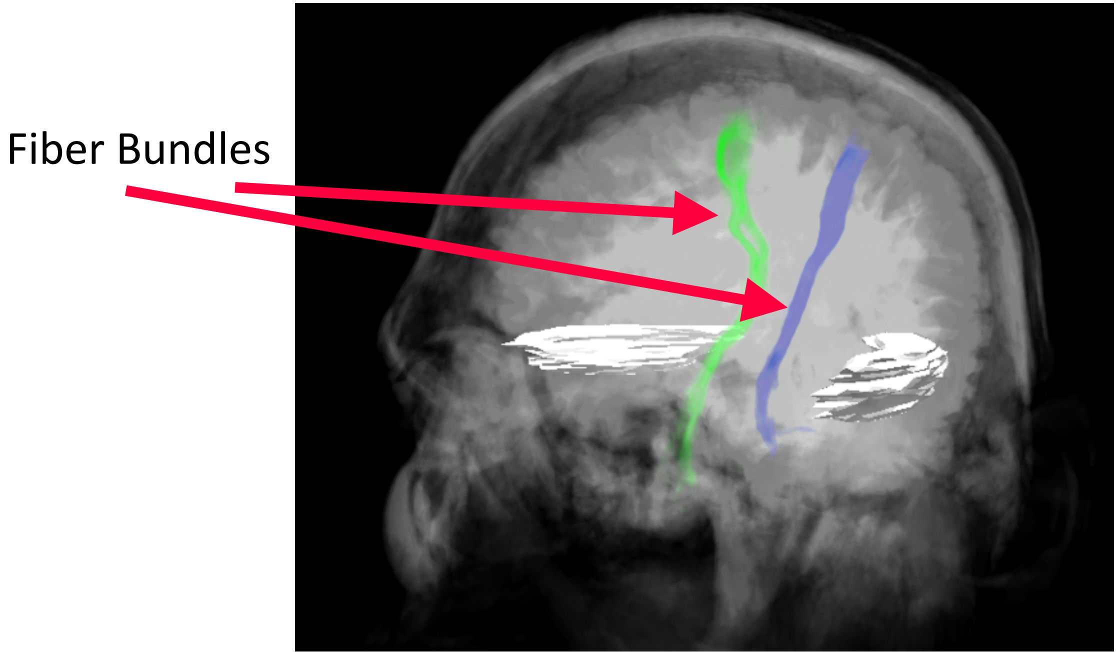
Rapid *k*-d tree construction for sparse volume data

Stefan Zellmann*, Jürgen Schulze**, Ulrich Lang*
* University of Cologne
** University of California San Diego

Sparse Volume Data

- Background: stereotactic operation planning
  - Insert "brain pacemaker" at designated position, Parkinson treatment.
  - Pacemaker: tiny probe, pushed using a stereotactic needle
- Datasets with multiple (~10-15) volume channels:
  - CT, T1/T2 MRI + Functional MRI
  - *Probabilistic* Fiber Tracking with FSL ([0..1] density volumes, each voxel denotes probability that fibers overlap)
    - Fiber bundles extremely sparse
  - Blood vessels as separate channel, also rather sparse
- Our goal (long term): real-time (VR-ready) visualization of multiple sparse channels

Sparse Volume Data



MR data set with two fiber bundles - each fiber bundle is a sparse volume channel

Objective: find path way for operation w/o penetrating vessels, liquor or fiber bundles.

Planning process guided by visualization

Visualization:

- Interactive (3D stereo)
- User can switch channels on/off
- Separate transfer function per volume channel

Sparse Volume Rendering

- Many channels, will likely not all fit into VRAM
  - even then, bandwidth is the limiting factor
  - ==> we simply need spatial indexing for sparse channels
- Mandatory: interactive transfer function editing
  - hard problem: rebuild spatial index in real-time
  - luckily, single channel moderately sized (256^3 to 512^3)

k-d Tree Construction for Sparse Volumes

- We base our work on previous work from Vidal et al.: *Simple empty-space removal for interactive volume rendering* (2008)
- First build a *summed volume table* (SVT) for the whole volume

0	0	0	1	2
0	0	1	0	0
1	0	3	2	2
0	0	1	1	0
0	0	0	1	0

0	0	0	1	3
0	0	1	2	4
1	1	5	8	12
1	1	6	10	14
1	1	6	11	15

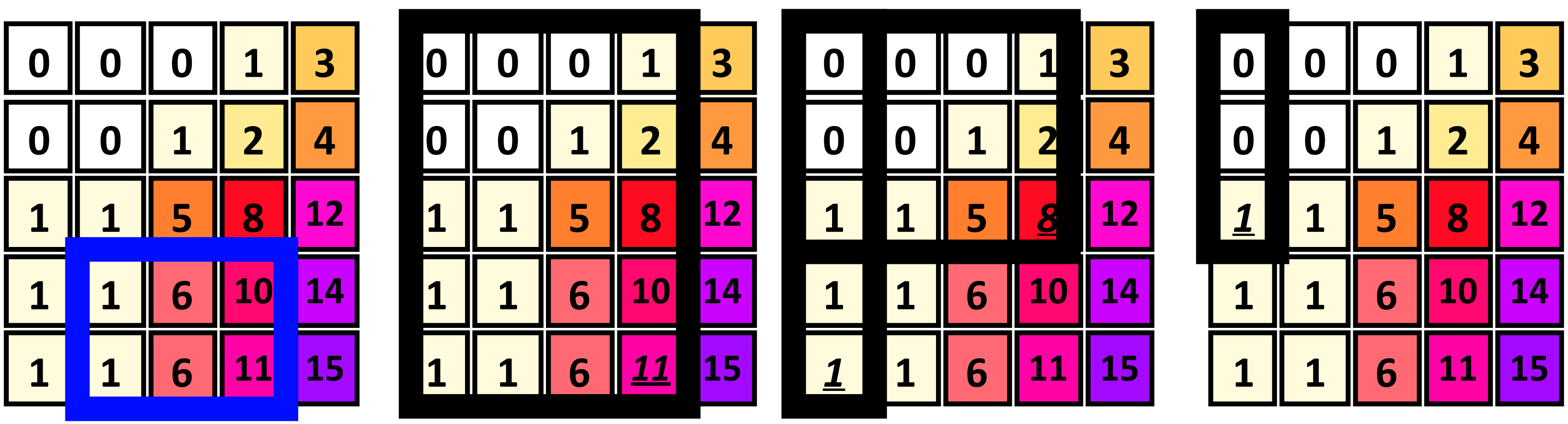
k-d Tree Construction for Sparse Volumes

- This is actually a (2D) summed *area* table (very similar in 3D)
- Constant time occupancy queries

0	0	0	1	3
0	0	1	2	4
1	1	5	8	12
1	1	6	10	14
1	1	6	11	15

k-d Tree Construction for Sparse Volumes

- This is actually a (2D) summed *area* table (very similar in 3D)
- Constant time occupancy queries



1.) Density in this box?	2.) Density in that bigger box	3.) Minus density in those two boxes	4.) But wait, we subtracted this here twice!
D=	**11**	**- (8+1)**	**+1 = 3**

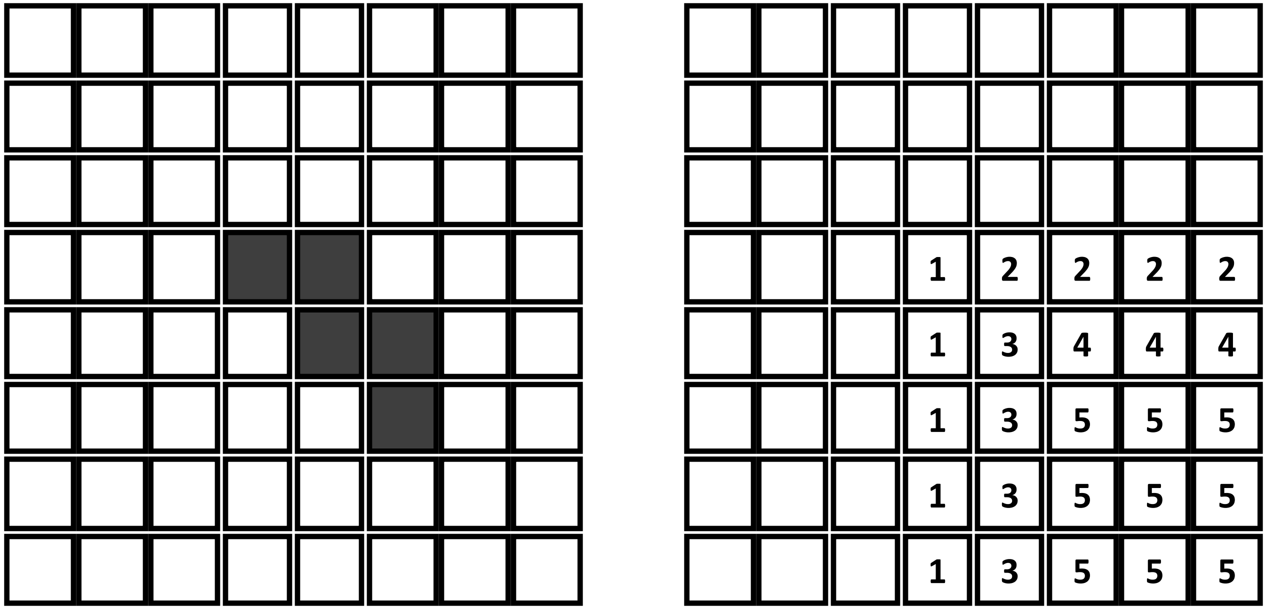
k-d Tree Construction for Sparse Volumes

- So that seems about right
- With SVTs it’s eight rather than four memory accesses

0	0	0	1	2
0	0	1	0	0
1	0	3	2	2
0	0	1	1	0
0	0	0	1	0

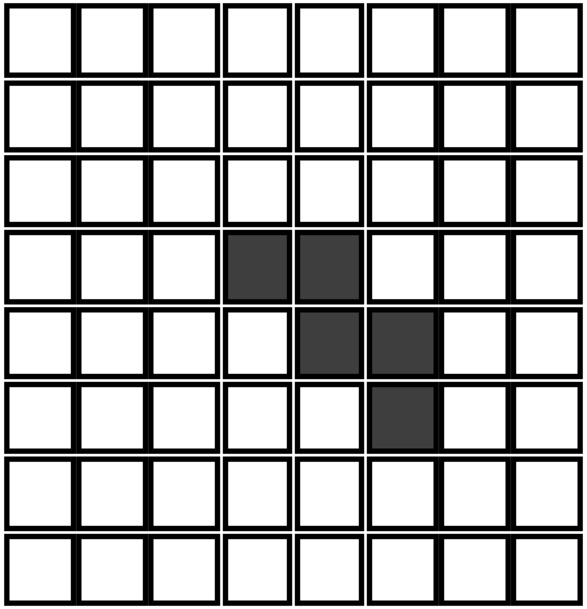
k-d Tree Construction for Sparse Volumes

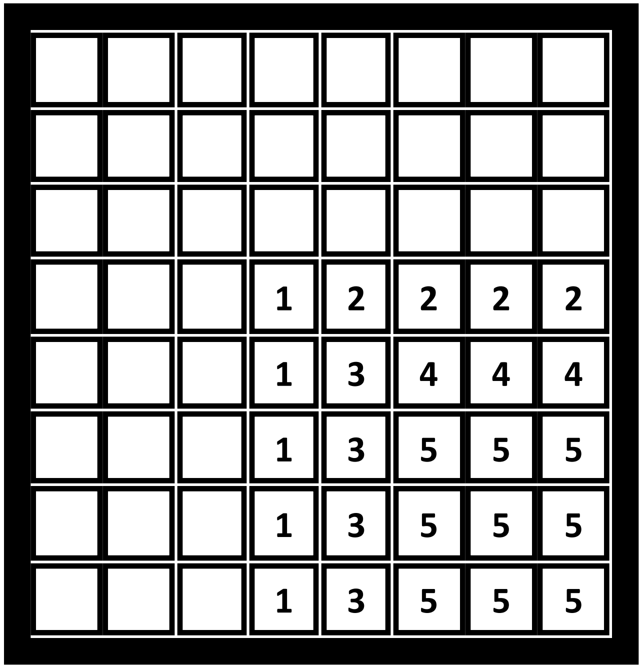
- With SVTs we can find tight bounding boxes around occupied regions
- Let's consider a different case: binary voxels, and sparse



k-d Tree Construction for Sparse Volumes

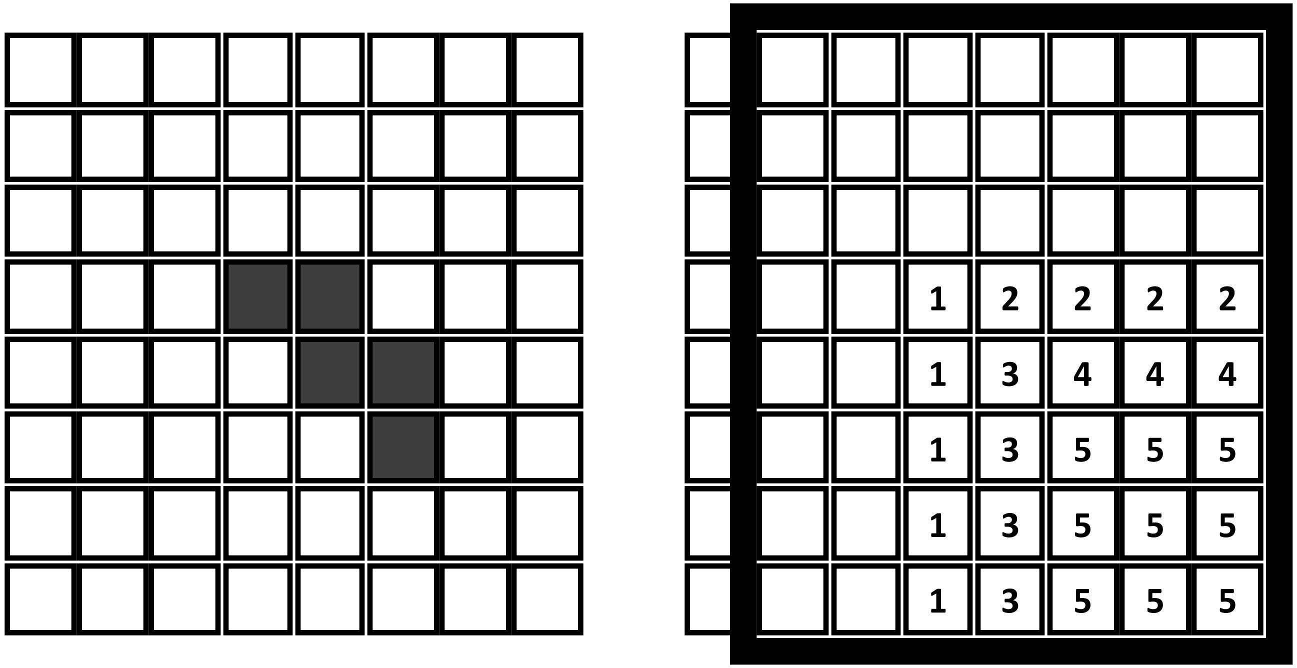
- We just find an initial bounding box and shrink it iteratively
- Density: **5**





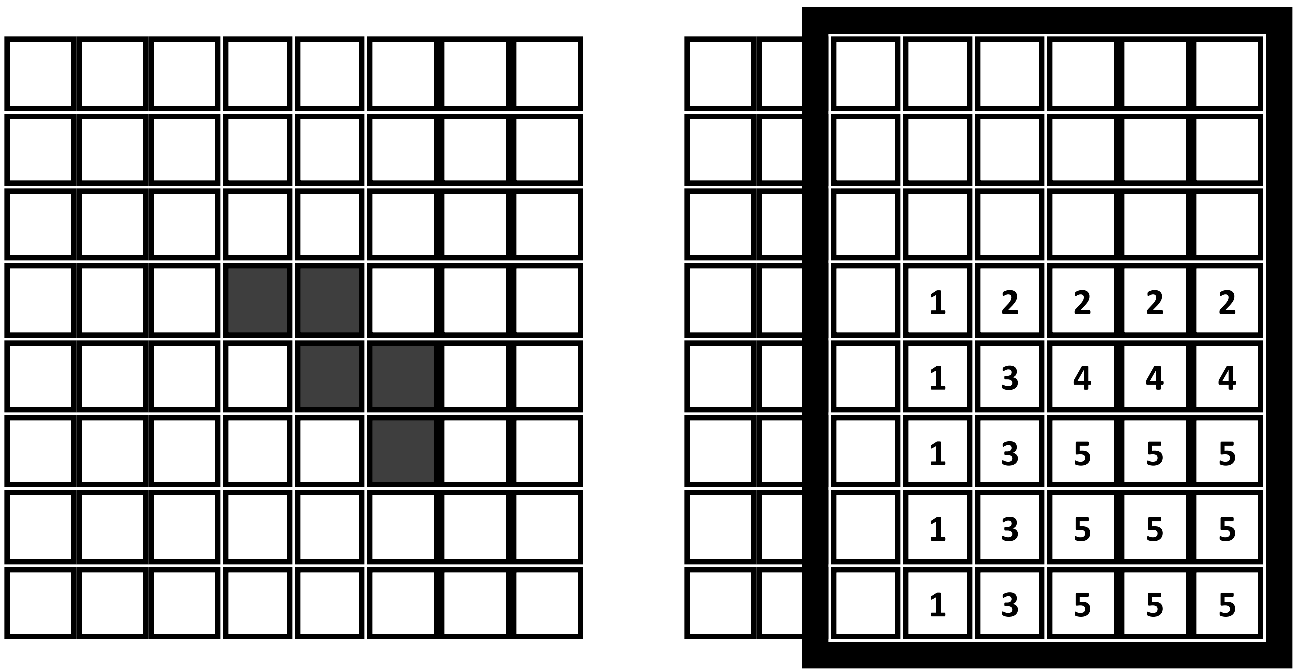
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- First in x-direction. Still **5**



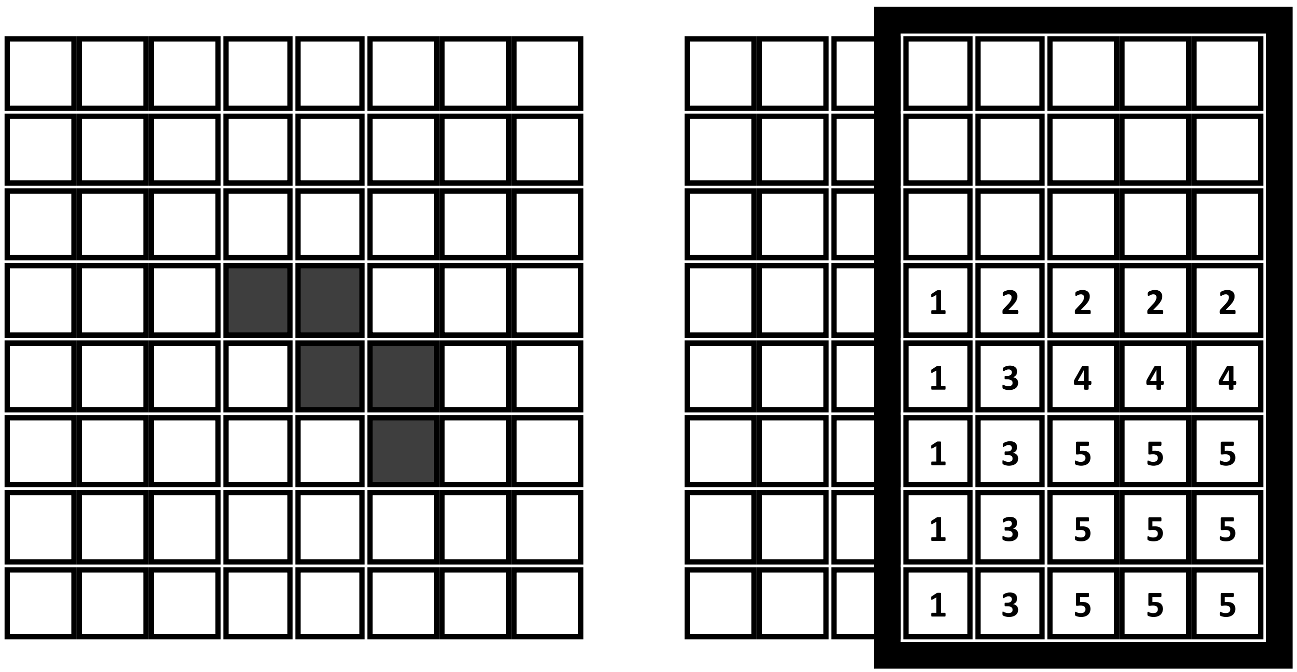
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- First in x-direction. And still.. **5**



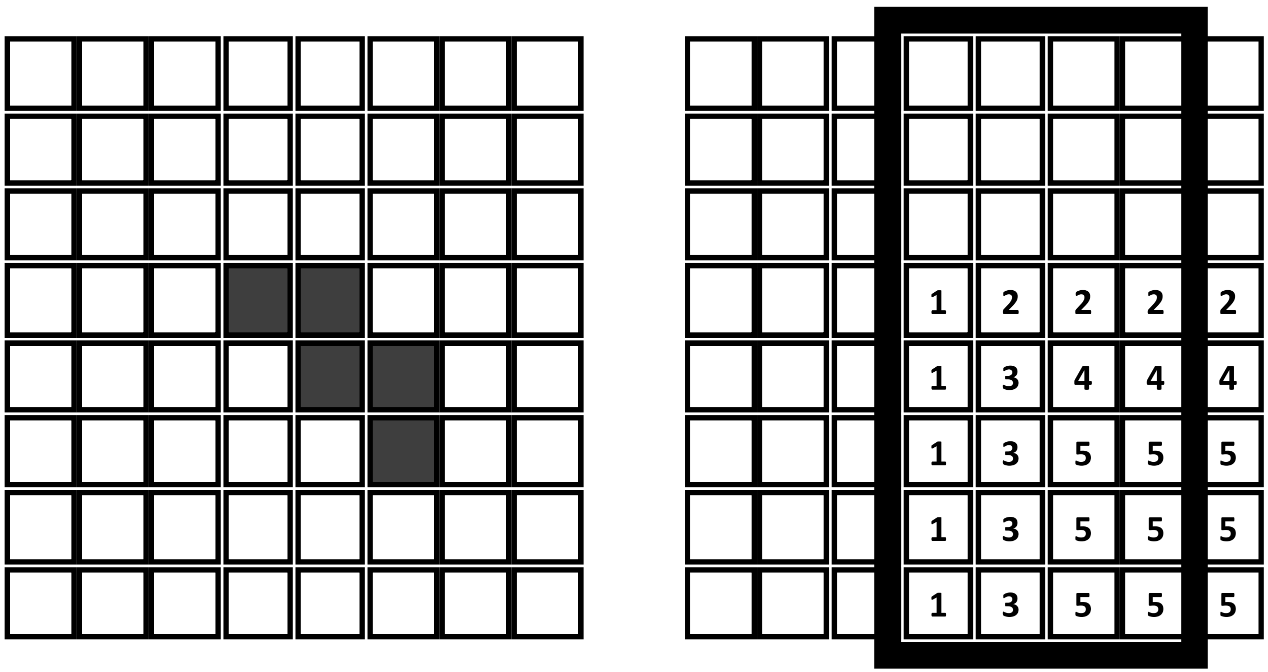
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- Ok, no step further, next we'd be **< 5**



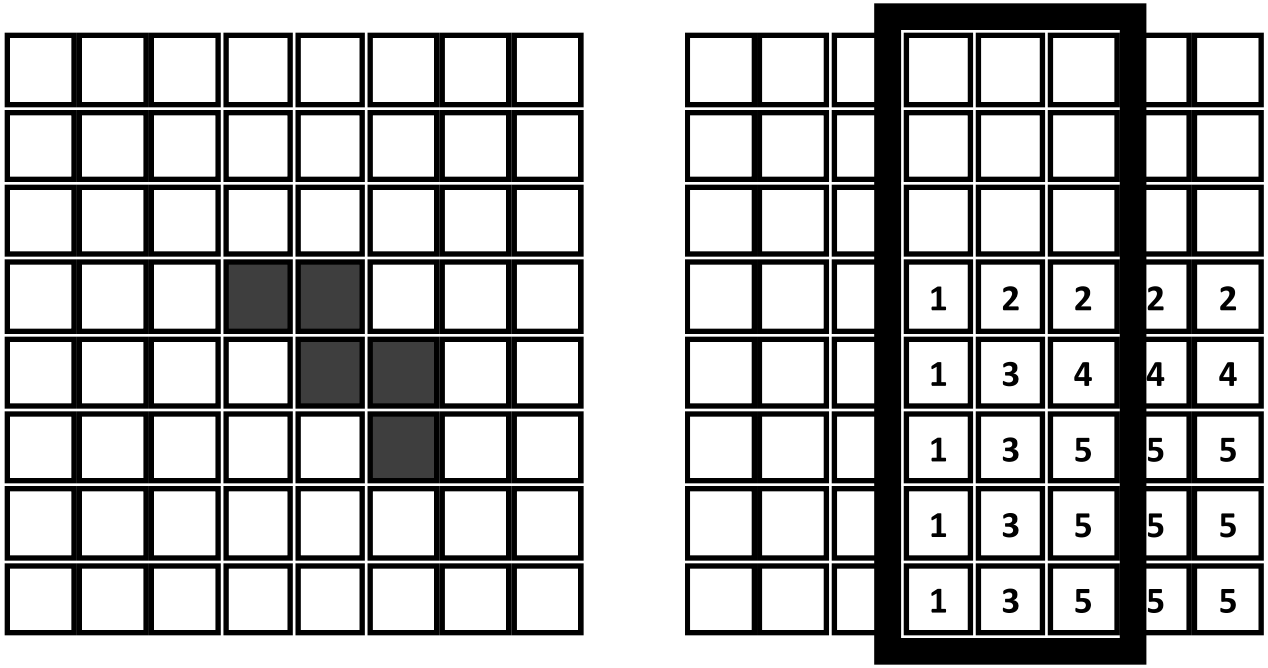
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- negative x. Density is **5**



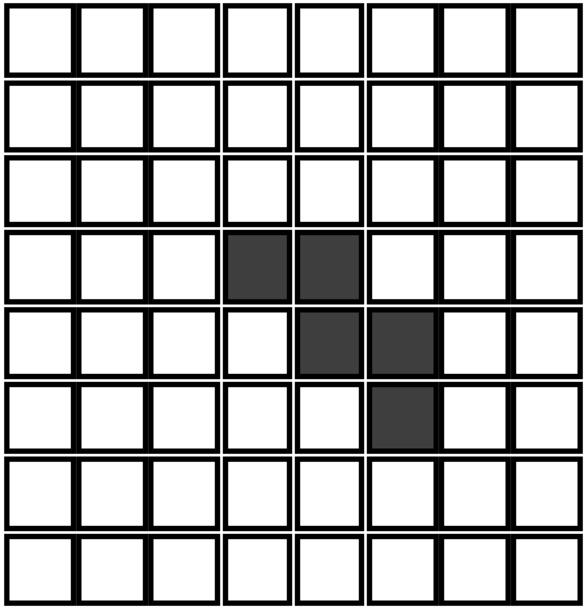
k-d Tree Construction for Sparse Volumes

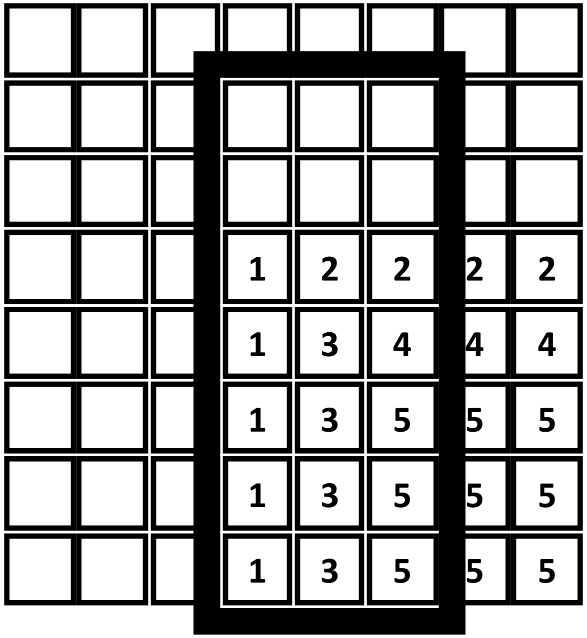
- We just find an initial bounding box and shrink it iteratively
- negative x - so here's a slope again, so full stop



k-d Tree Construction for Sparse Volumes

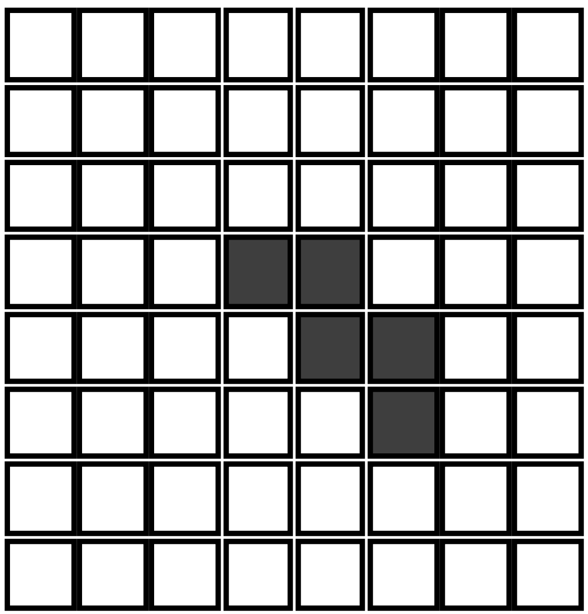
- We just find an initial bounding box and shrink it iteratively
- same thing with y and -y

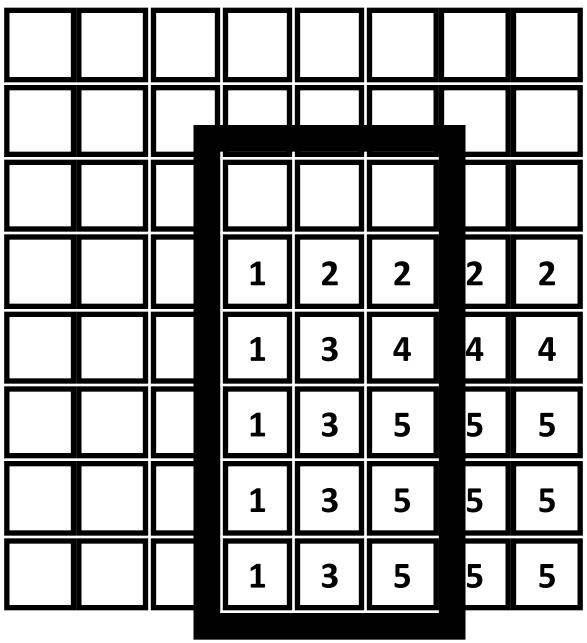




k-d Tree Construction for Sparse Volumes

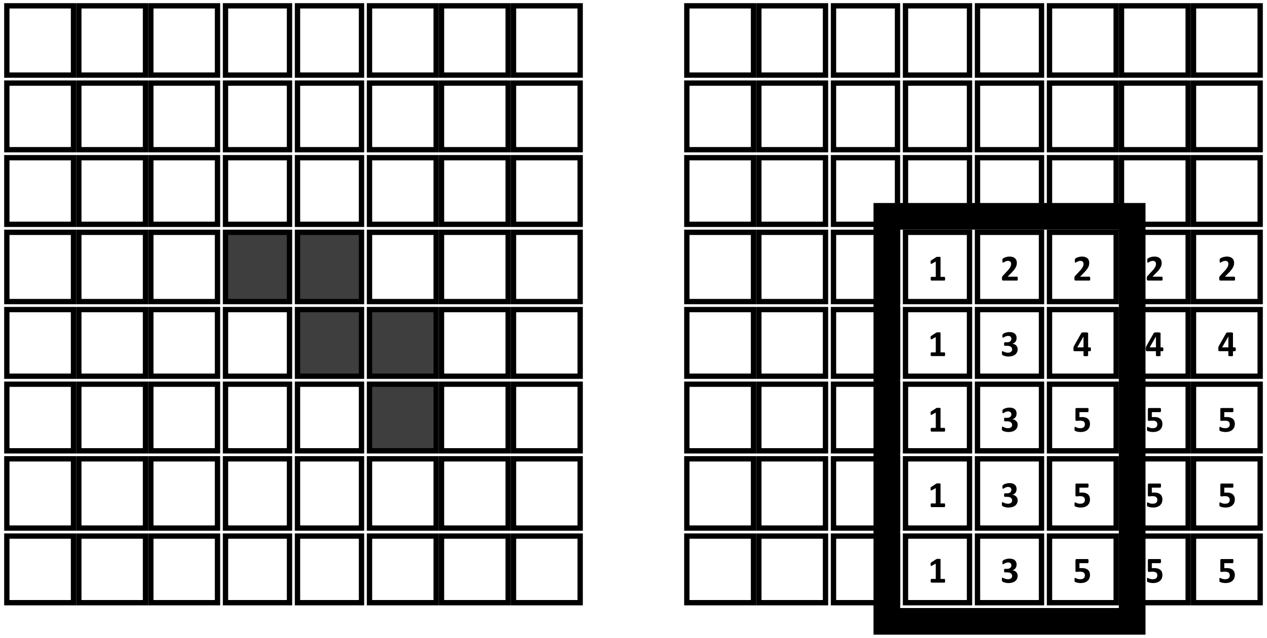
- We just find an initial bounding box and shrink it iteratively
- same thing with y and -y





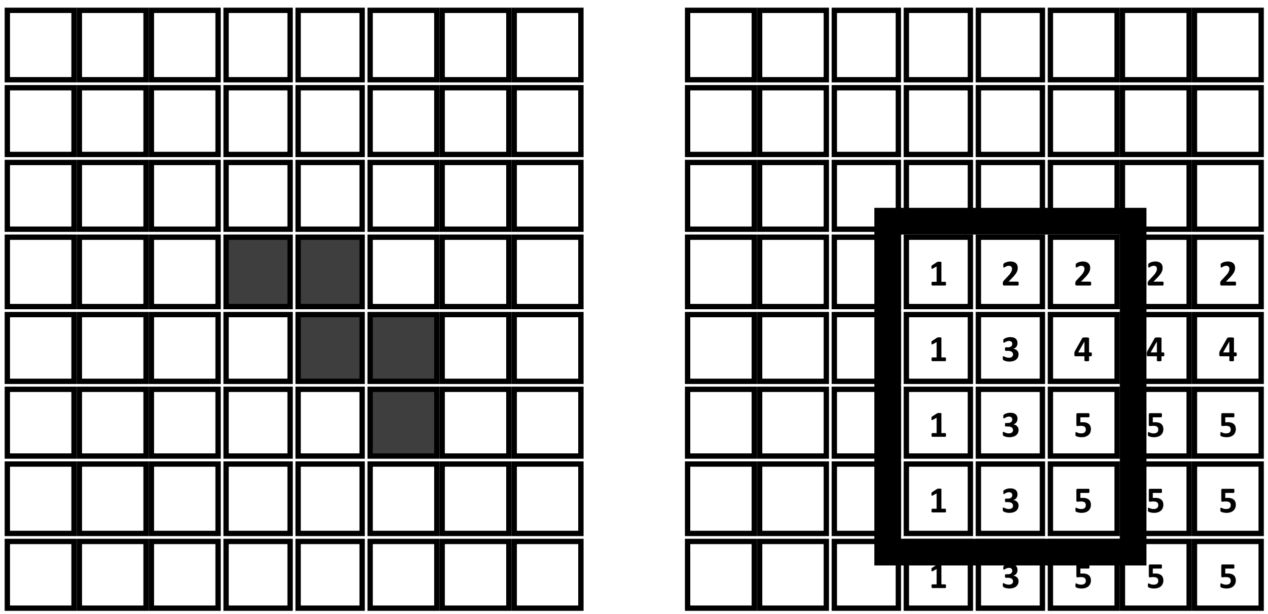
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- same thing with y and -y



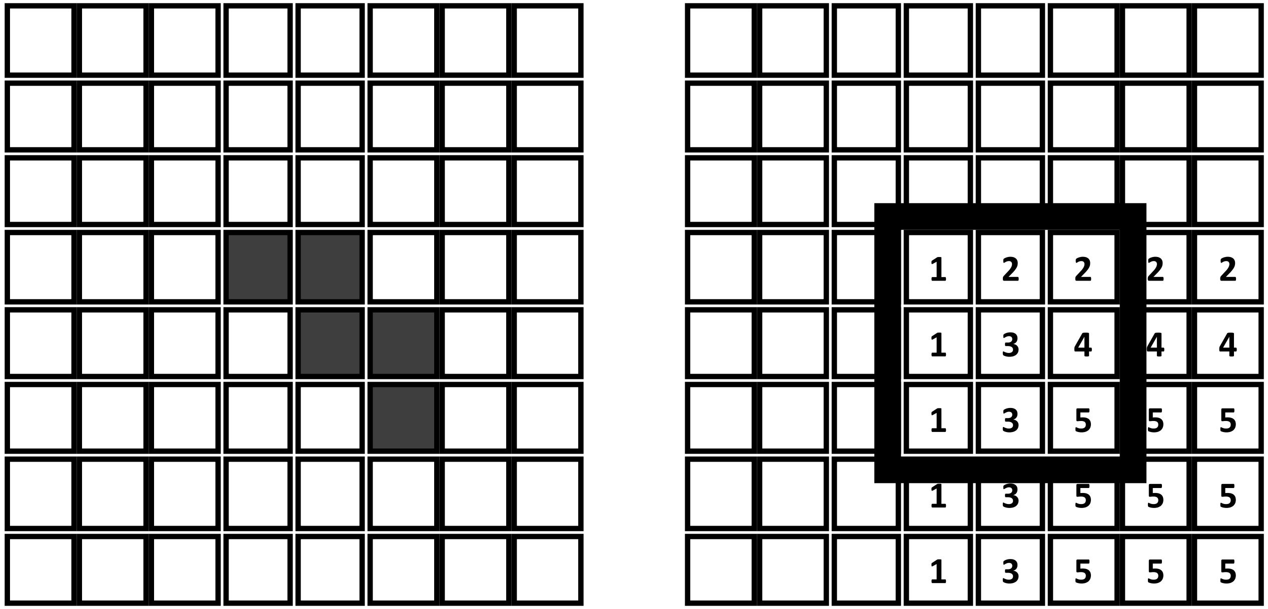
k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- same thing with y and -y



k-d Tree Construction for Sparse Volumes

- We just find an initial bounding box and shrink it iteratively
- Found an AABB, contains all the voxels, since density is **5**

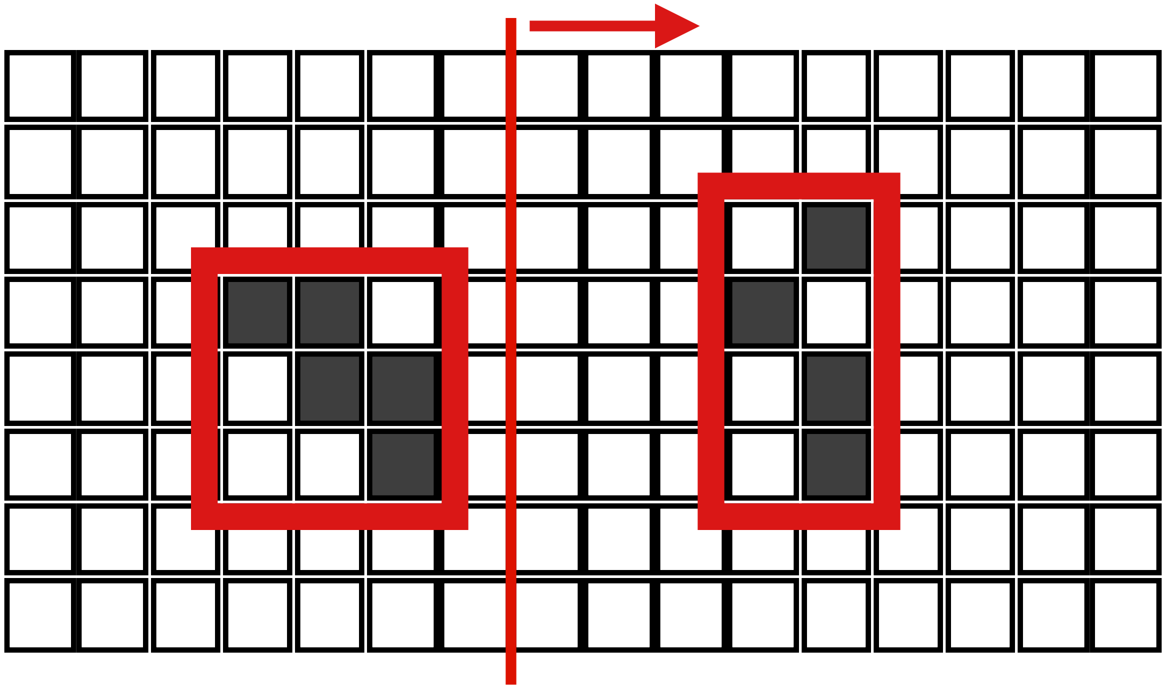


Two-Phase Algorithm

- Phase 1: ***(SVT Phase)*** construct SVT
- Phase 2: ***(Split Phase)*** use SVT to top-down construct *k*-d tree

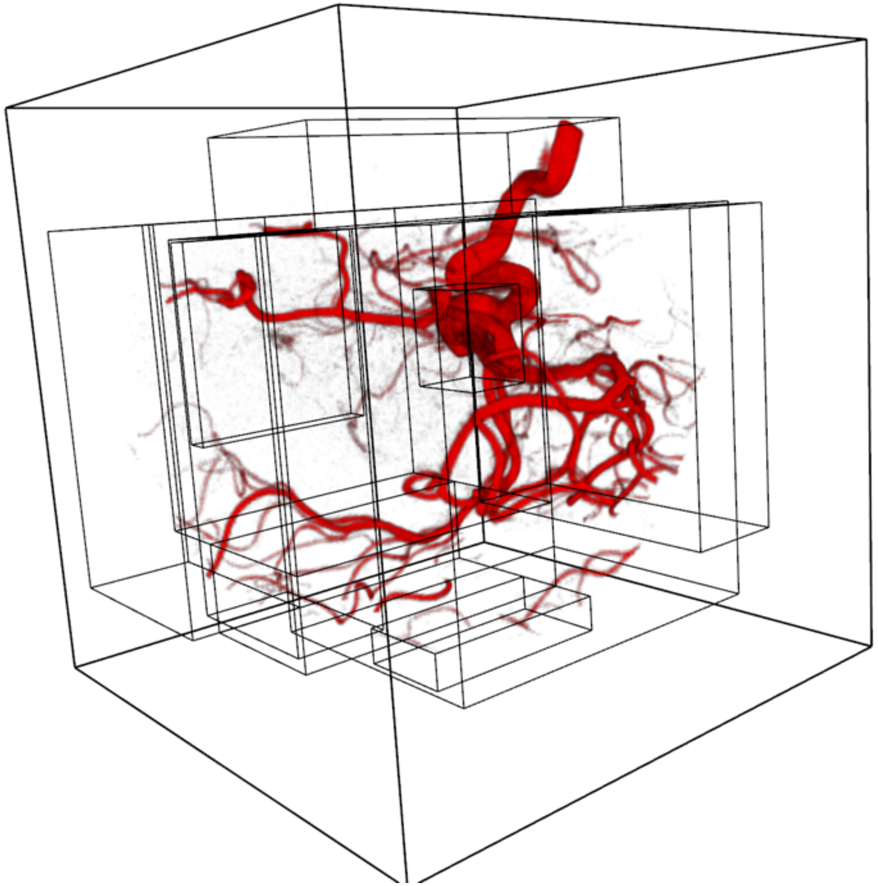
Greedy Top-Down Construction

- Similar to binned surface area heuristic builders for triangles
  - Candidate planes, minimize cost function based on box volumes: $C(p) = V(B_L(p)) + V(B_R(p))$
  - Binary Split until certain criteria like min. AABB volume etc. apply



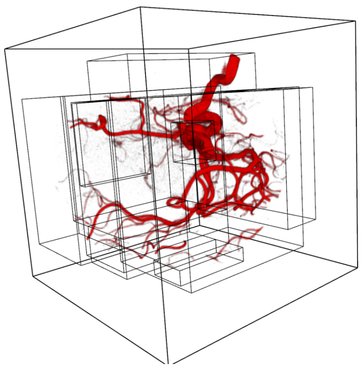
Greedy Top-Down Construction

- Result: *non-overlapping* boxes that we can sort back-to-front (*k*-d tree traversal)
- Volume rendering for each box
- Or an outer loop over boxes for the ray marcher

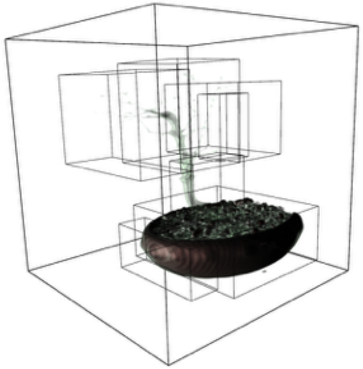


Problem with the Approach

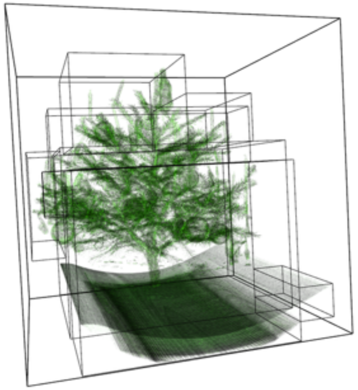
- Serial construction algorithm dominated by SVT construction time
    - SVT invalid after transfer function has changed



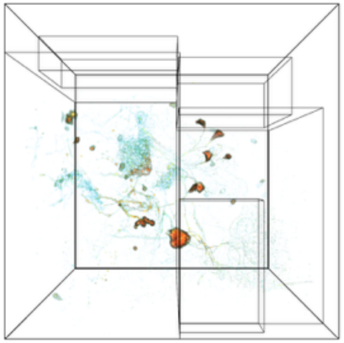
256^3 voxels
0.180 sec. SVT construction
0.002 sec. top-down build



256^3 voxels
0.180 sec. SVT construction
< 0.001 sec. top-down build



512^2 x 499 voxels
1.436 sec. SVT construction
0.007 sec. top-down build

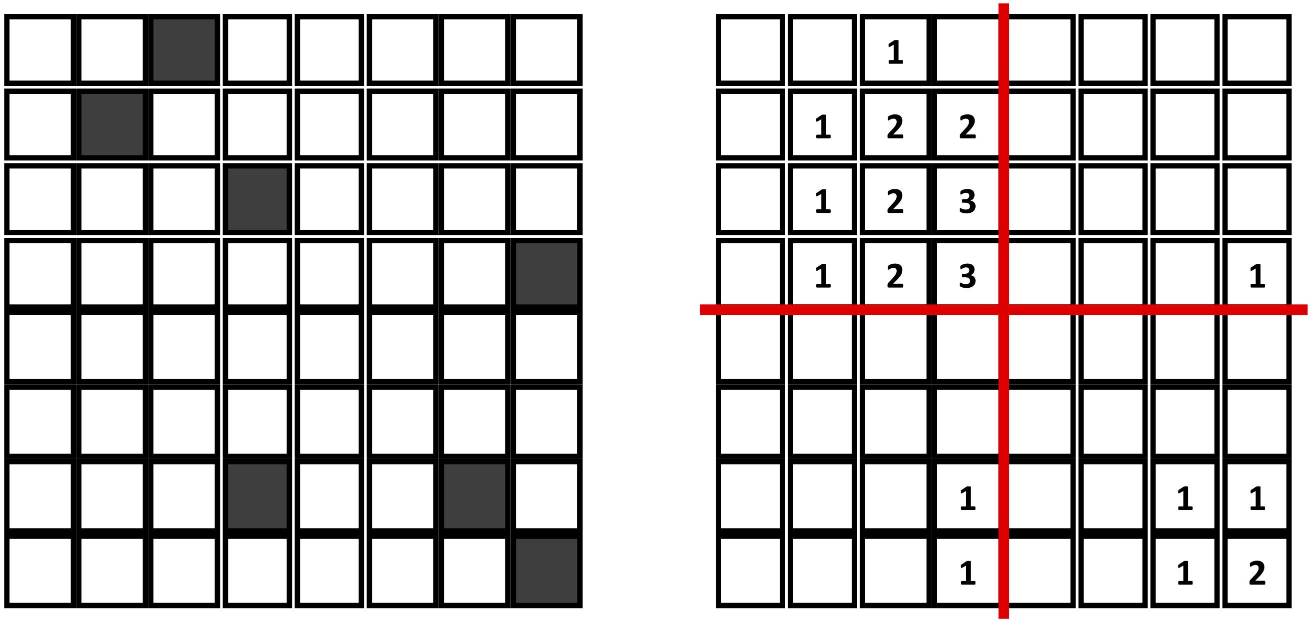


1000^2 x 910 voxels
9.361 sec. SVT construction
0.020 sec. top-down build

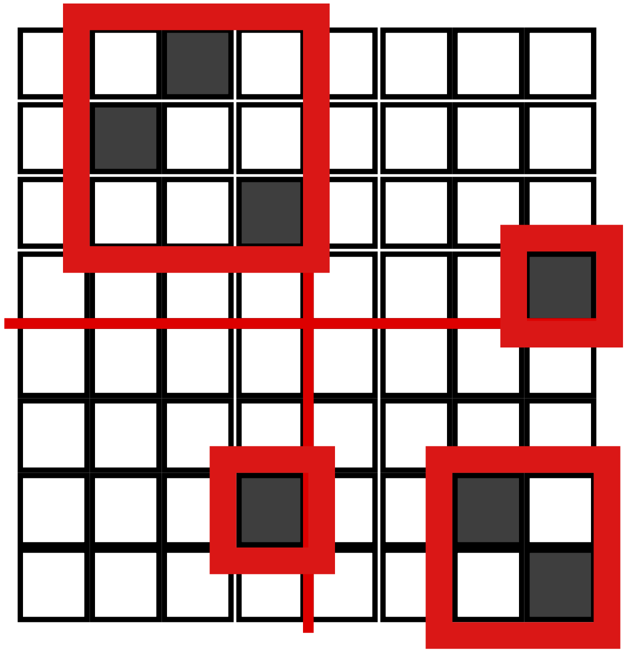
Parallel Construction Algorithm

- Multi-Core CPU: build only *partial SVTs* **(in parallel!)**
  - Volume bricks that fit into L1 memory (on our machine: 32^3 bricks)
- Whenever we want to find a tight AABB:
  - First find tight AABBs in L1 within bricks **(in parallel!)**
  - Then trivially combine the AABBs (serial min/max combine) to find the global tight AABB
- Enables parallelism with an otherwise rather serial algorithm
- Memory accesses fully cached
- Top-down construction slightly more time-consuming
  - Shifts construction time SVT construction to top-down builder

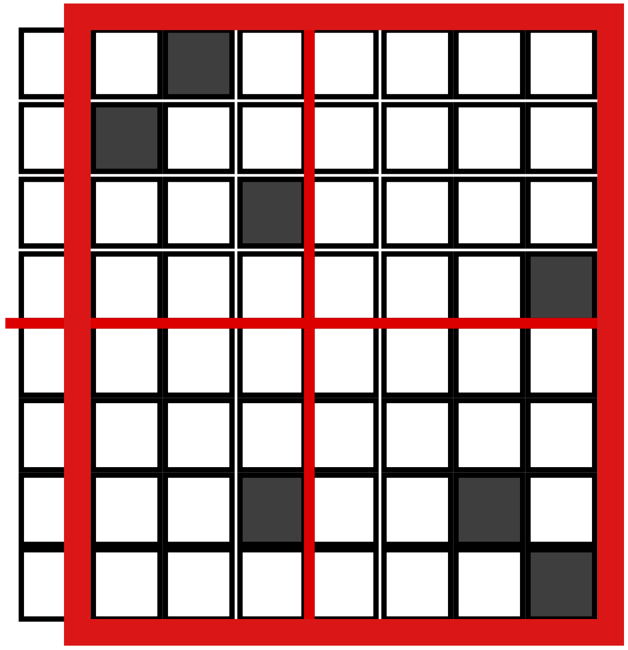
Find Bounds with Partial SVTs



Find Bounds with Partial SVTs



Find **local bounds** in parallel and in L1

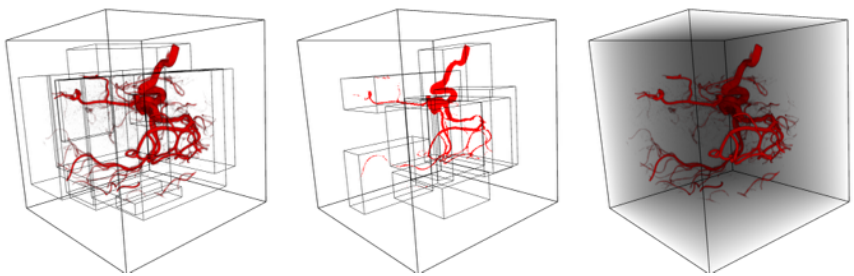


Find **global bounds** with trivial combine

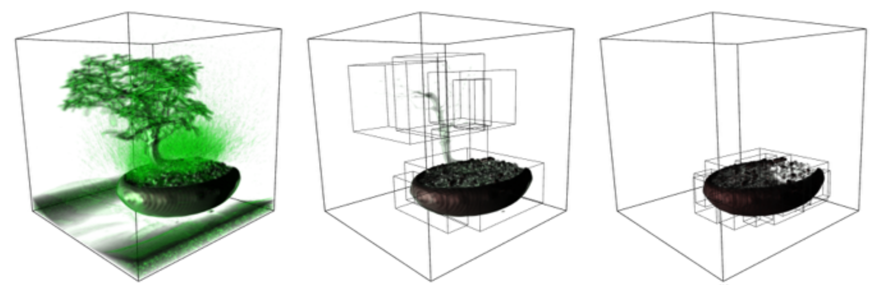
Results

4 datasets (3 well-known, 1 from microbiology, courtesy Kei Ito, University of Cologne), **3 transfer functions**

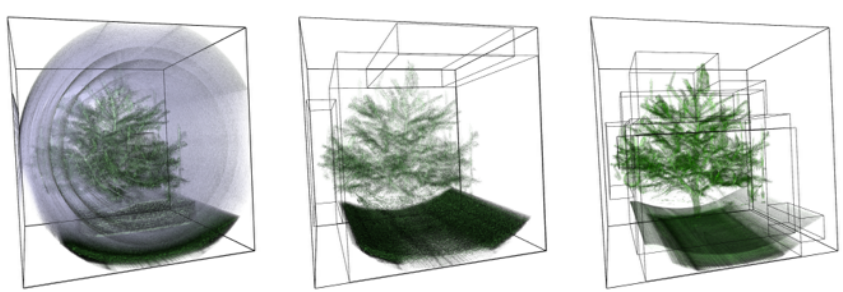
256^3 voxels



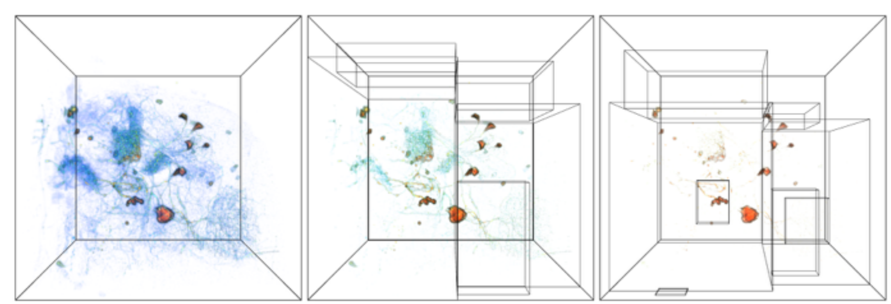
256^3 voxels



512^2 x 499 voxels

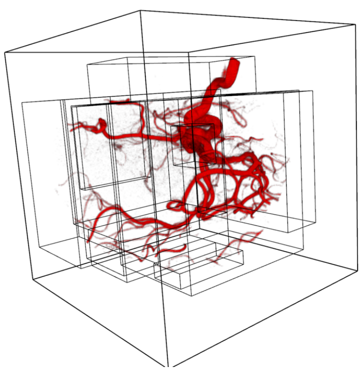


1000^2 x 910 voxels



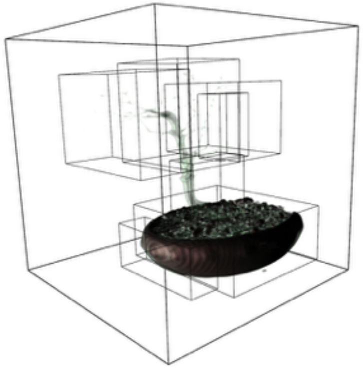
Results

Intel Core i7-3960X processor, 6 Cores, 12 Threads,
Times in sec., three different transfer functions



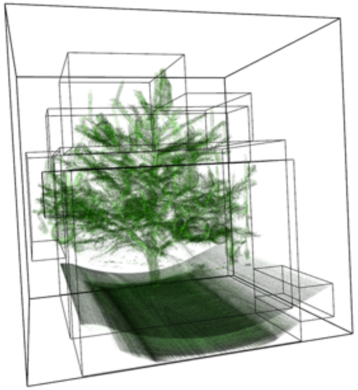
256^3 voxels

	SVT	SPLIT	TOTAL
SERIAL	*0.179*	*0.002*	*0.181*
PAR.	*0.020*	*0.002 - 0.016*	***0.022 - 0.036***



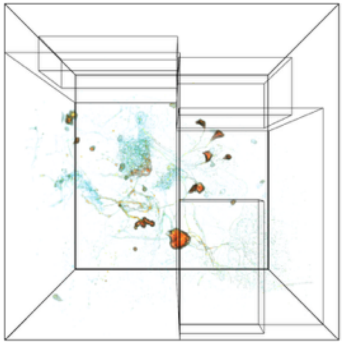
256^3 voxels

	SVT	SPLIT	TOTAL
SERIAL	*0.180*	*0.001*	*0.181*
PAR.	*0.026*	*0.003 - 0.014*	***0.029 - 0.040***



512^2 x 499 voxels

	SVT	SPLIT	TOTAL
SERIAL	*1.436*	*0.004*	*1.440*
PAR.	*0.192*	*0.036 - 0.148*	***0.226 - 0.340***



1000^2 x 910 voxels

	SVT	SPLIT	TOTAL
SERIAL	*9.361*	*0.020*	*9.381*
PAR.	*1.103*	*1.692 - 4.114*	***2.795 - 5.217***

Conclusion

- Parallel *k*-d tree construction algorithm based on prior work by Vidal et al. (2008)
- Optimized for multi-core architecture
- Good scalability for moderately sized data sets, promising for larger data sets
  - Just meets our use case: moderately sized volumes from radiology
- Whole idea based on keeping underlying SVT data set in thread-local L1 memory to exploit parallelism
  - Wagers SVT construction time for split-plane sweeping overhead
  - A win: SVT construction time *the* dominant bottleneck with serial variant of the algorithm
- Future work: scale with larger datasets, distributed memory systems, construction on the GPU