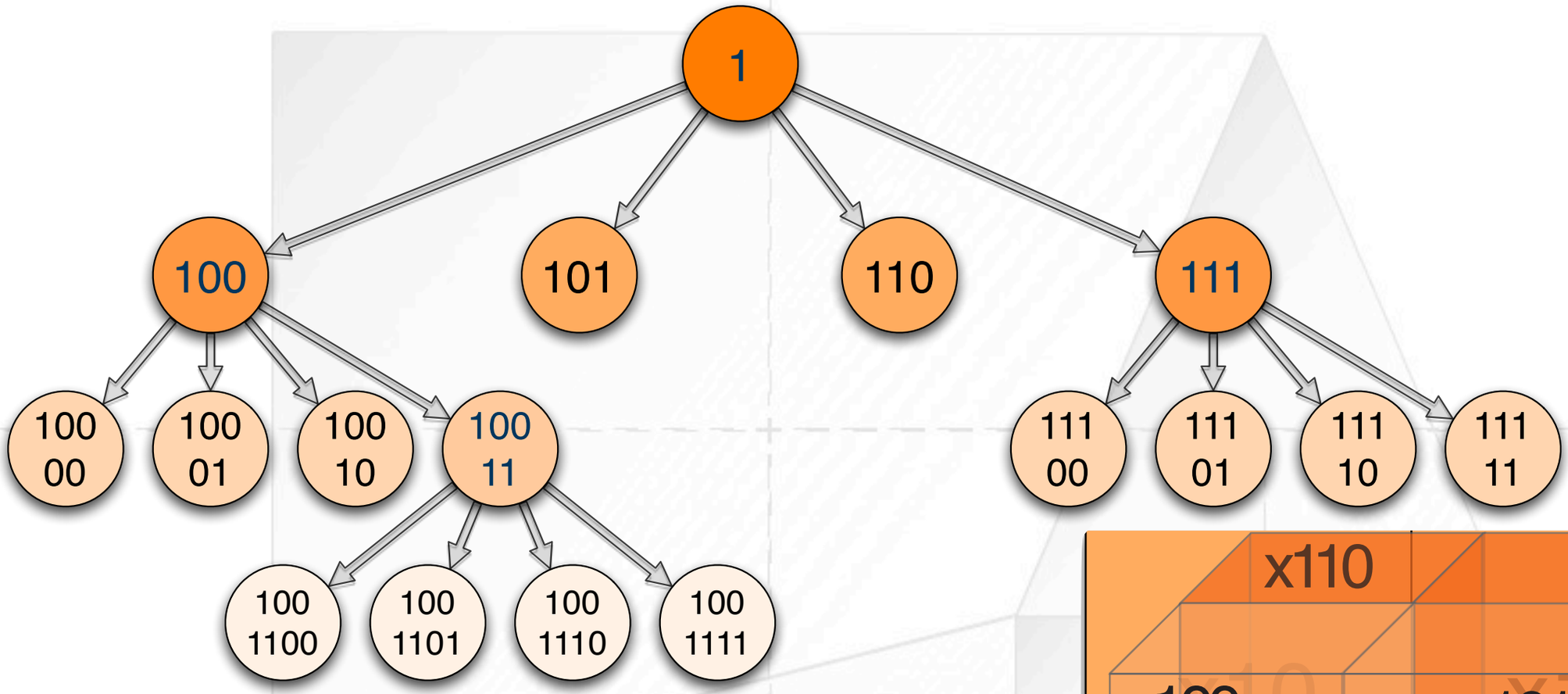
Hierarchic representation

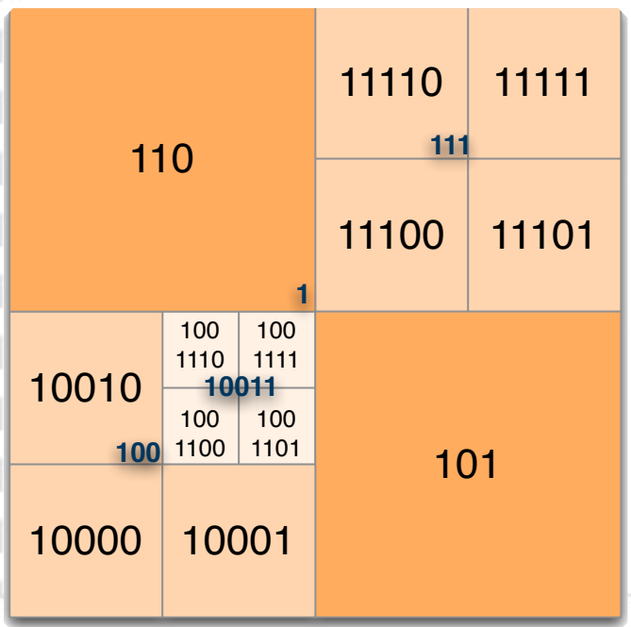






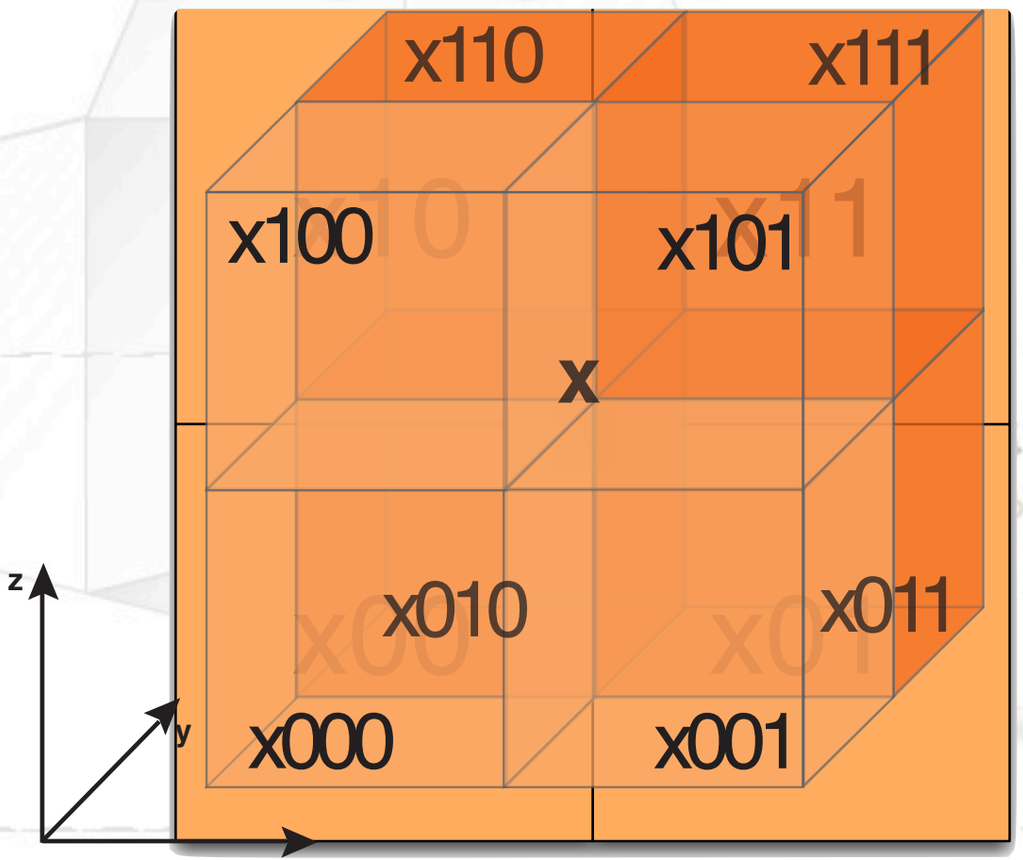
Hierarchic Morton codes





Node's sons: add n-bits suffix

Depth: (code length - 1)/n

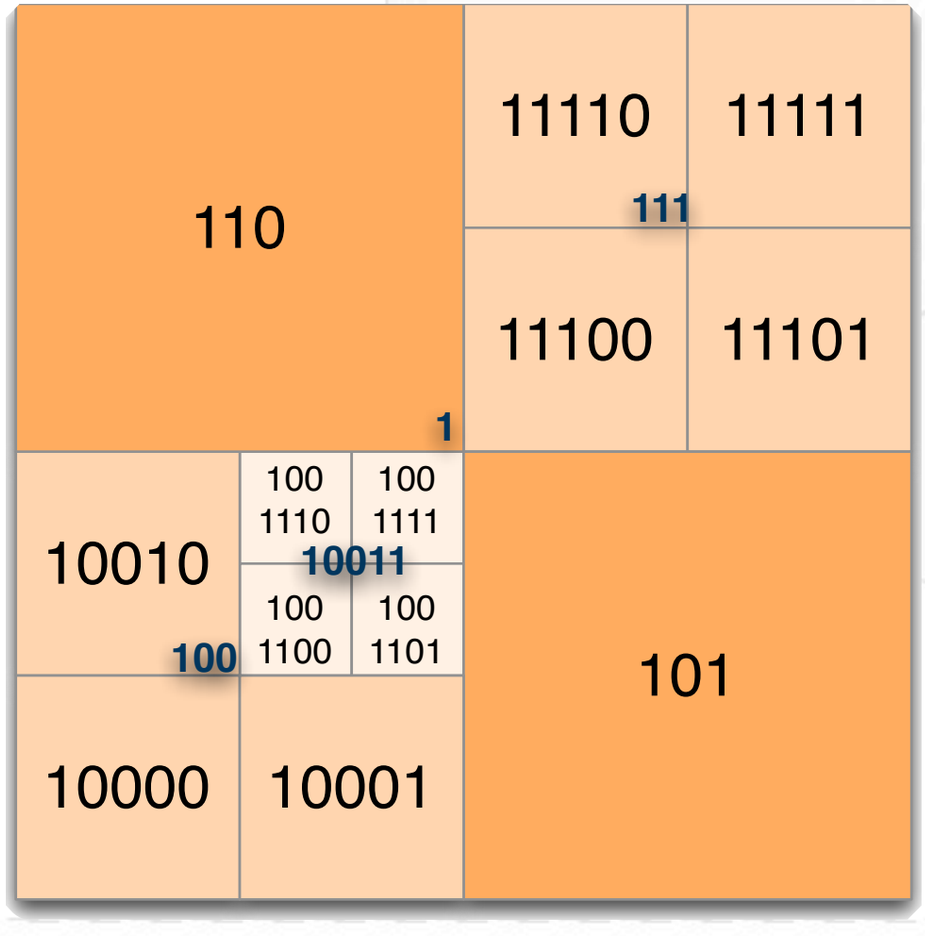


Geometric Morton codes

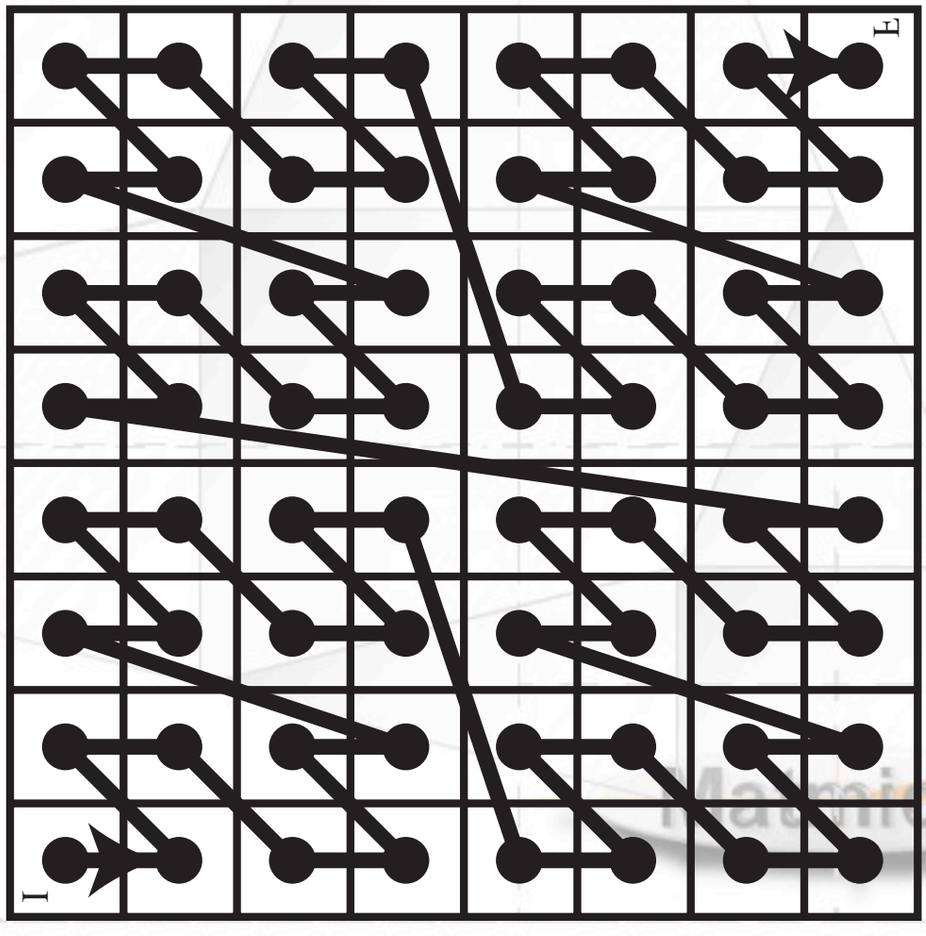
$$center(n) = (x, y) \in [0, 1]^2$$

$$x = 0.x_1 x_2 \dots x_l \dots x_M \qquad y = 0.y_1 y_2 \dots y_l \dots y_M$$

$$k_l(n) = \bar{0}\ 1\ y_1 x_1\ y_2 x_2\ \dots\ y_l x_l$$



level *l*=3

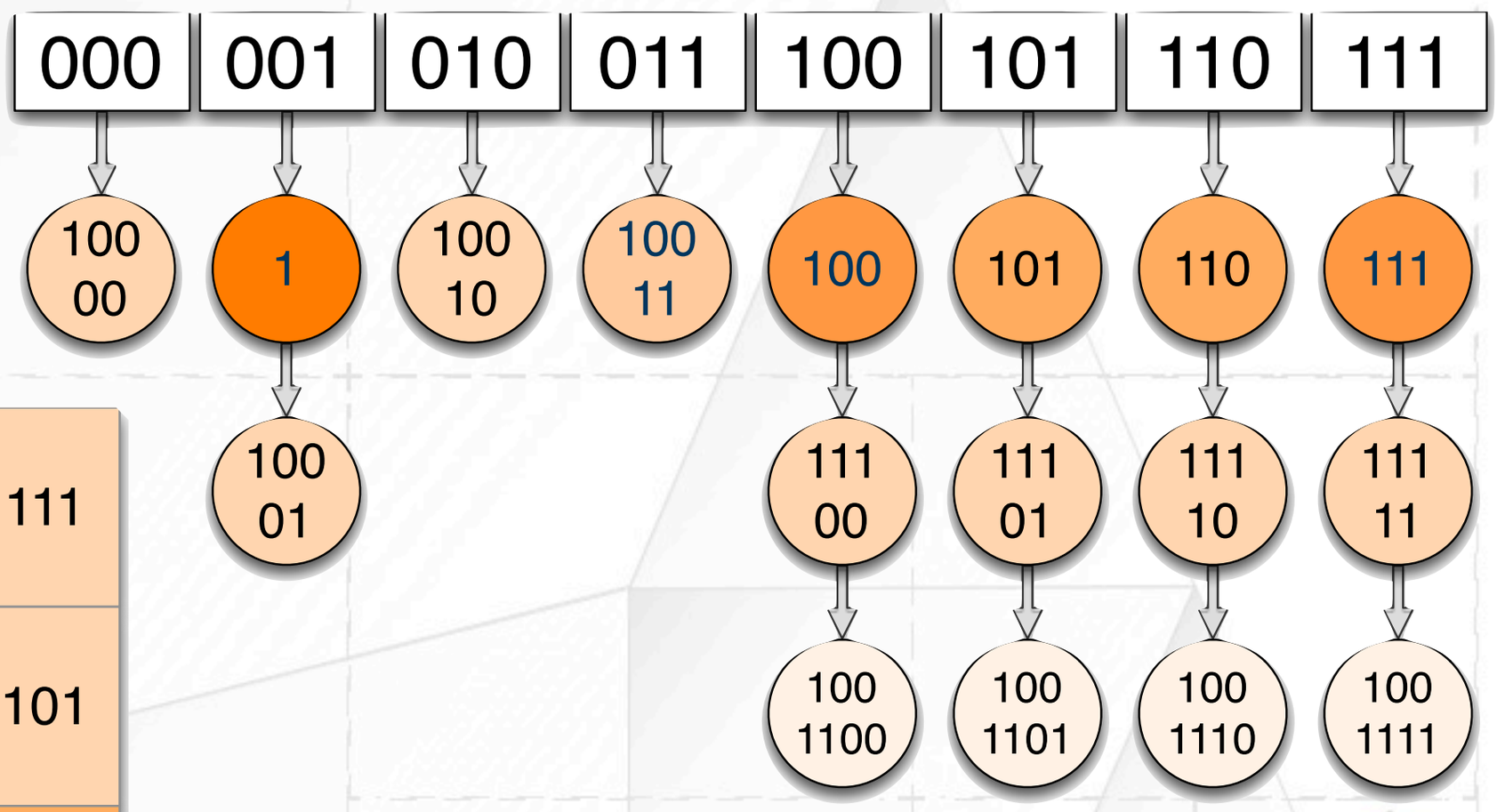


Pointerless 2^n-trees

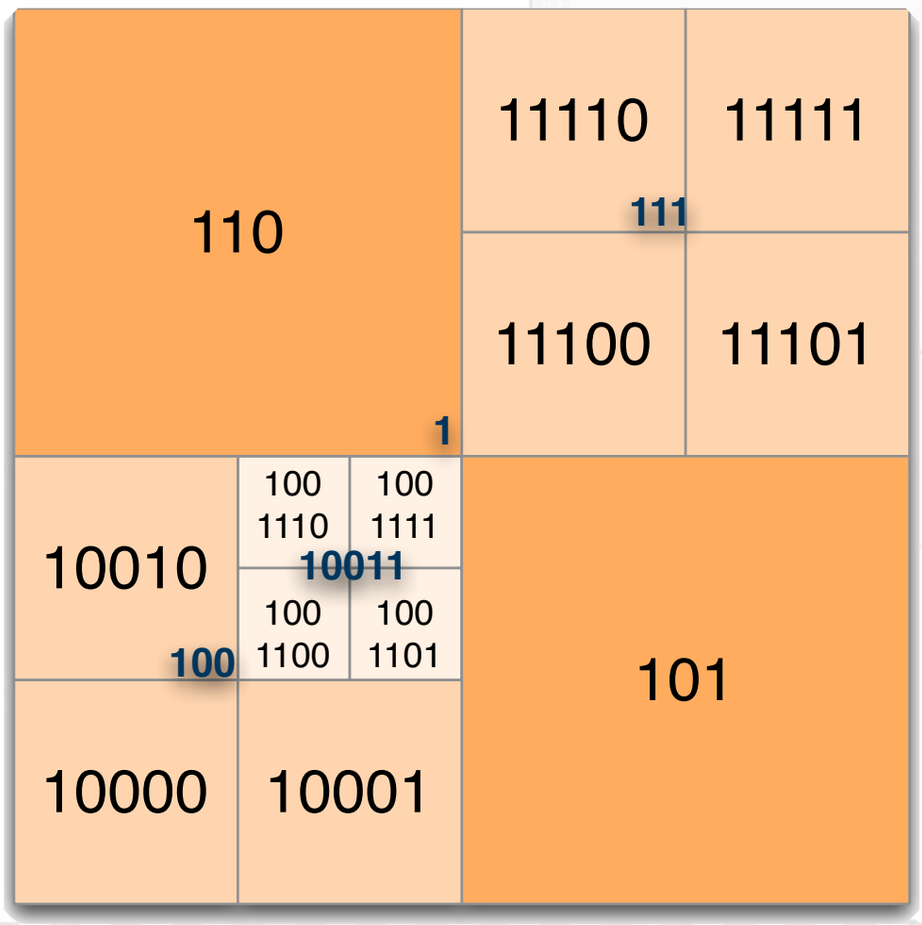
less memory

parallelizable

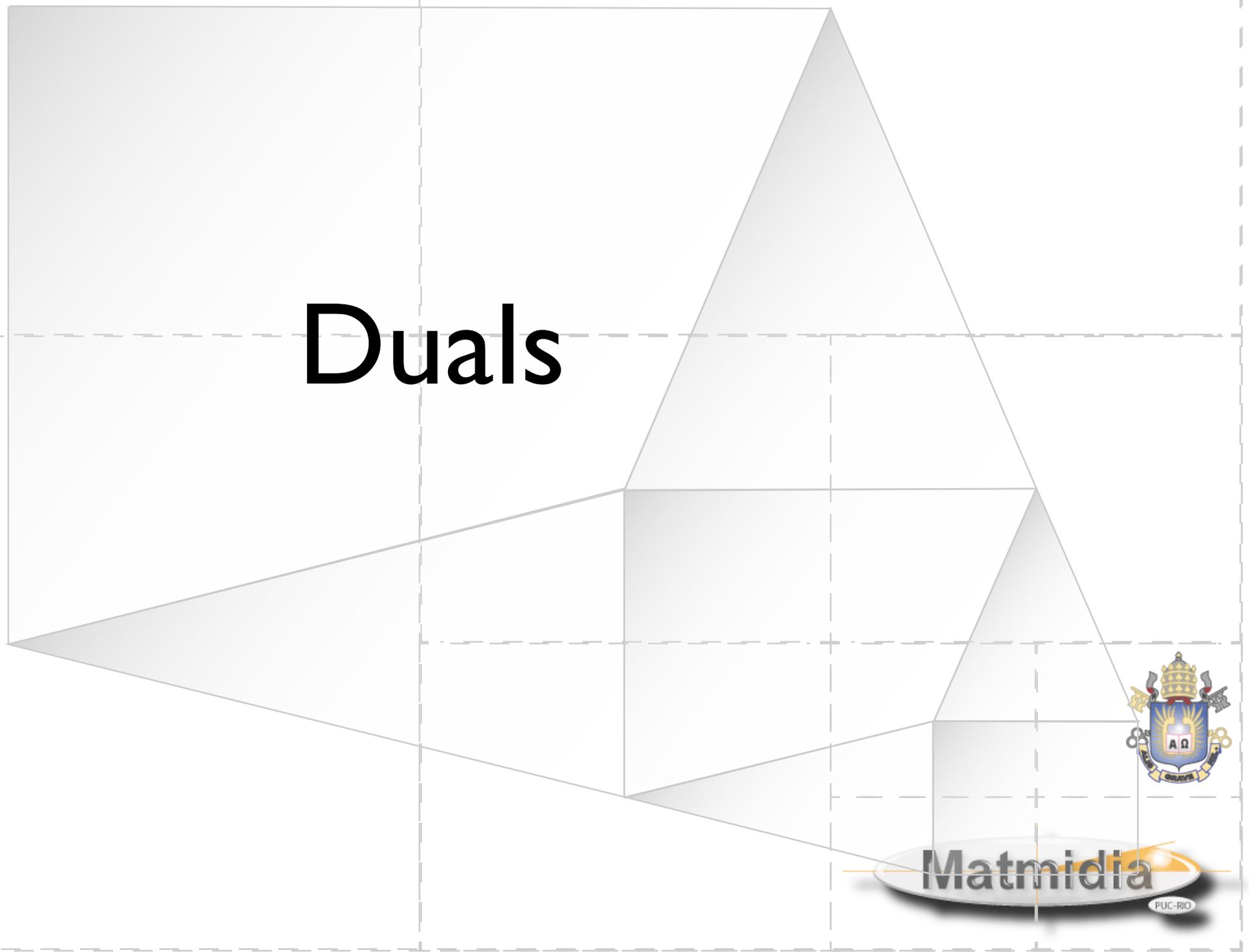
direct access to leaves



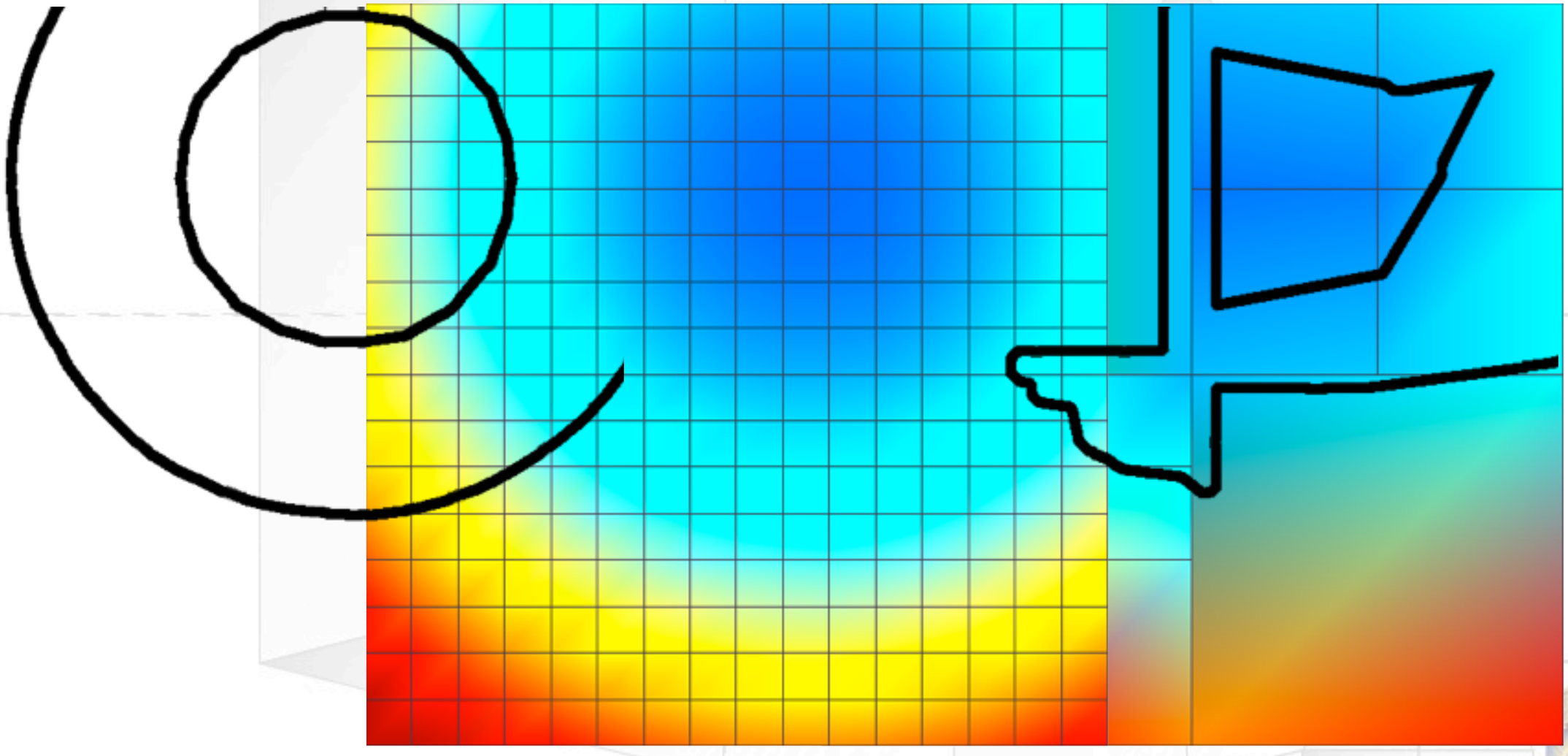
Hashtable



Duals

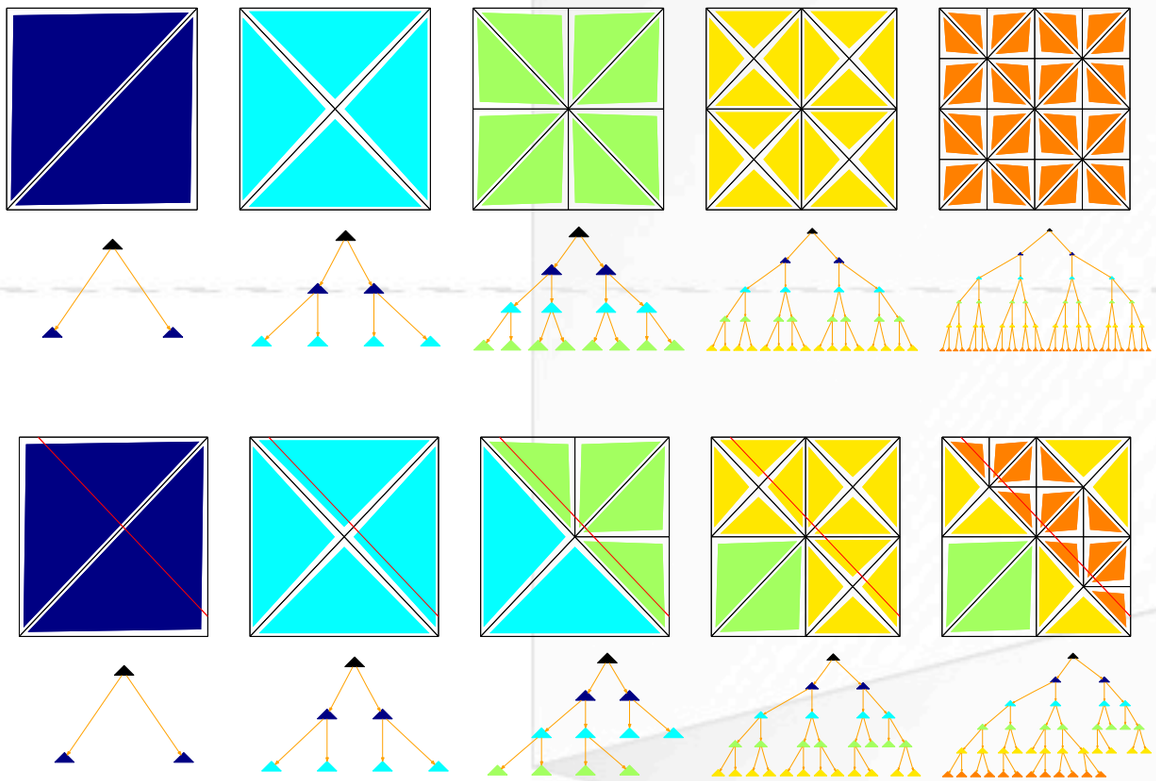


Octree discontinuity



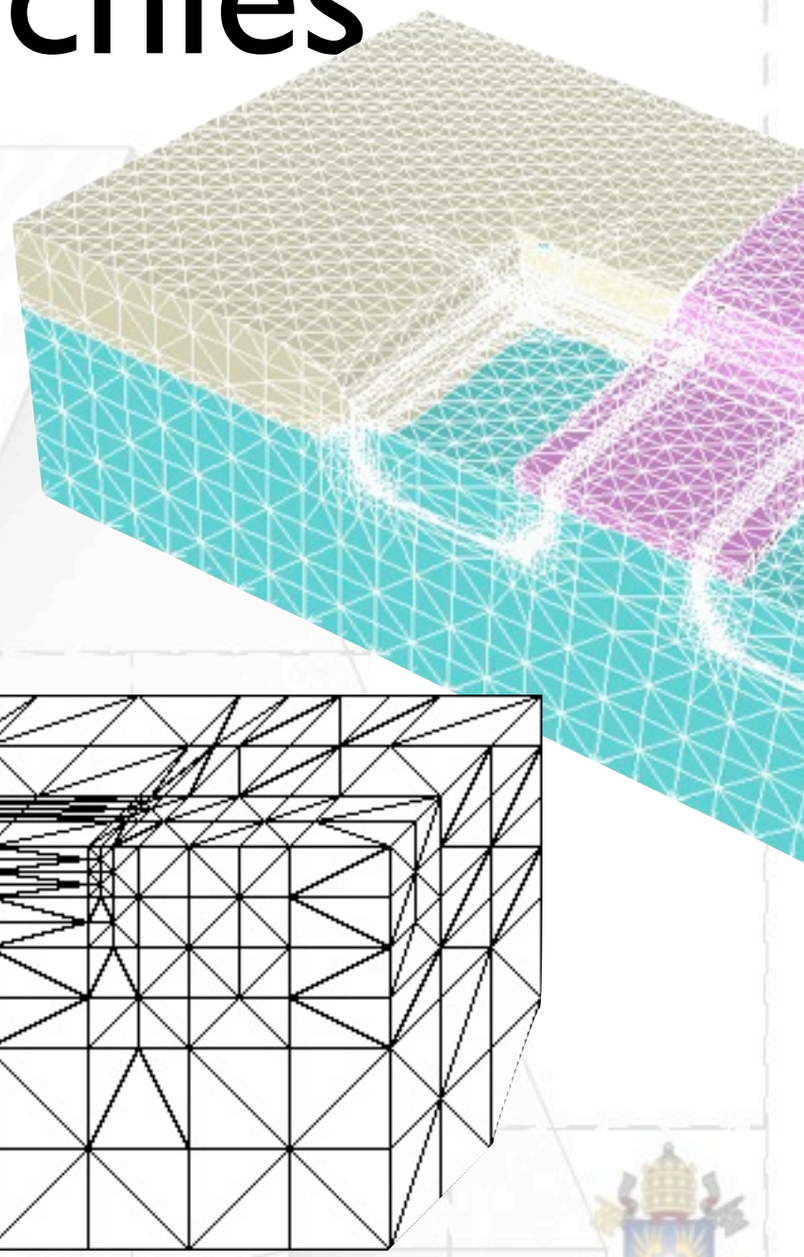
per-leaf interpolation not continuous

Continuous hierarchies



Multi-triangulation

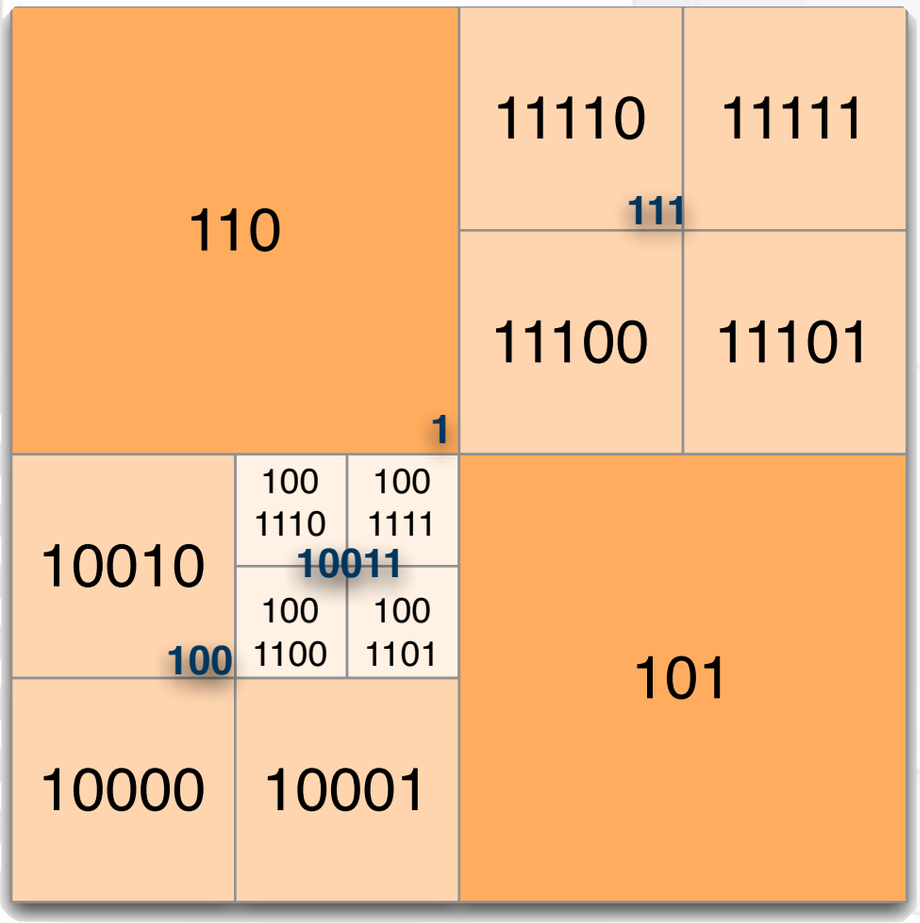
(image from V. Mello *et al*.)

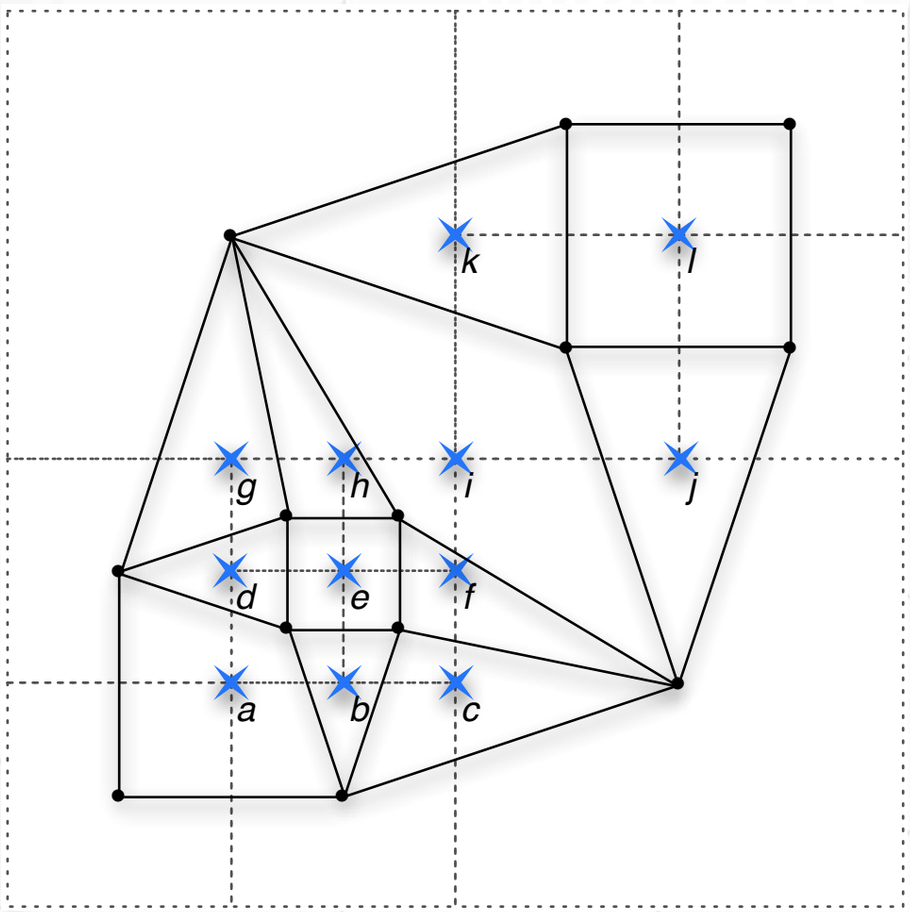


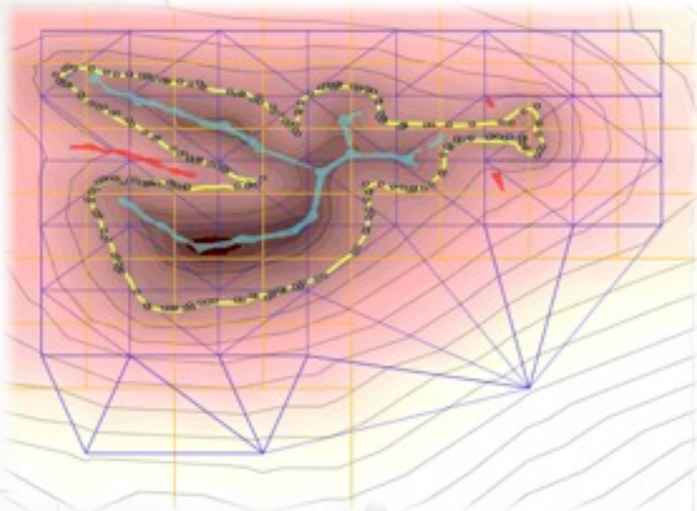
T-Junctions

(image from T. Chen - Camino)

Octree duals







Finite Element Method

(image from A. Sharf *et al*.)

octree leaf ⇔ dual vertex

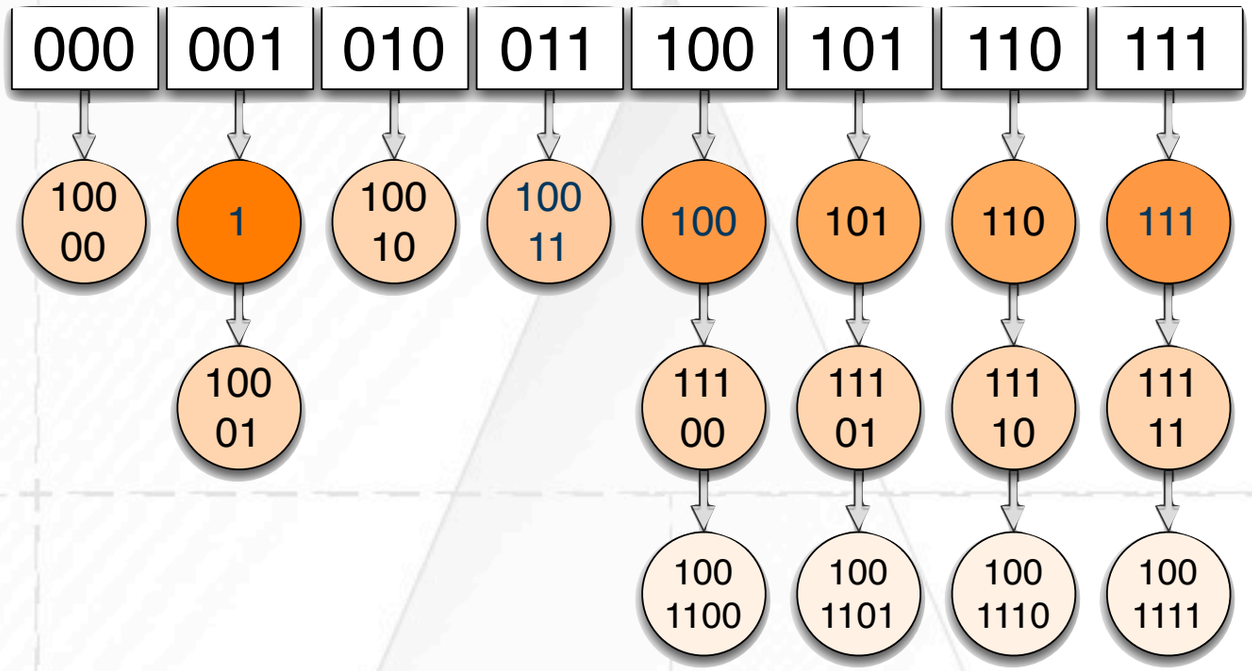
leaf vertex ⇔ dual facet

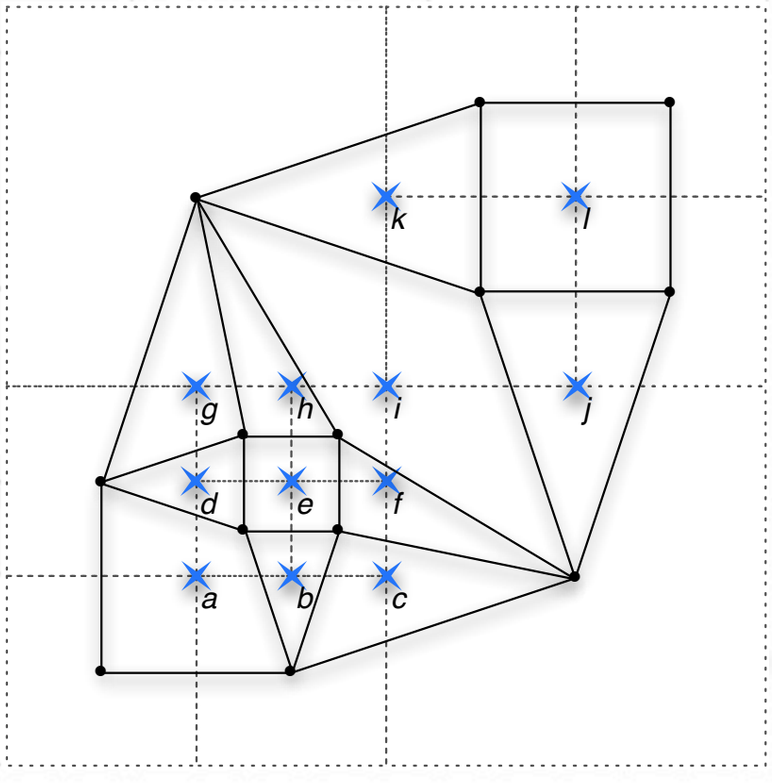
Contributions

Efficient generation of dual

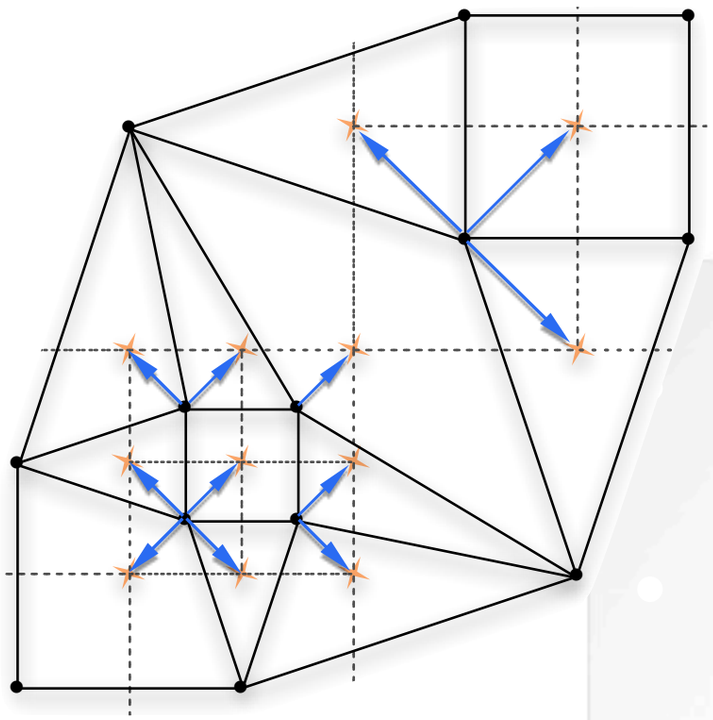
Pointerless structure

Average 3x faster with less memory!





Related work



Related work

JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of Hermite data. *Siggraph* (2002).

SCHAEFER S., WARREN J.: Dual Marching Cubes: primal contouring of dual grids. *Pacific Graphics* (2004).

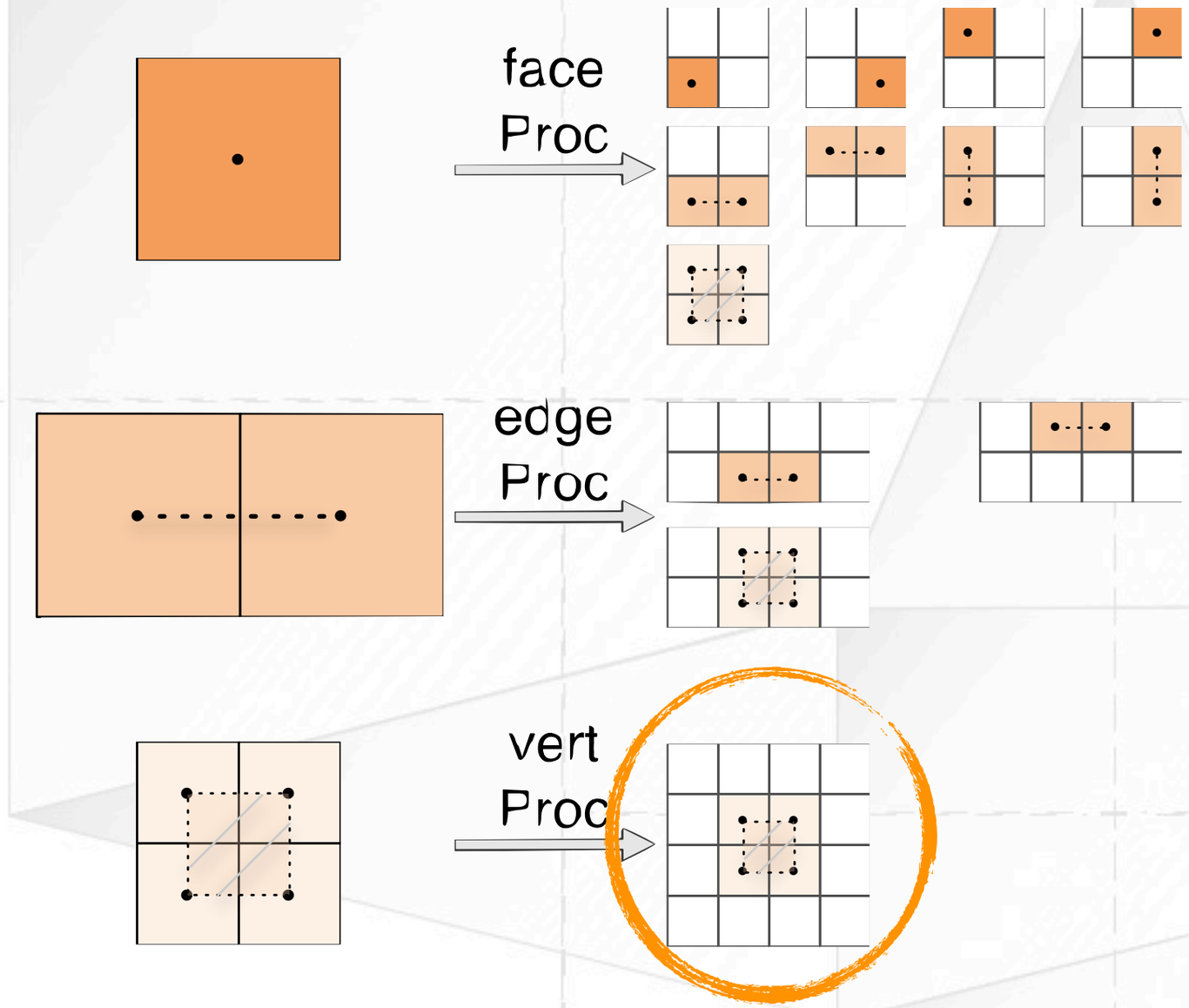
PAIVA A., LOPES H., LEWINER T., de FIGUEIREDO L.H.: Robust adaptive meshes for implicit surfaces. *Sibgrapi* (2006)

LEÓN A., TORRES J. C., VELASCO F.: Volume octree with an implicitly defined dual grid. *Computers & Graphics* (2008).

CASTRO R., LEWINER T., LOPES H., TAVARES G., BORDIGNON A.: Statistical optimization of octree searches. *Computer Graphics Forum* (2008).

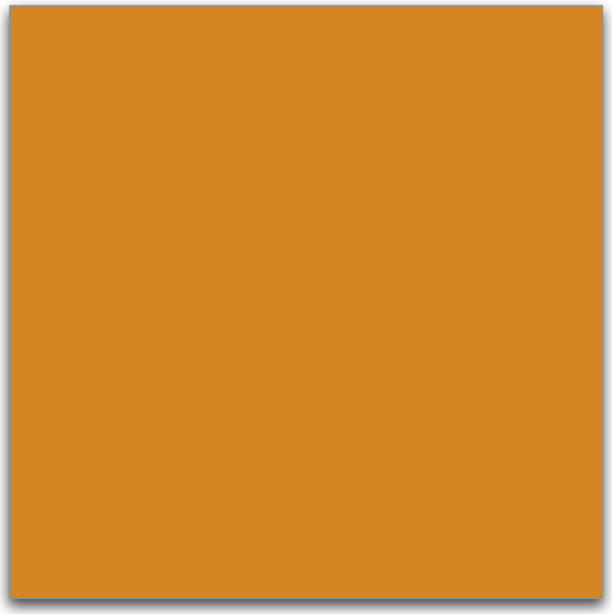
STOCCO L. J., SCHRACK G.: On spatial orders and location codes. *Transactions on Computers* (2009).

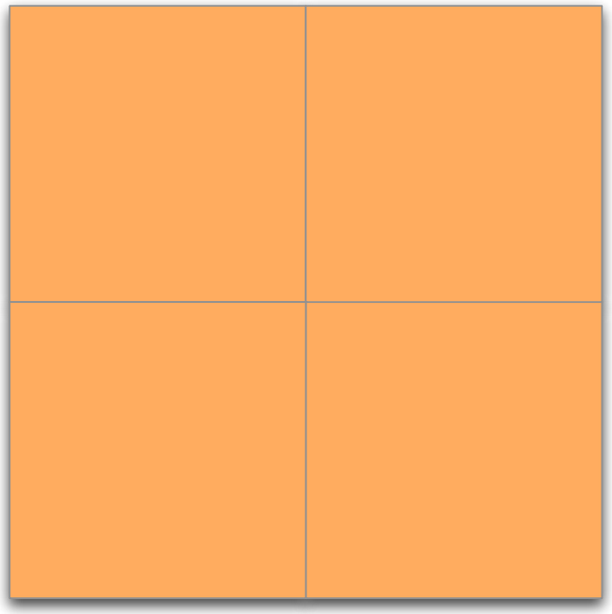
Usual Dual Generation: Recursive

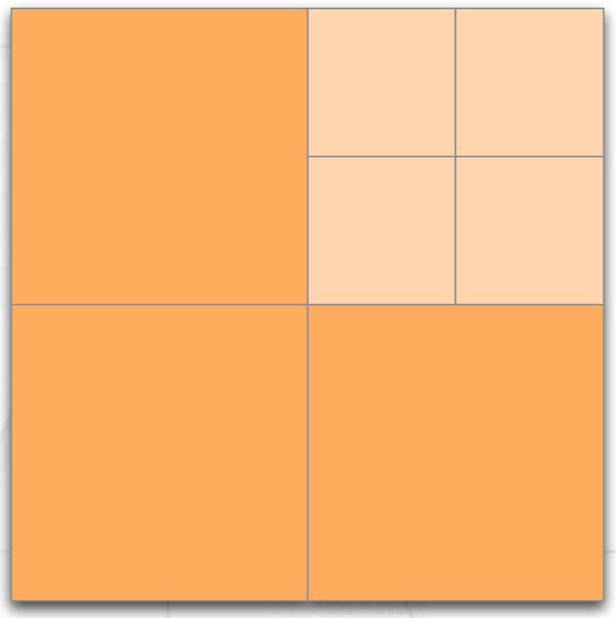


processes each cell of each intermediate level once

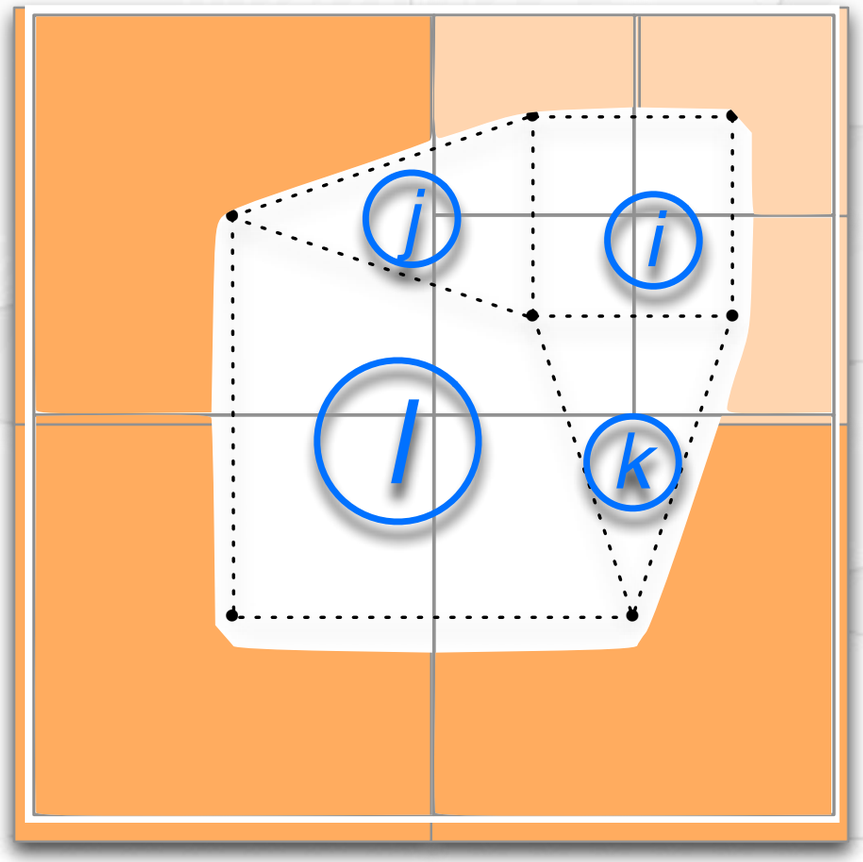
Example



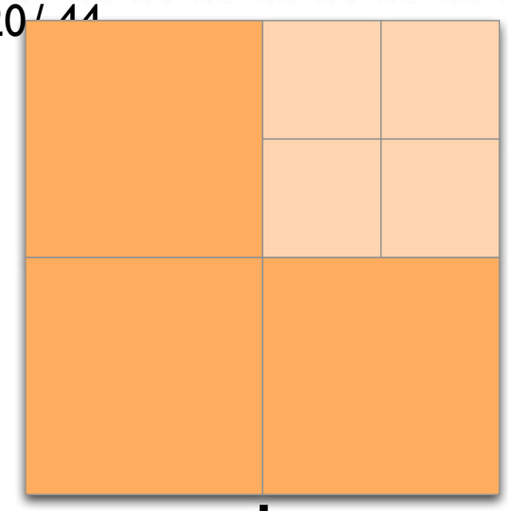




quadree

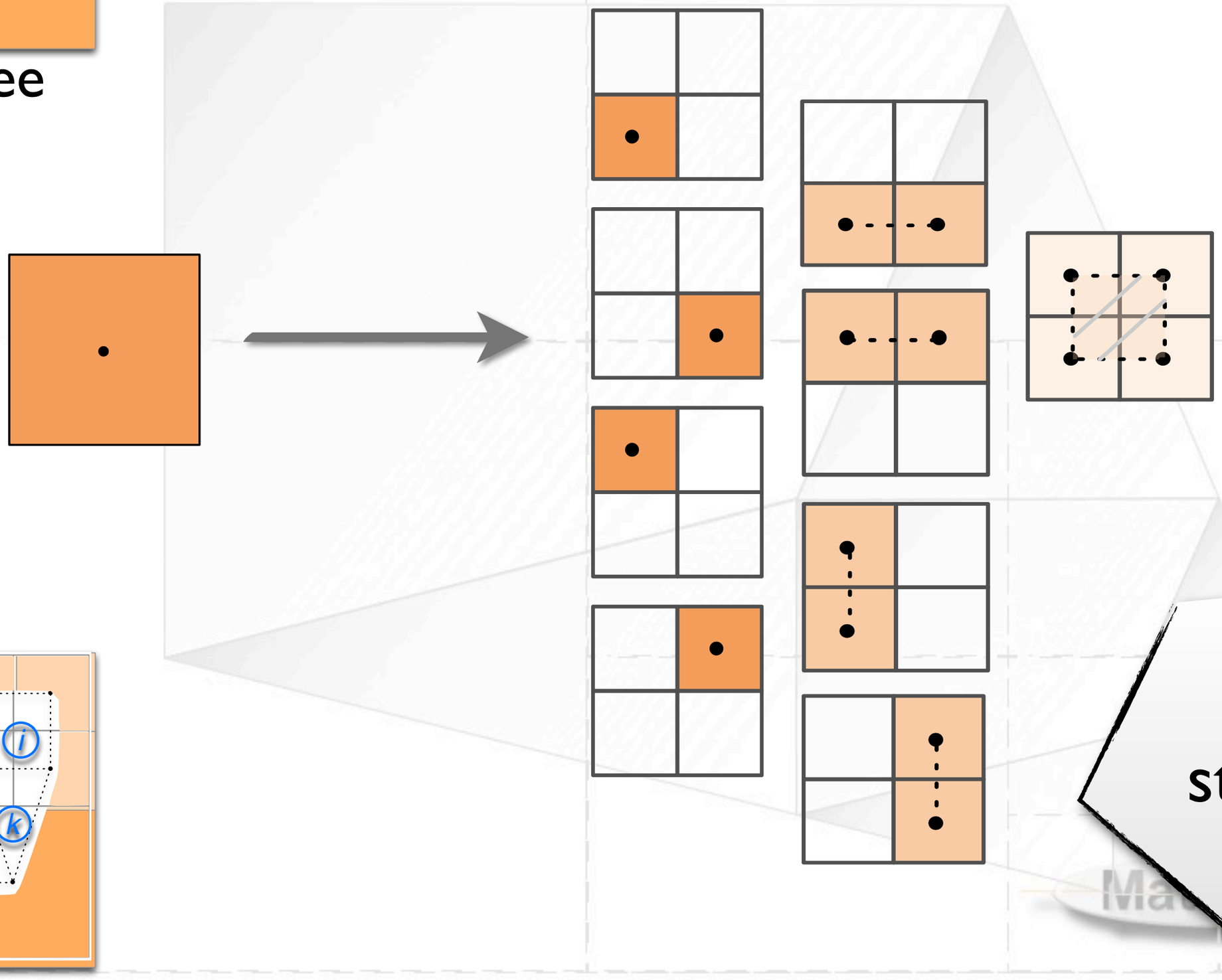


dual

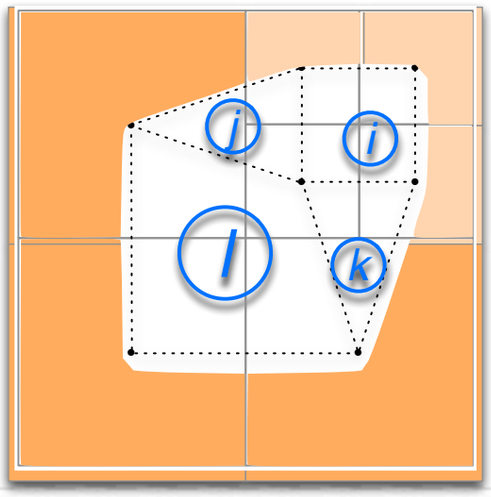


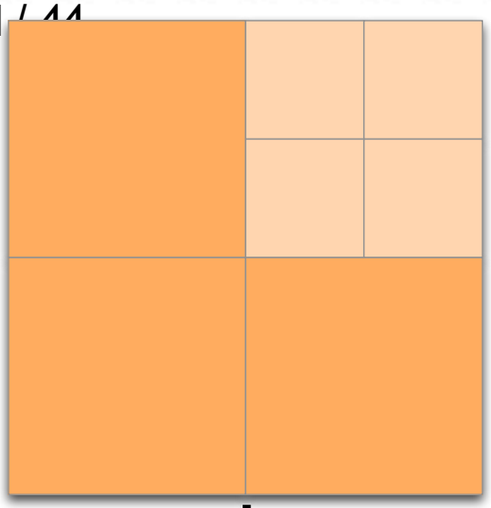
quadtree

Initial FaceProc



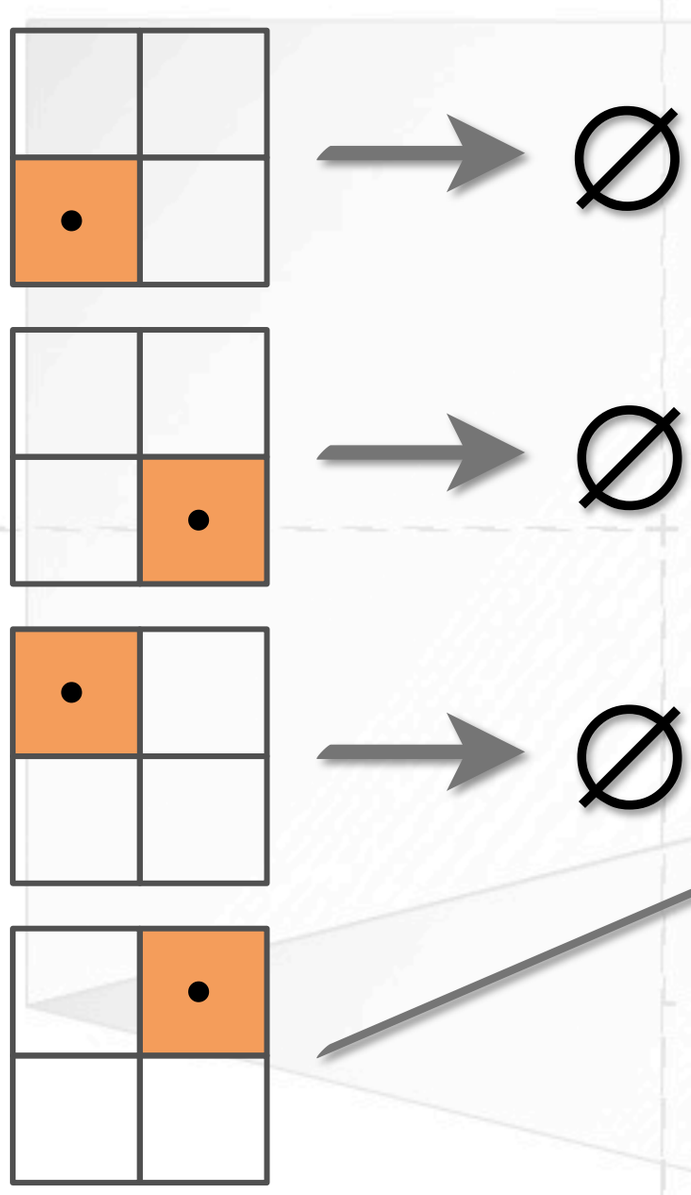
dual

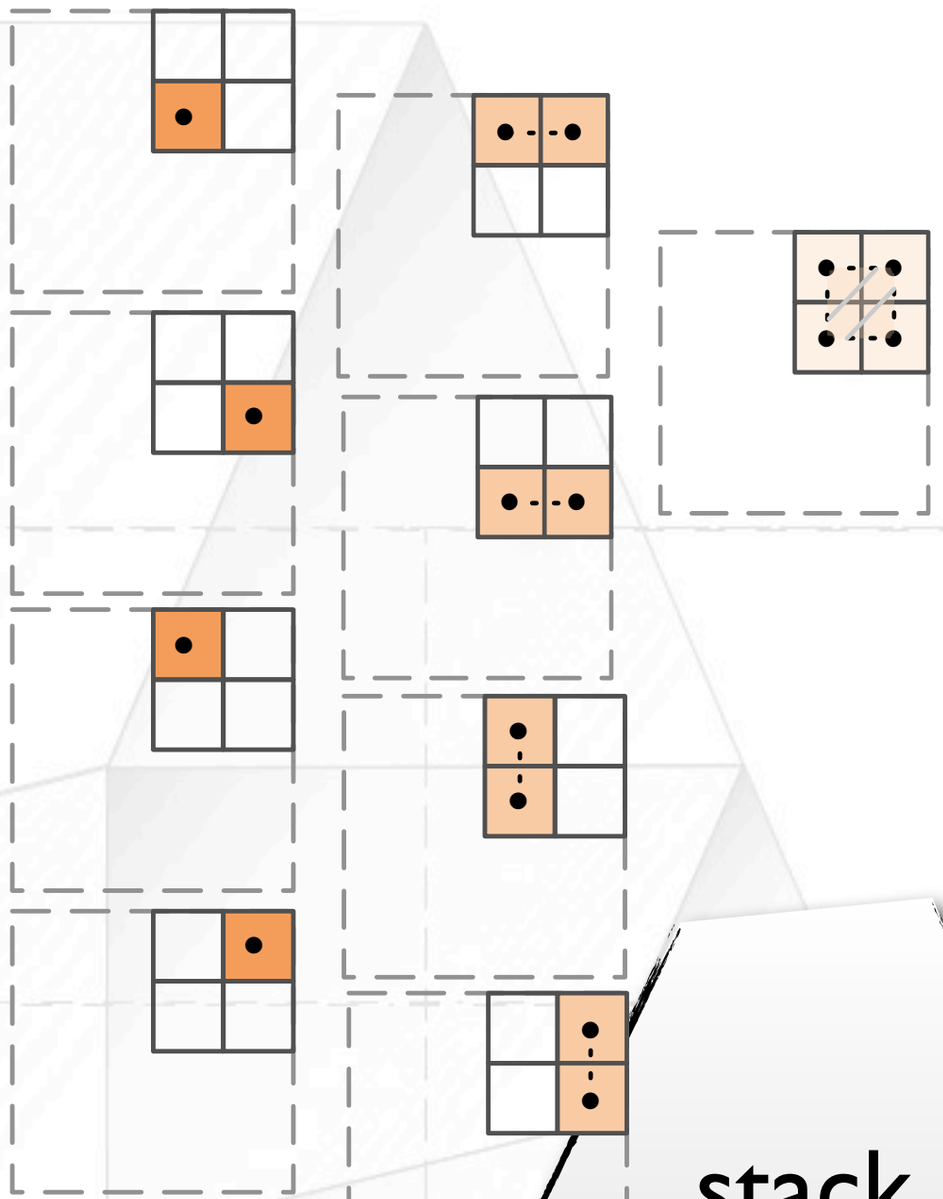




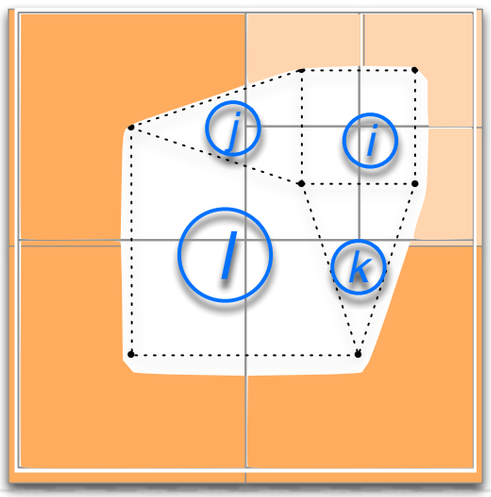
quadtree

Subsequent FaceProc's

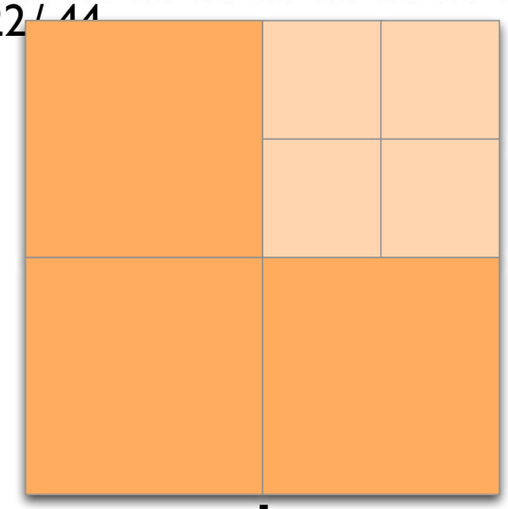




dual

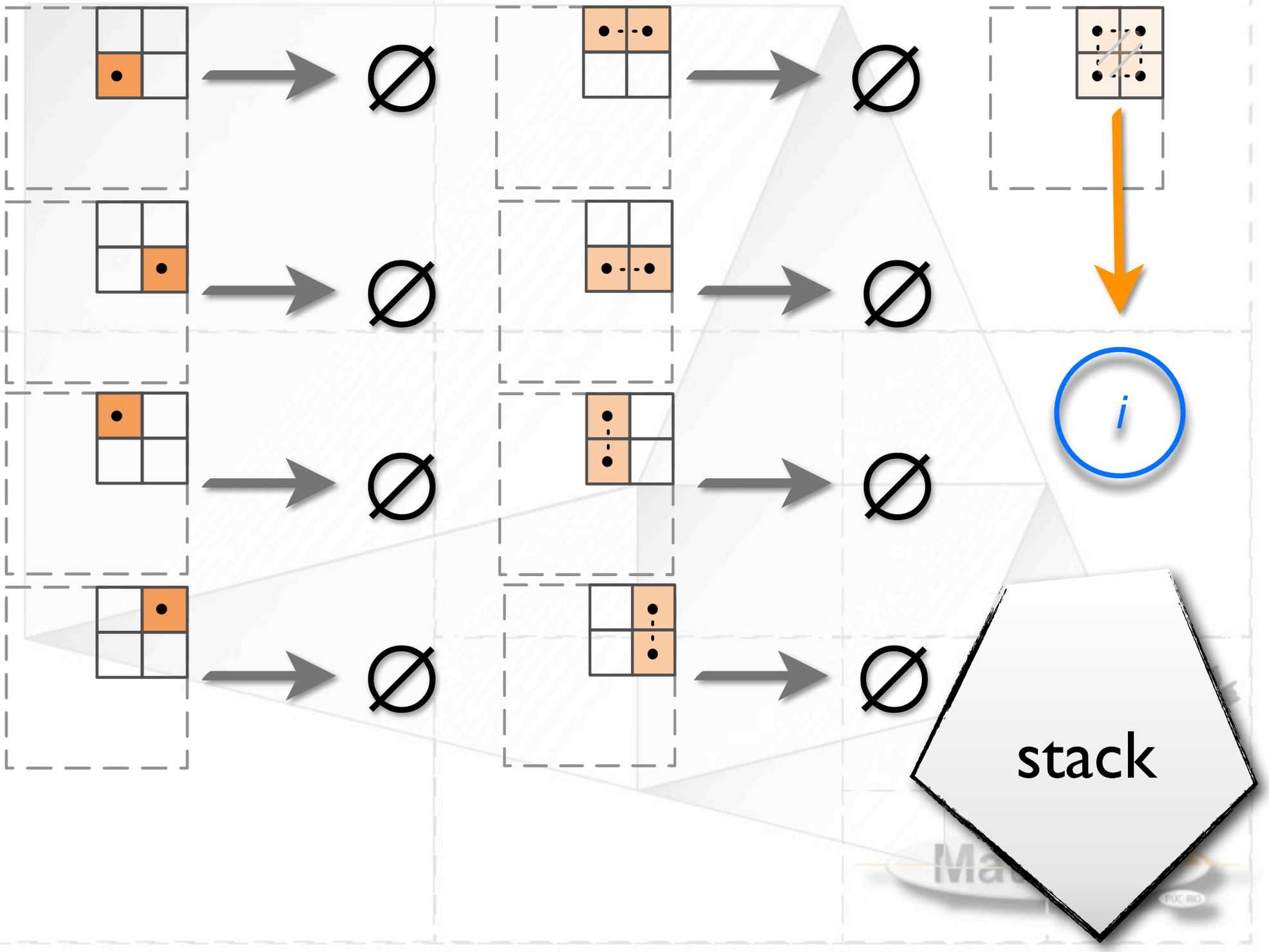


stack

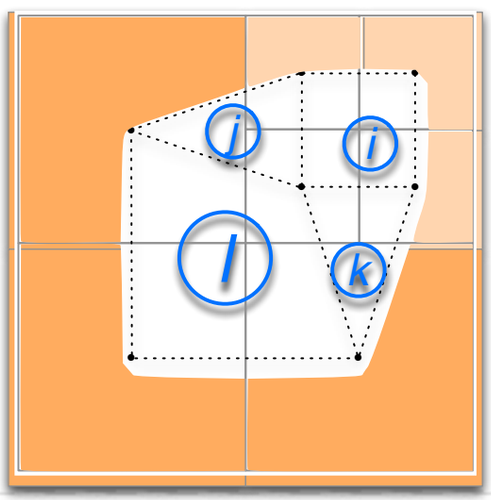


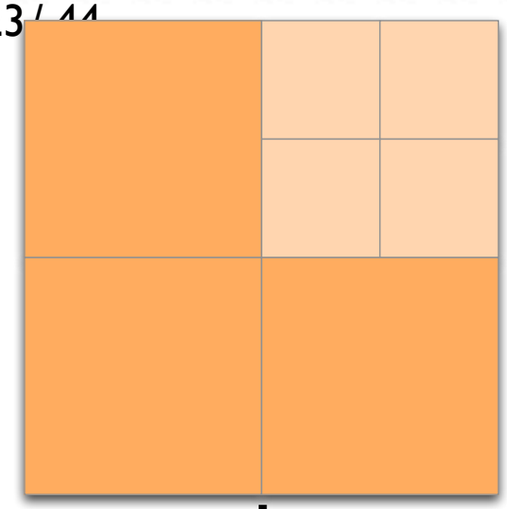
quadtree

Second level



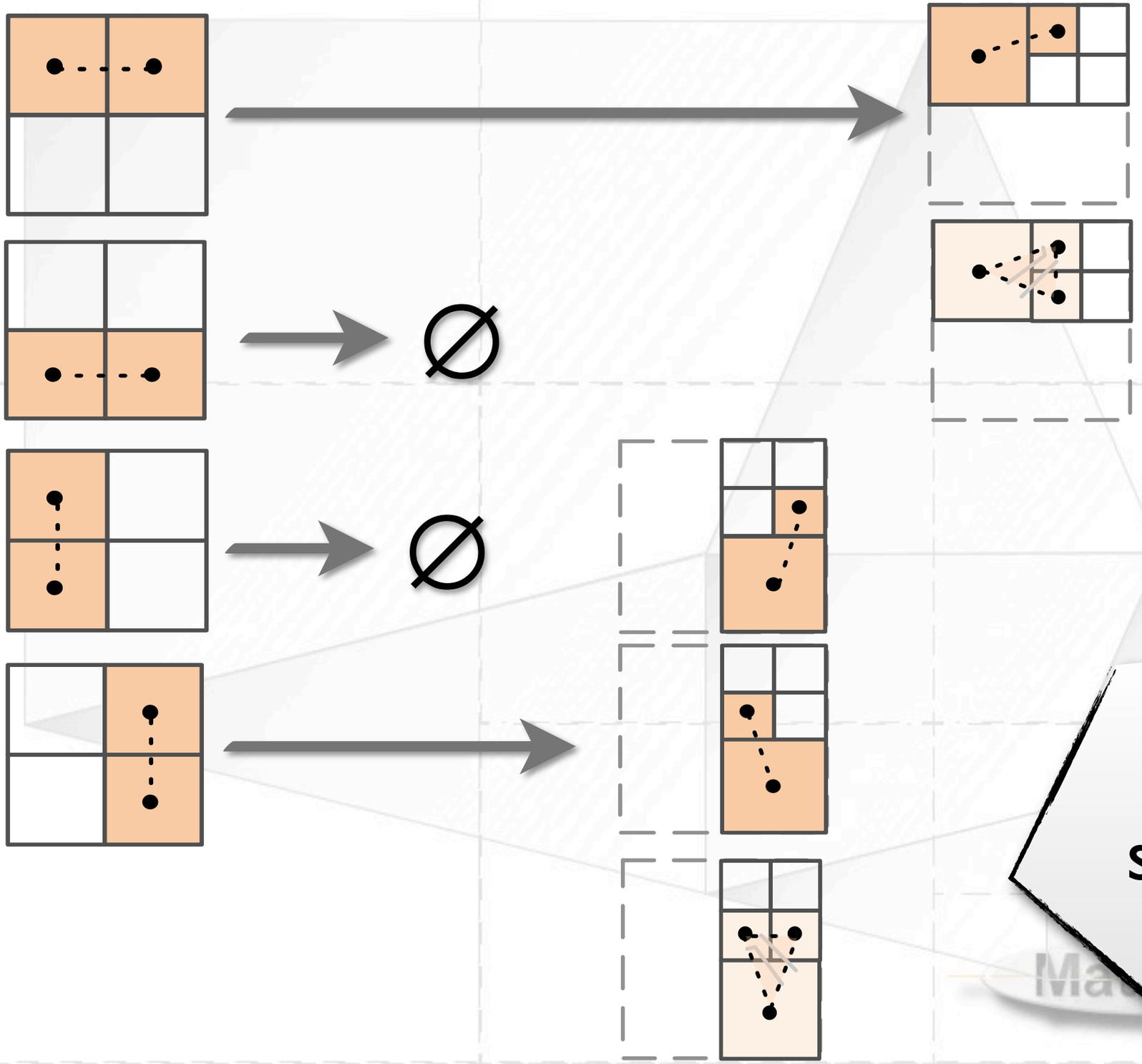
dual



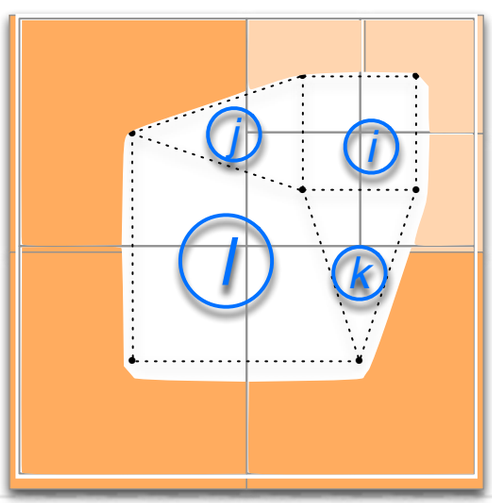


quadtree

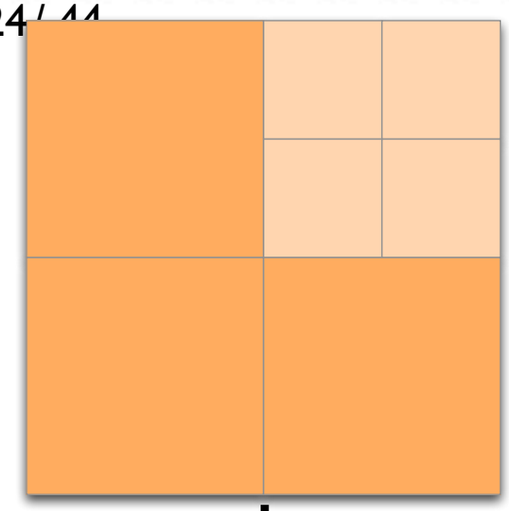
EdgeProc's



dual

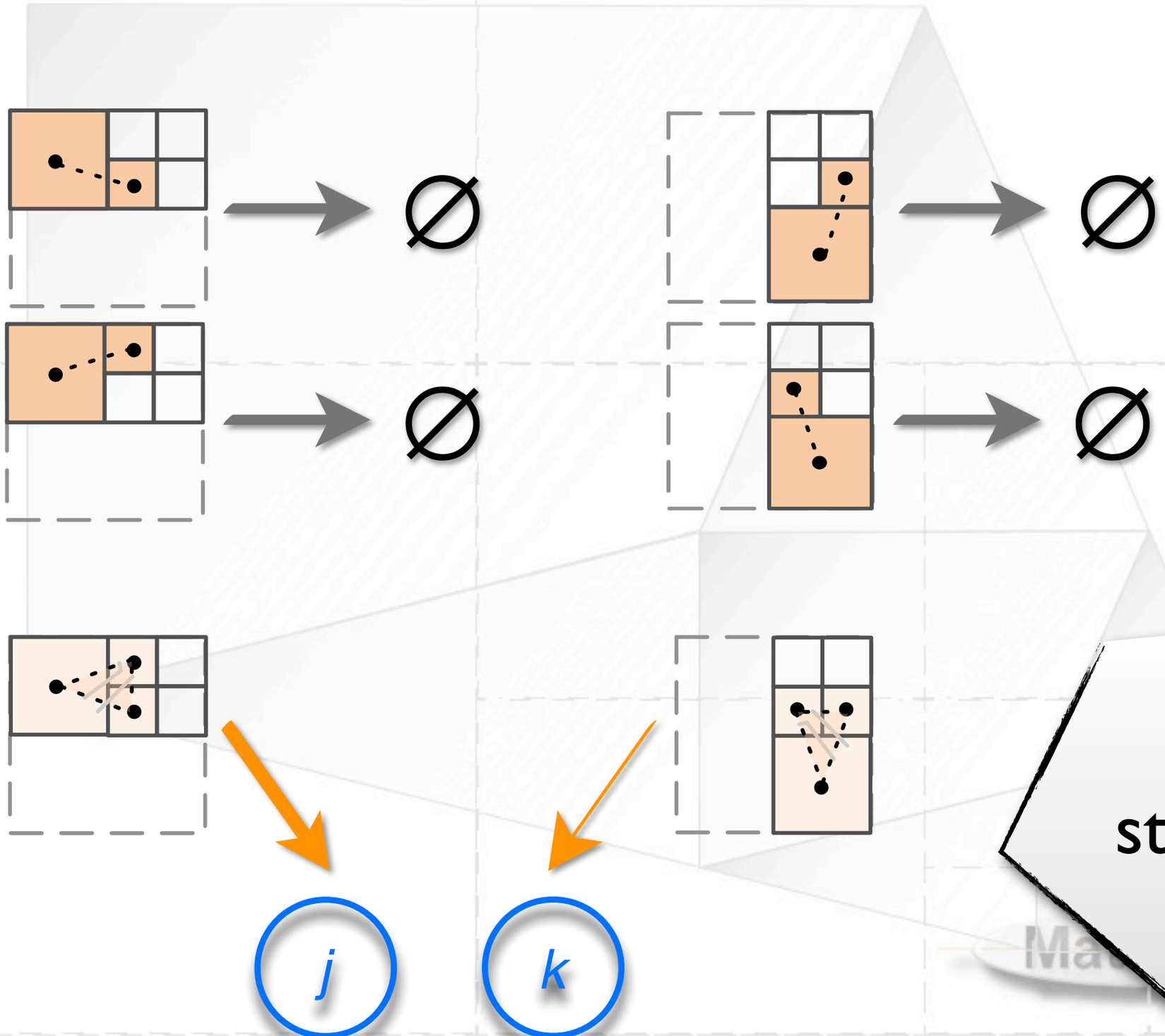


stack

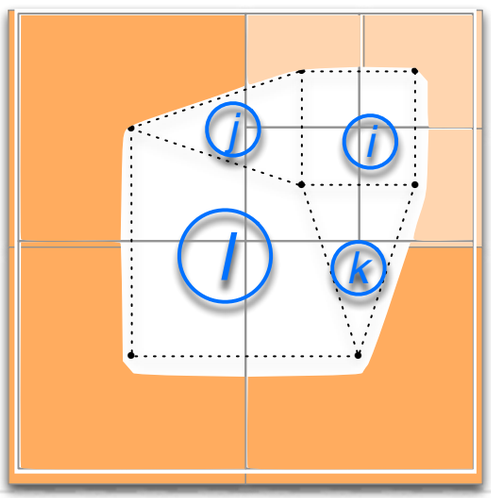


quadtree

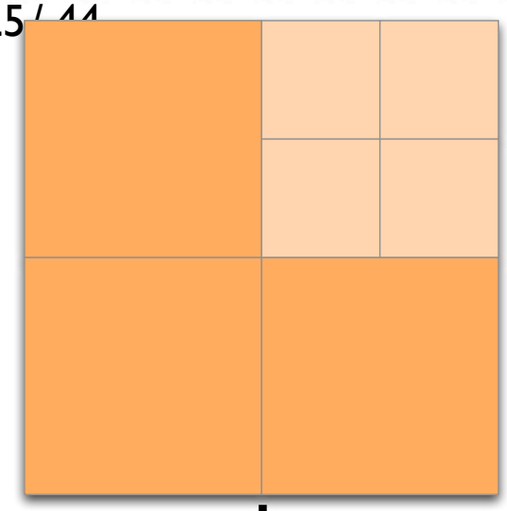
Second level



dual



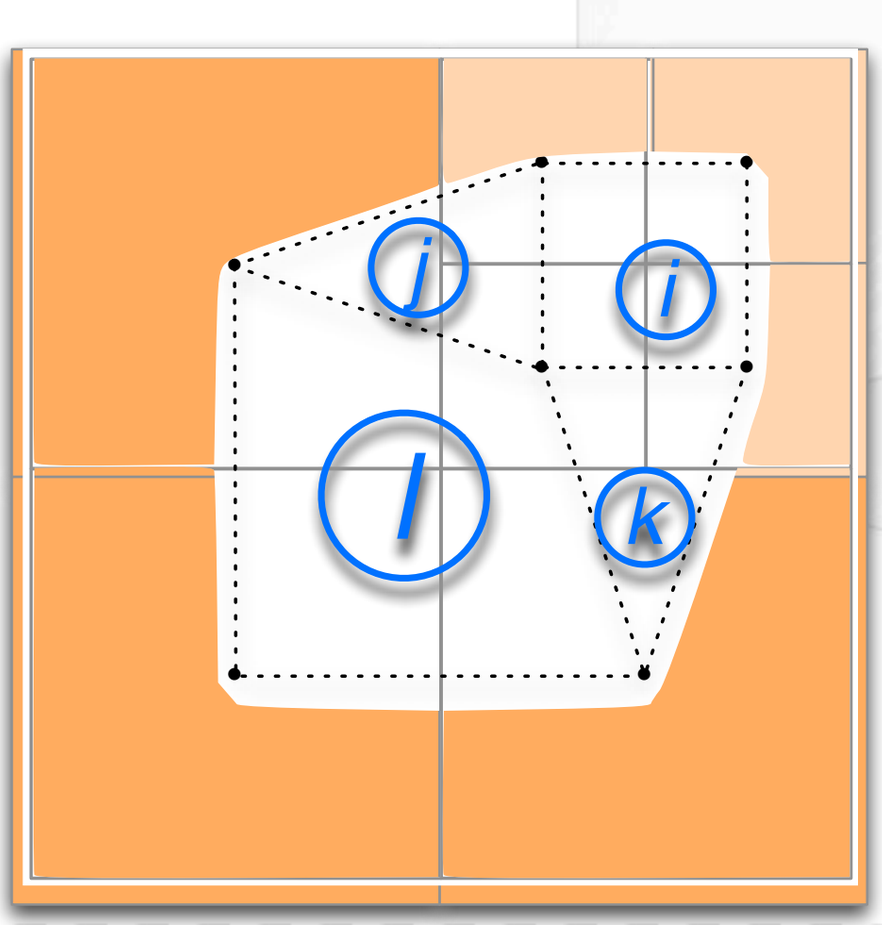
stack

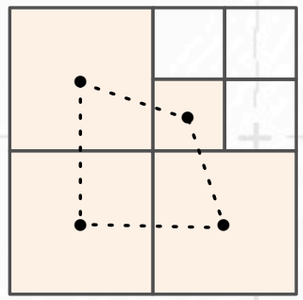


quadtree

Last vertProc

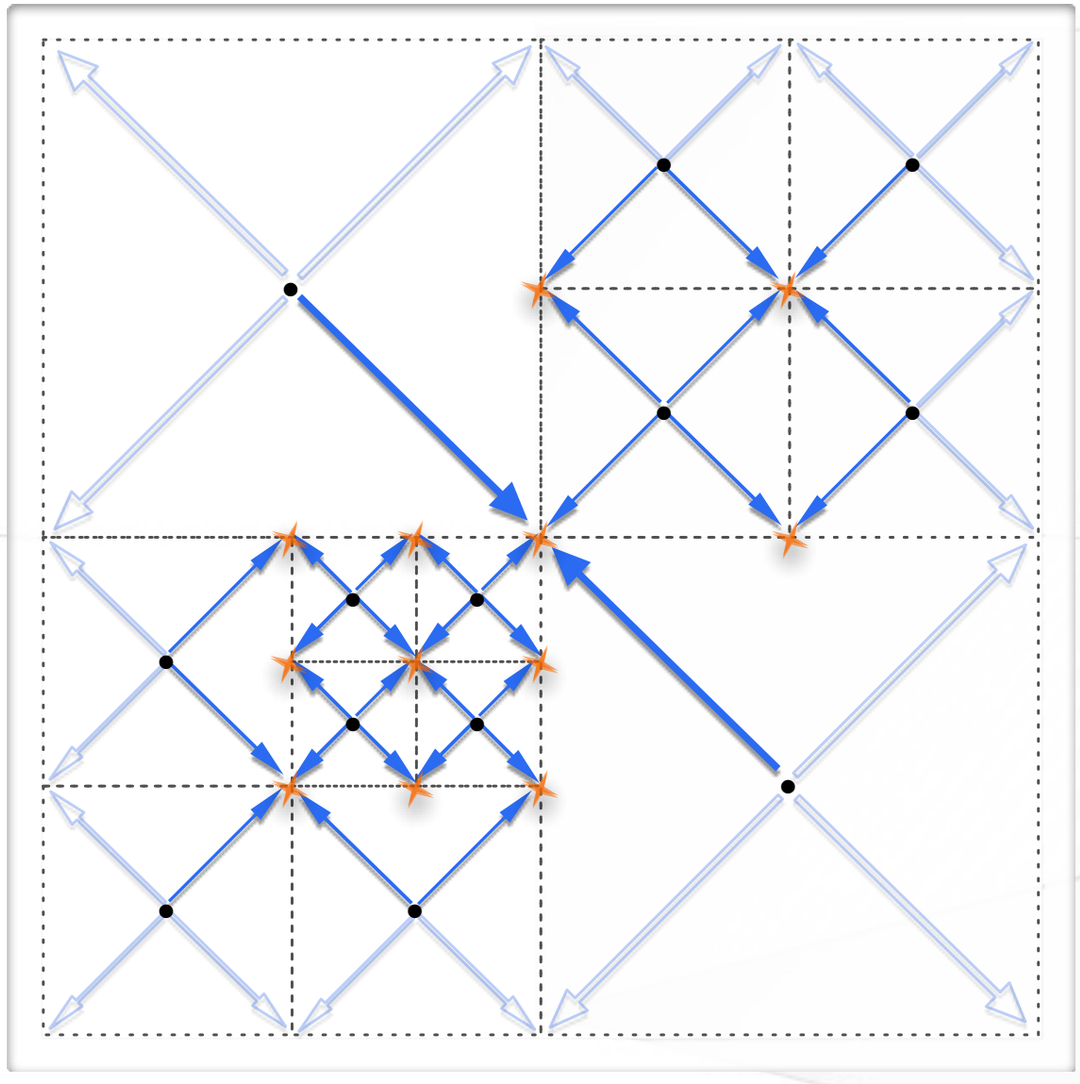
dual

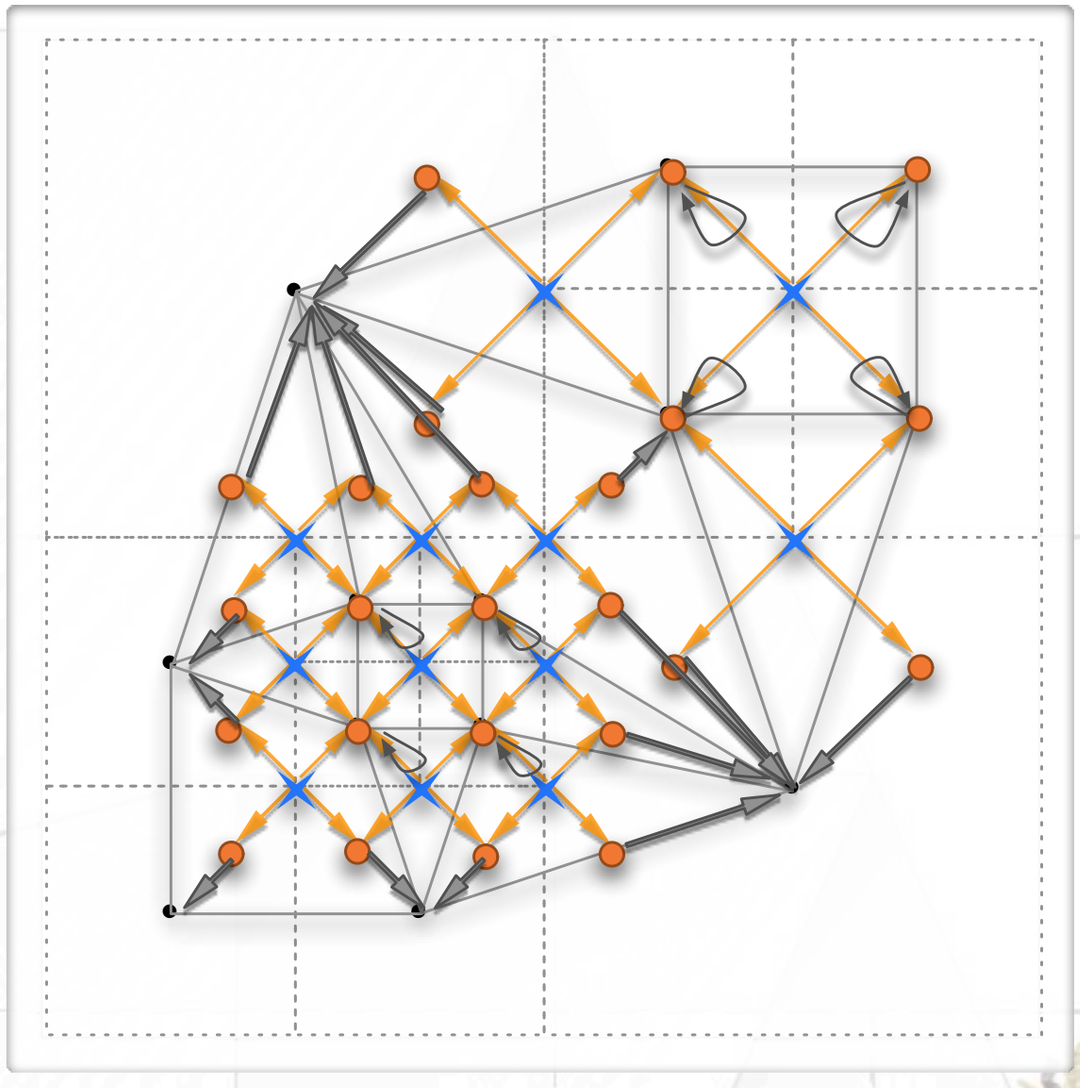




Proposed algorithms

Algorithm overview

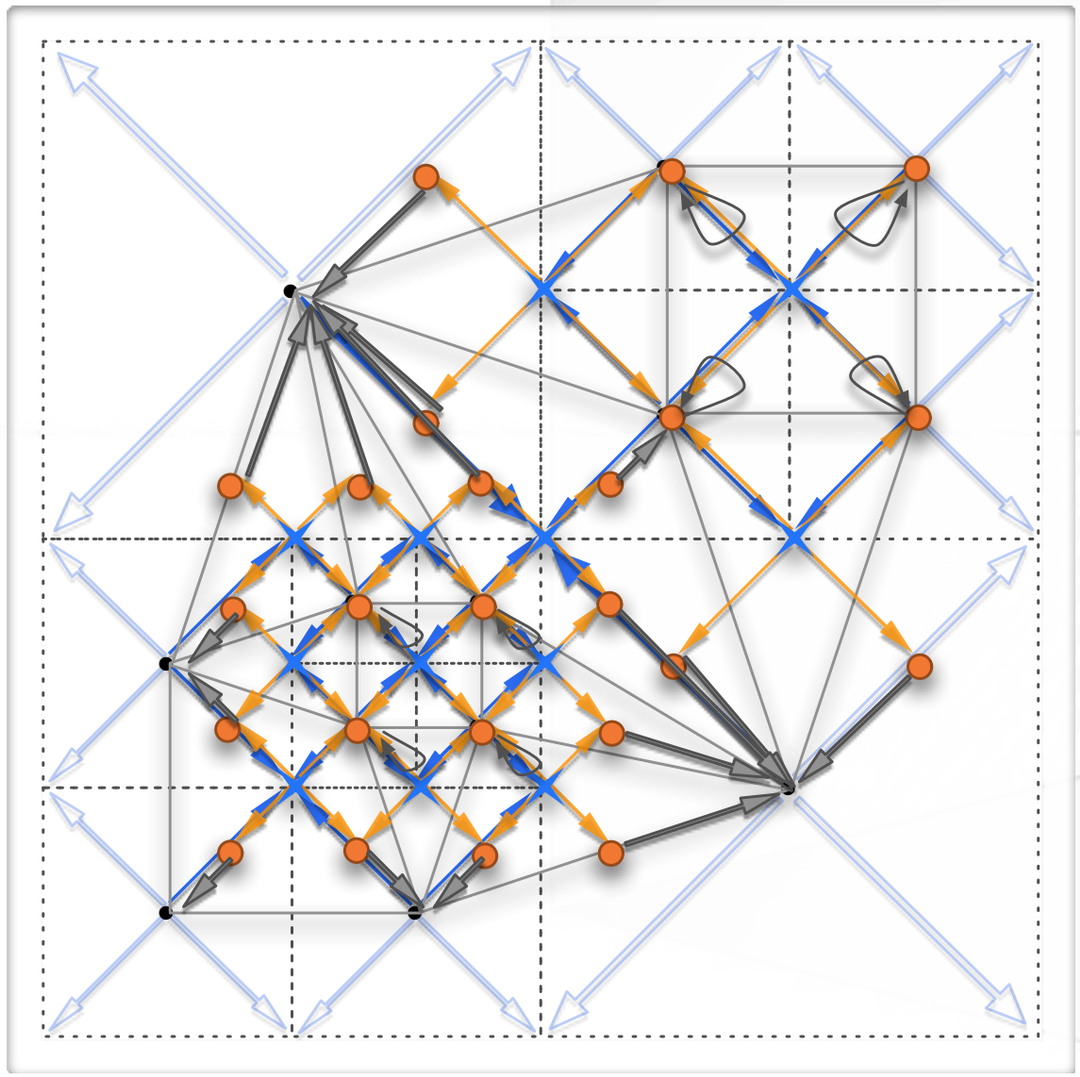




Step 1: generate leaf vertices with deepest level

Step 2: search for leaves around vertices

Main issues



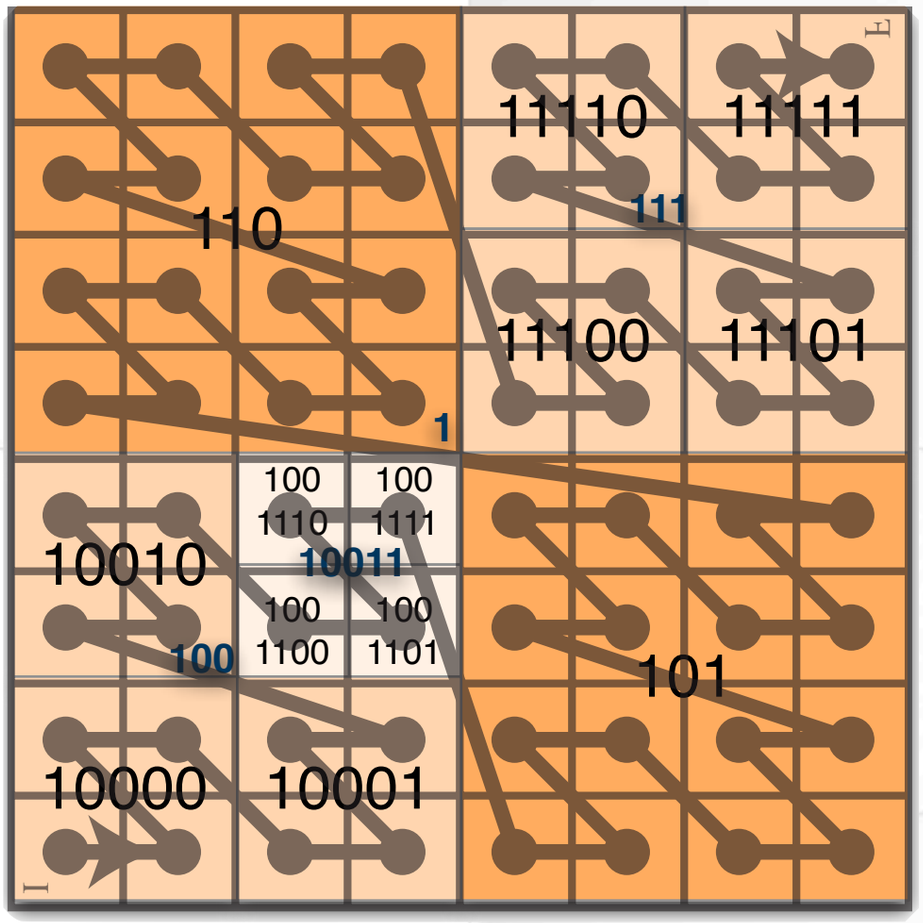
Pointerless representation for vertices

Efficient search for neighbors

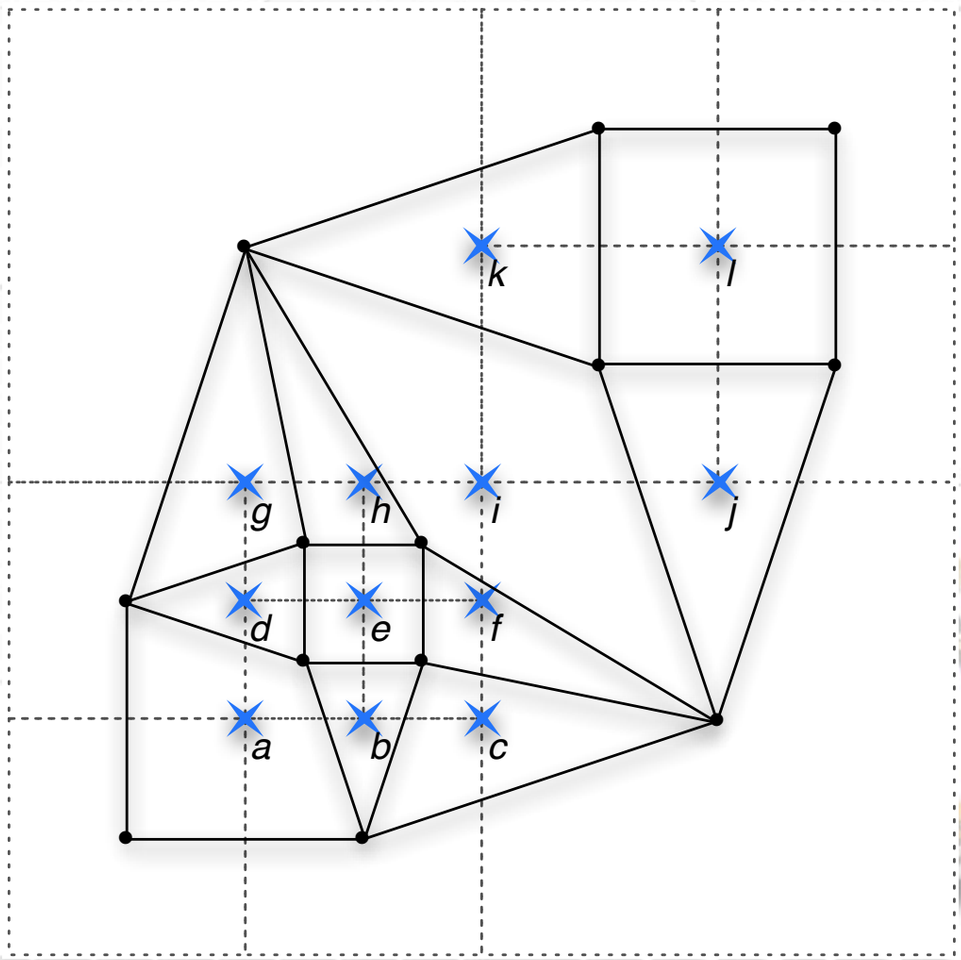
No duplicate vertex process

- static
- dynamic

Pointerless leaf vertex representation



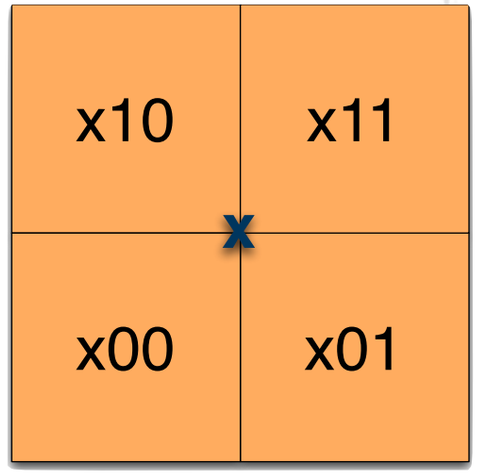
a: 10011000̄
b: 10011010̄
c: 10110000̄
d: 10011100̄
e: 10011110̄
f: 10110100̄
g: 11001000̄
h: 11001010̄
i: 11100000̄
j: 11101000̄
k: 11110000̄
l: 11111000̄



Geometric Morton code from vertex position

- 0 padding: hash function issue

Fast code conversion



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** $\{ [(dc \,\&\, dil_x) - (i \,\&\, dil_x)] \,\&\, dil_x \} \mid$
  $\{ [(dc \,\&\, dil_y) - (i \,\&\, dil_y)] \,\&\, dil_y \} \mid$
  $\{ [(dc \,\&\, dil_z) - (i \,\&\, dil_z)] \,\&\, dil_z \}$;

Dilated integer sum and subtraction:

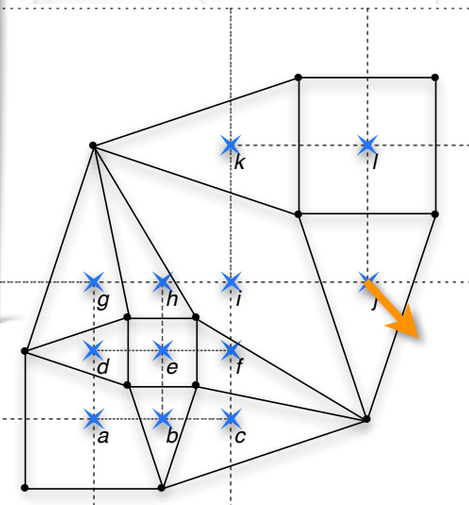
Code of leaf vertex x10 = Morton code of leaf ⊕ 10 *(with overflow test)*

Morton code of leaf = Code of vertex ⊖ 10

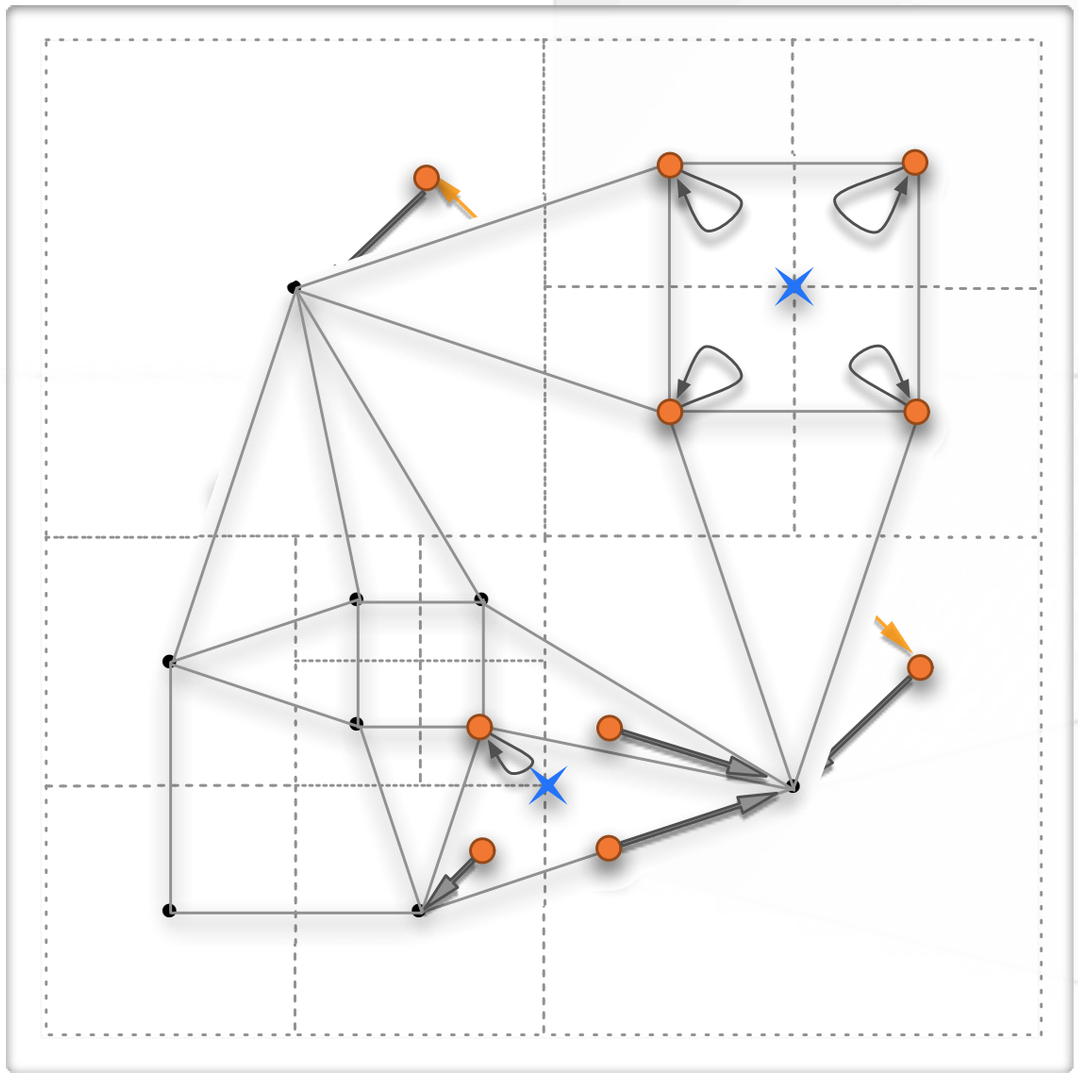
Ex: leave ⊖10 around *j*, *level 2*

j = 1.11.01.00.0-
j$_2$ = 1.11.01 *(level 2)*
j$_2$ ⊖10 = 1.11.01⊖10
= [(1.11.01 & 0101)-(10 & 0101)] & 1010 ||
[(1.11.01 & 1010)-(10 & 1010)] & 0100
= [1.01.01 - 00] & 1010 ||
[1.10.00 - 10] & 0100
= [1.01.01& 1010] || [1.01.10 & 0100]
= [1.01.01] || [0.00.10] = 1.01.11

j: 11101000̄



Optimized neighbor search



```
keys[8] ← vert2leaf(v,lv) ;
for j ∈ [0···7] do
    // optimized search for leaf: up from level lv
    while keys[j] ≠ ∅ & ¬node_exists(keys[j]) do
        keys[j] ≫= 3;
output keys ;
```

with the deepest adjacent leaf

⇒ look up only (less hash accesses)

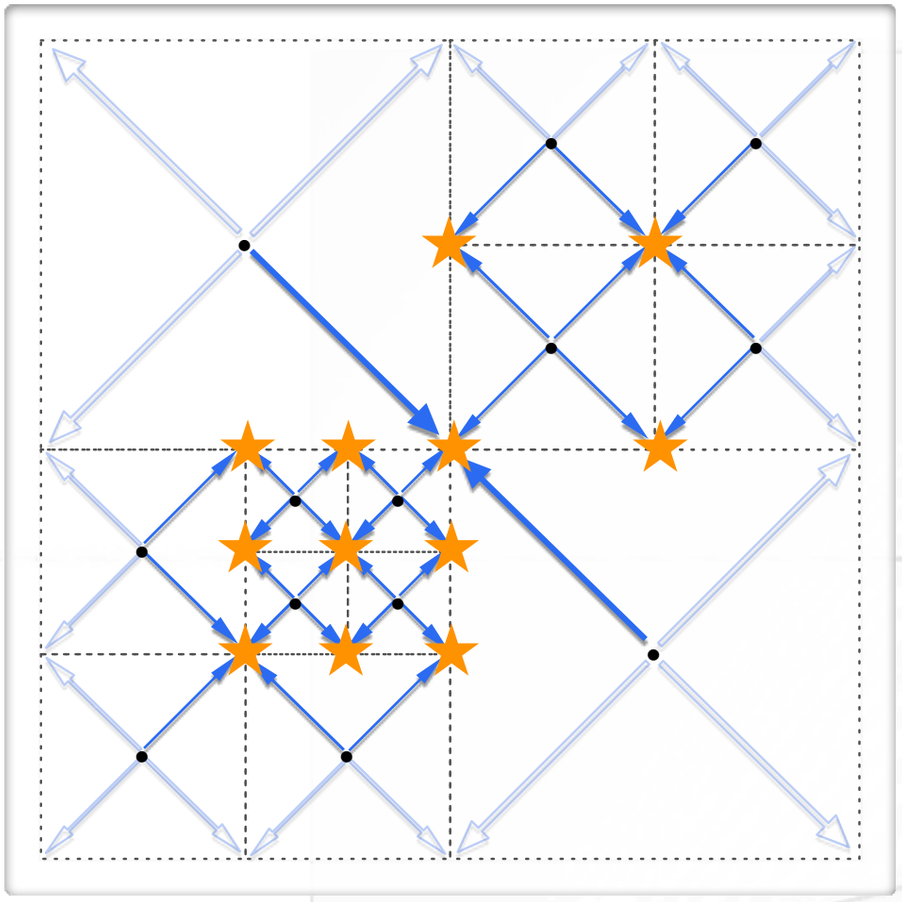
output a combinatorial cube

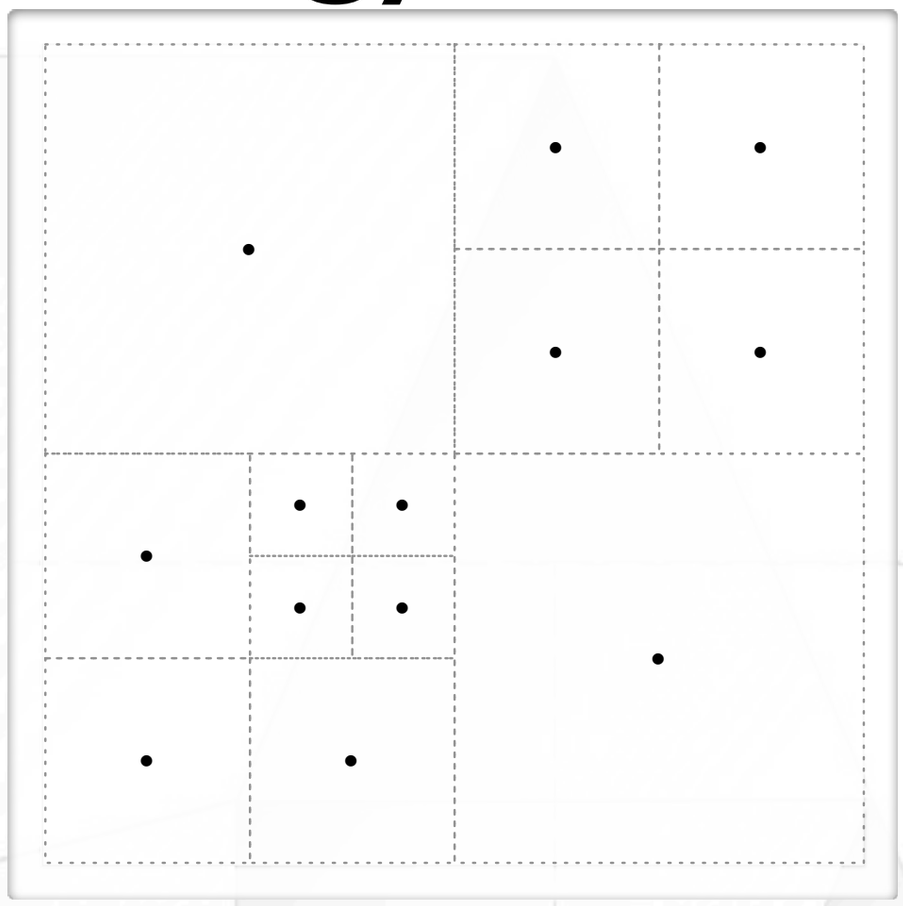
⇒ dual marching cubes

Process each vertex once

static / dynamic strategies

Static strategy





Algorithm 1: Preprocessing step: vertex generation

in : The octree
out: Auxiliary hash table *aux* with the vertices

foreach *key k of the octree's leaves* **do**
  // codes for k's vertices, see Algorithm $leaf2vert_{opt}$
  $level, v\_codes[8] \leftarrow \text{leaf2vert}(k)$;
  for $i \in [0 \cdots 7]$ **do**
    if $v\_codes[i] = \emptyset$ **then next** vert i ;
    // get current vertex/level data of aux
    $v, lv \leftarrow \text{aux}[\ v\_codes[i]\ ]$;
    if $v = \emptyset$ **then** $\text{aux}[\ v\_codes[i]\ ] \leftarrow level$;
    else $lv \leftarrow \min(level, lv)$;

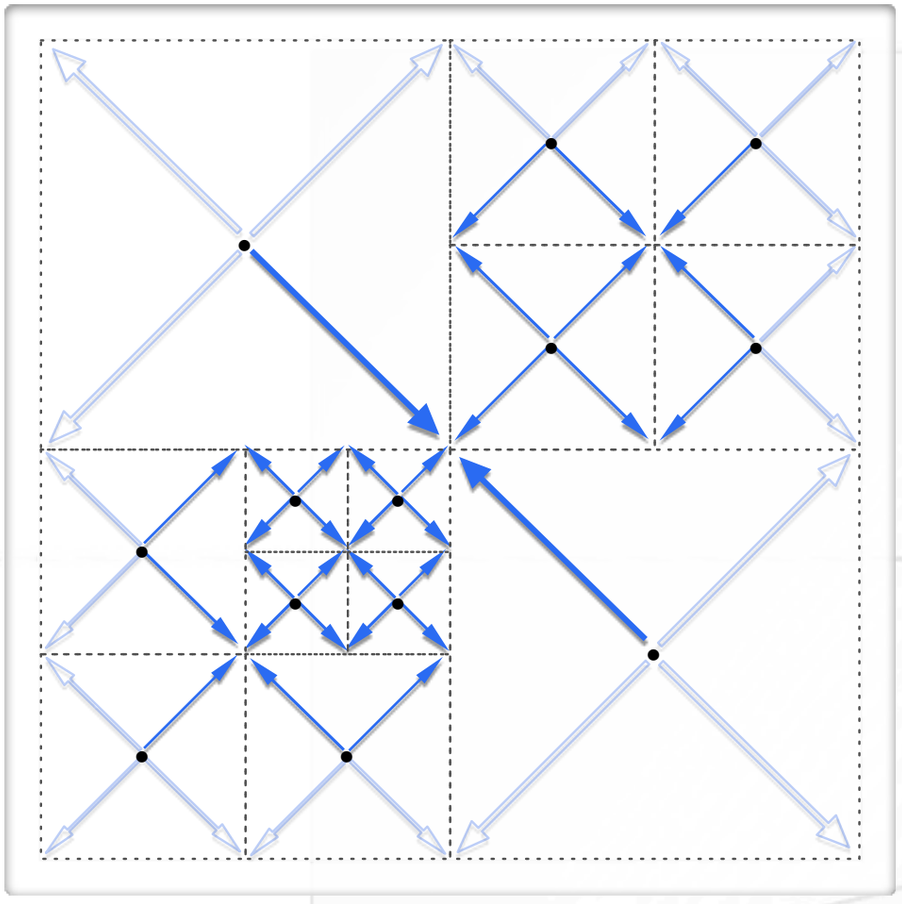
hash

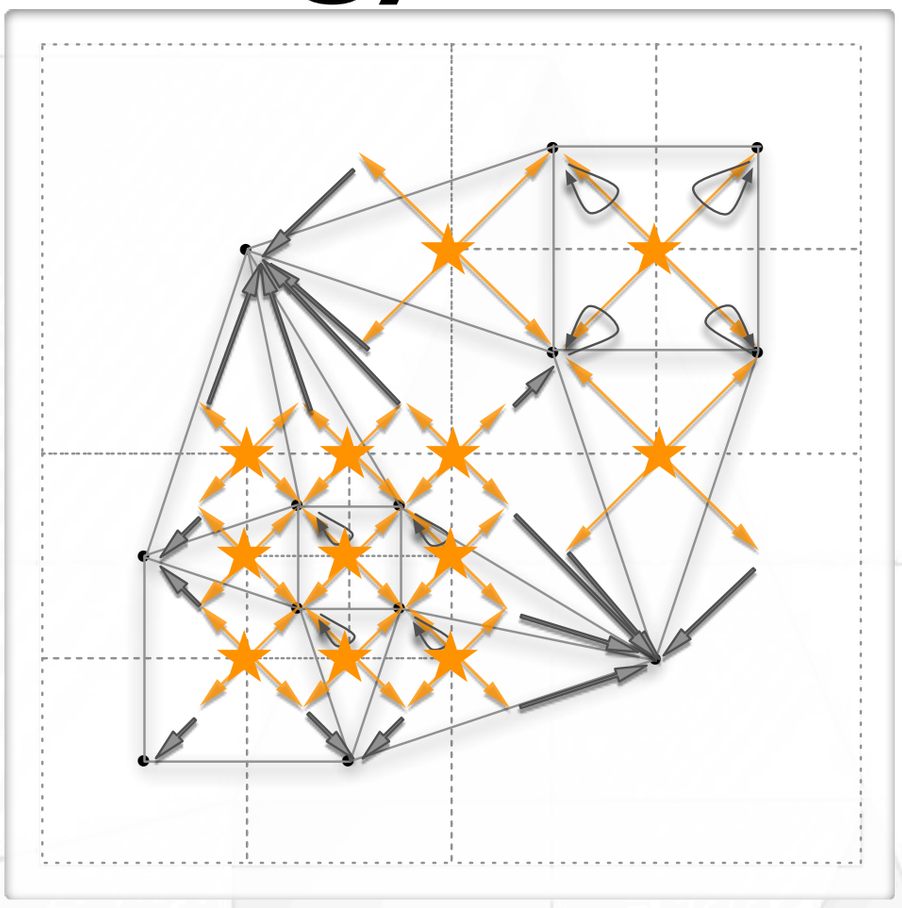
Morton codes for vertices

⇒ direct hashing

⇒ store deepest level

Static strategy





Algorithm 1: Preprocessing step: vertex generation

in : The octree
out: Auxiliary hash table *aux* with the vertices

foreach *key k of the octree's leaves* **do**
  // codes for k's vertices, see Algorithm $leaf2vert_{opt}$
  $level, v\_codes[8] \leftarrow$ leaf2vert(k) ;
  for $i \in [0 \cdots 7]$ **do**
    if $v\_codes[i] = \emptyset$ **then next** vert i ;
    // get current vertex/level data of aux
    $v, lv \leftarrow$ aux[$v\_codes[i]$] ;
    if $v = \emptyset$ **then** aux[$v\_codes[i]$] $\leftarrow level$;
    else $lv \leftarrow \min(level, lv)$;

hash

Algorithm 2: Traversal step: dual generation

in : The octree and the auxiliary hash table
out: The dual volumes

foreach *vertex code / level v, lv in aux* **do**
  // node keys at level lv, see Algorithm $vert2leaf_{opt}$
  $keys[8] \leftarrow$ vert2leaf(v,lv) ;
  for $j \in [0 \cdots 7]$ **do**
    // optimized search for leaf: up from level lv
    while $keys[j] \neq \emptyset$ & $\neg node\_exists(keys[j])$ **do**
      $keys[j] \gg= 3$;
  output *keys* ;

Static strategy

Good points:

second and further generation: only second step (>2x faster)

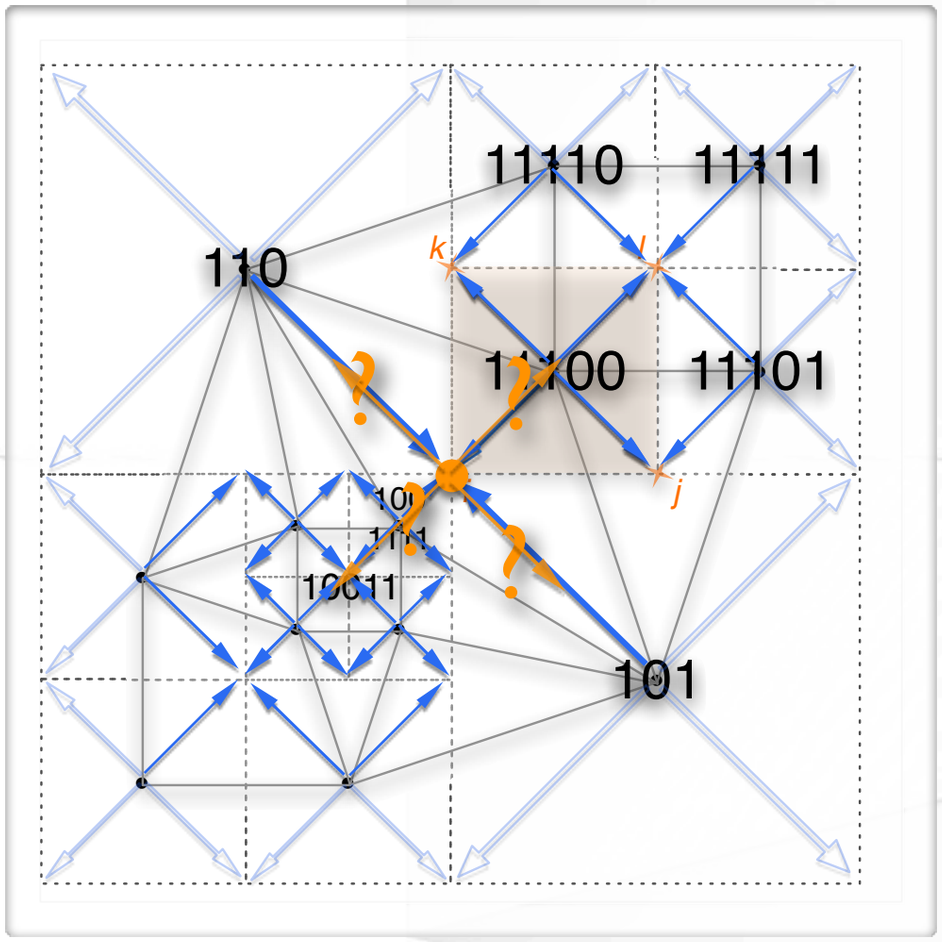
works with octree represented only by their leaves

Limitations:

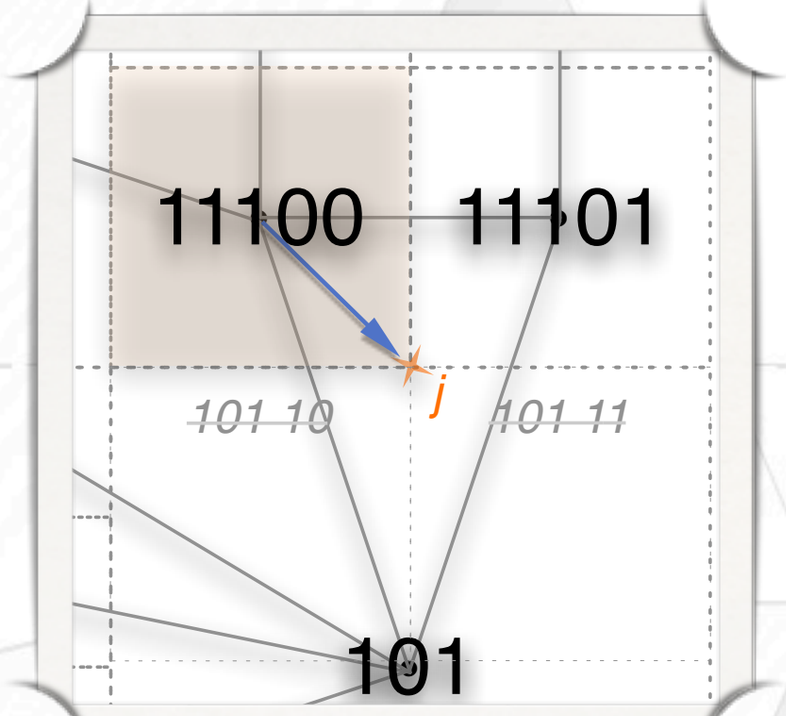
additional hash table

second hash function to optimize

Dynamic strategy



ex: leaf l =11100



l responsible for j?

One-to-one vertex to leaf mapping

Leaf vertex → deepest adjacent leaf
Tie breaking: first leaf in Morton order

Dynamic strategy

Good points:

- single pass
- no extra memory

Limitations:

- few extra hash accesses

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] ← leaf2vert(k) ;
3      for i ∈ [0···7] do
4          if v_codes[i] = ∅ 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] ← vert2leaf(v_codes[i],lv) ;
6          for j ∈ [0···7] \ {i} do
               // leaf k is deeper than neighborkeys[j]: OK,
                  check next neighbor key
7              if ¬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 ¬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 ∈ [0···7] do
               // optimized search for leaf: up from level lv
11             while keys[j] ≠ 0 & ¬node_exists(keys[j])
               do  keys[j] ≫= 3;
12         output keys ;
```

Results

Results: random octrees

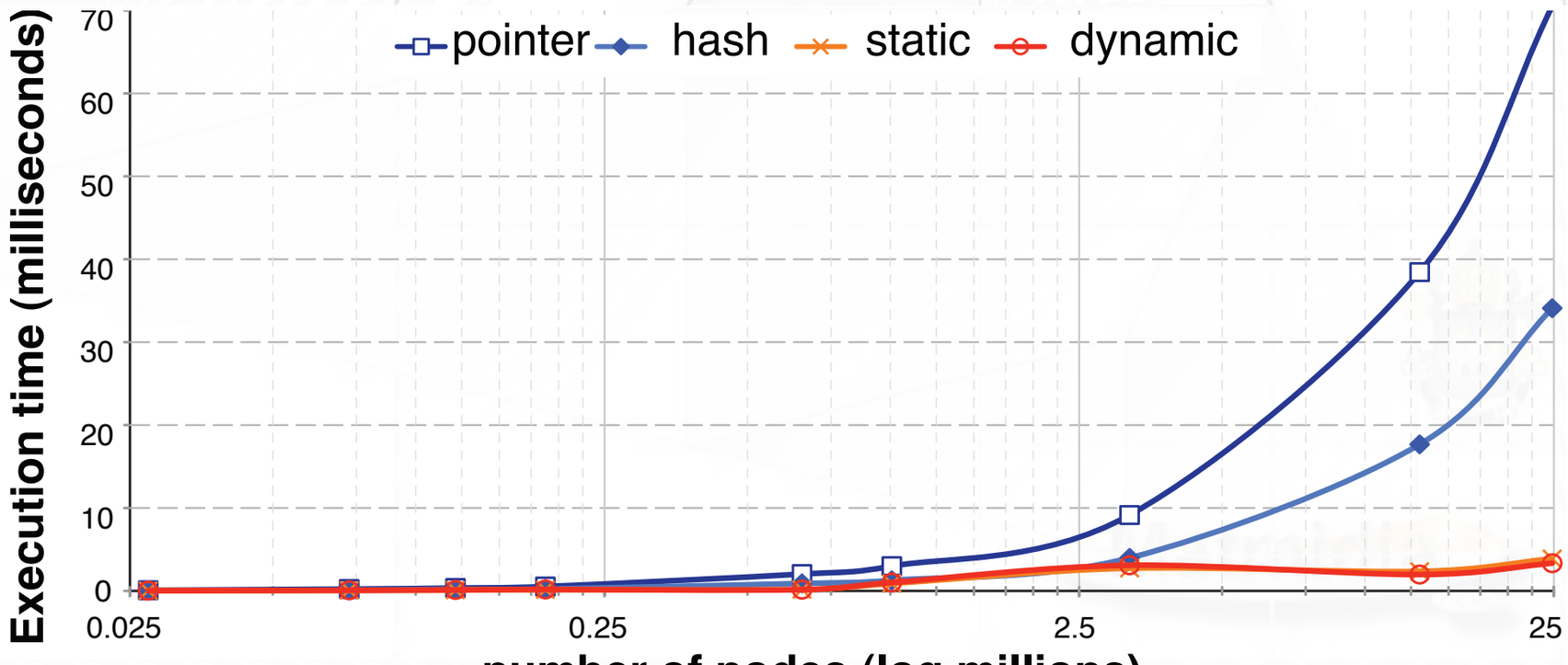
number of nodes (millions)		0.03	0.07	0.12	0.19	0.65	1.01	3.19	13.04	24.81
number of vertices (millions)		0.05	0.14	0.22	0.33	0.22	1.87	5.69	2.23	3.18
octree maximal level M		8	10	8	8	12	10	10	12	12
subdivision probability p		30%	30%	40%	45%	30%	40%	45%	40%	45%
bits for node hashing		21	21	21	21	21	24	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

Execution time

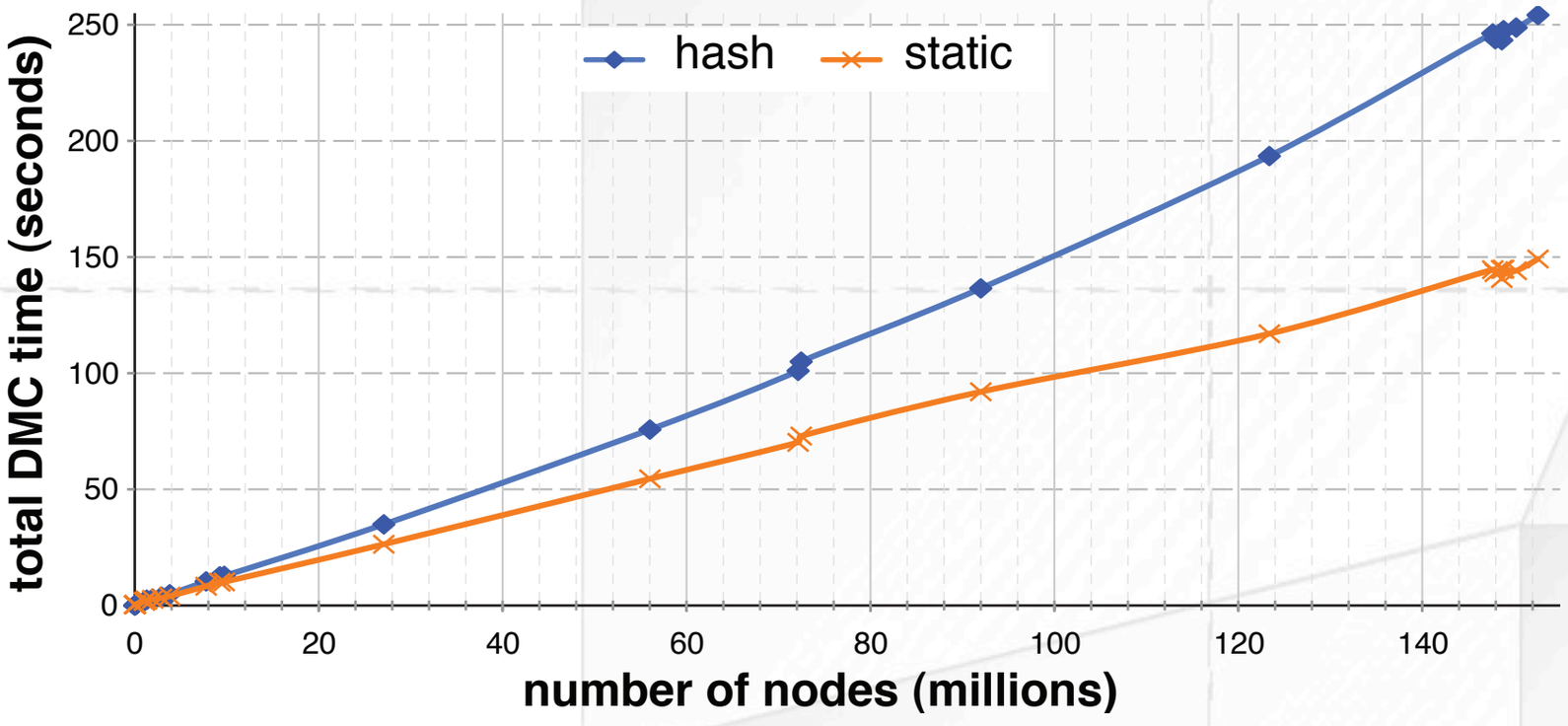
- 7x over pointer-based recursive
- 3x over pointerless recursive
- 6x for the static strategy, 2nd run
- faster on larger model

Memory

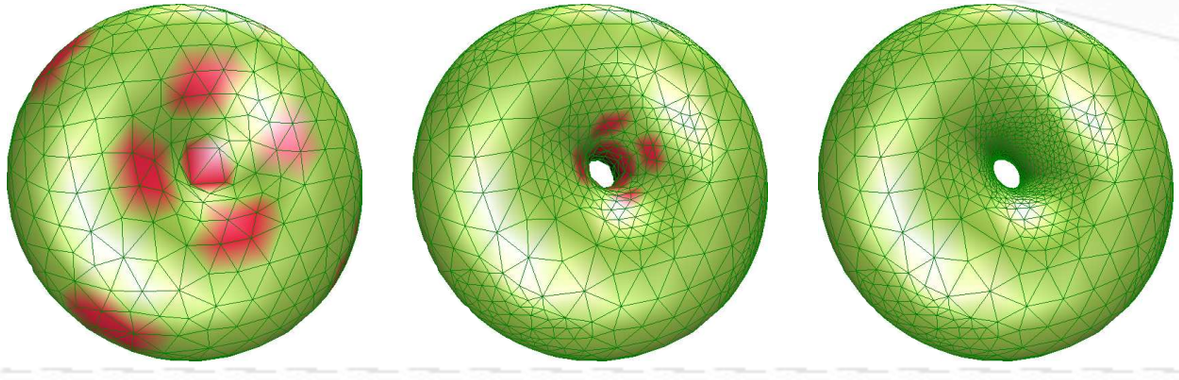
- 1.5x less for static over pointer
- 3x less for dynamic over pointer

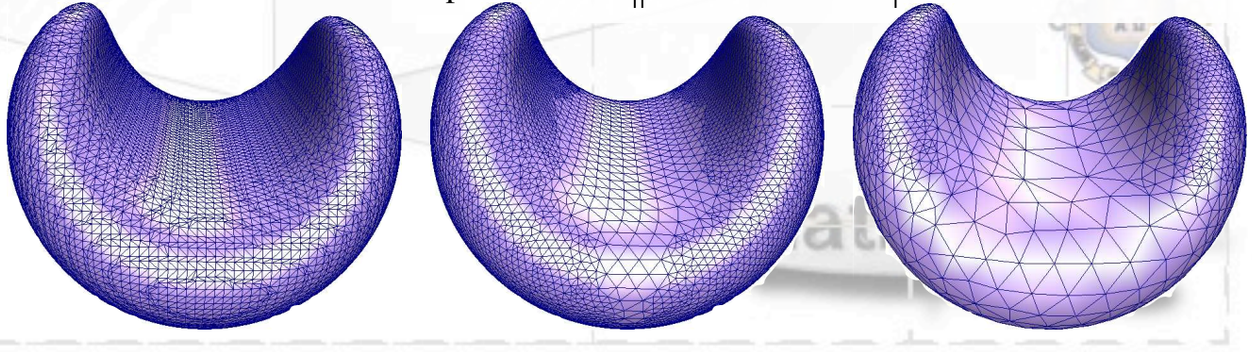


Application: Robust Dual Marching Cubes

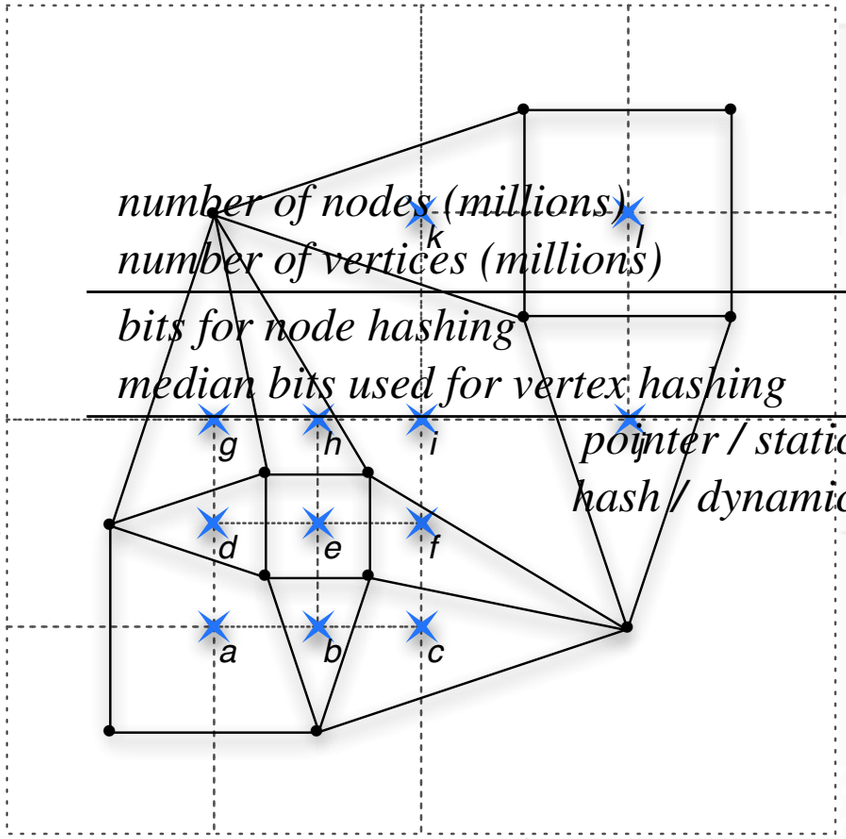


Implicit function	nodes $x10^6$	verts $x10^6$	hash *sec*	dyn *sec*	gain %
Torus	0.00	1	0.1	0.3	-67
Blob	0.03	4	0.1	0.4	-65
Cross cap	0.05	10	0.2	0.4	-53
Spheres in	1.0	85	1.8	1.8	2
Cylinders	1.3	159	2.5	2.3	6
2 Spheres	2.0	105	2.9	2.6	11
Glob tear	2.9	24	3.8	3.2	17
Weird cube	3.8	3	4.8	4.0	21
Lemniscate	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





Limitations



number of nodes (millions)	0.03	0.07	0.12	0.19	0.65	1.01	3.10	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
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
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*

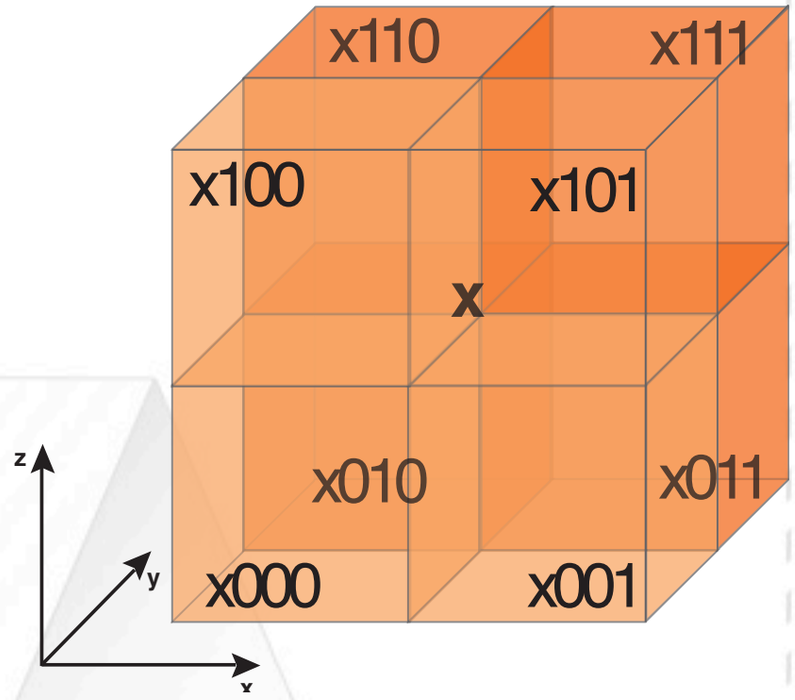
a: 10011000
b: 10011010
c: 10110000
d: 10011100
e: 10011110
f: 10110100
g: 10010000
h: 11001010
i: 11100000
j: 11101000
k: 11110000
l: 11111000

Hashing

Hash function for vertex codes

Hash efficiency crucial: large hash tables

Conclusions



Dual generation optimized with pointerless octree

Also linear, but processing 1/9th of the cells

Directly extendable to n-dimension

In average 3x faster with less memory

Next step: perfect hashing optimized in GPU?

Thank you for your attention!

http://www.matmidia.org/tomlew
http://www.matmidia.org/research/2010/560/

open source code soon...

Thanks to: Brazilian Ministry of Science / CNPq Productivity Scholarship,
Rio de Janeiro's State / FAPERJ Young Scientist of our State Program,
Alagoas State / FAPEAL Summer School Funding
PUC-Rio / Research Grant