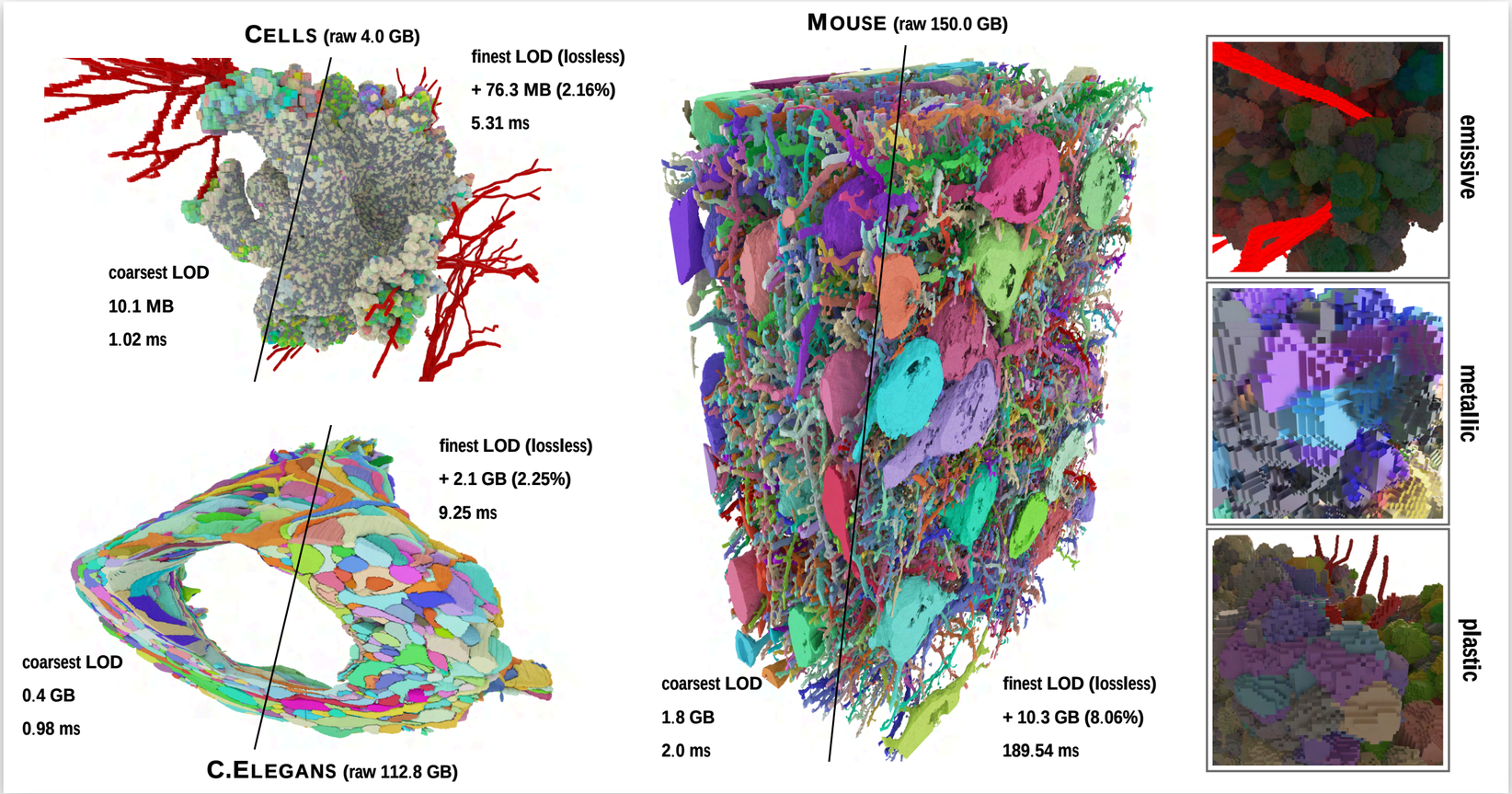


SVDAG Compression for Segmentation Volume Path Tracing

Mirco Werner†, Max Piochowiak†, and Carsten Dachsbacher

(† joint first authors)

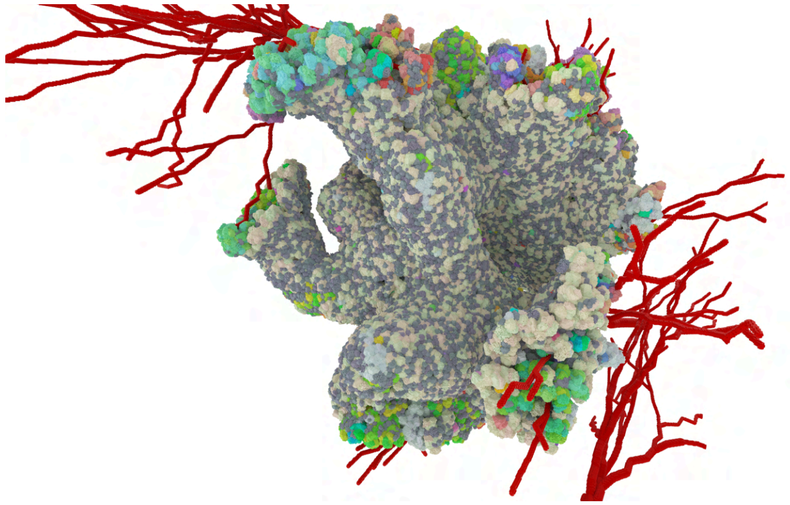
Karlsruhe Institute of Technology

Speaker notes

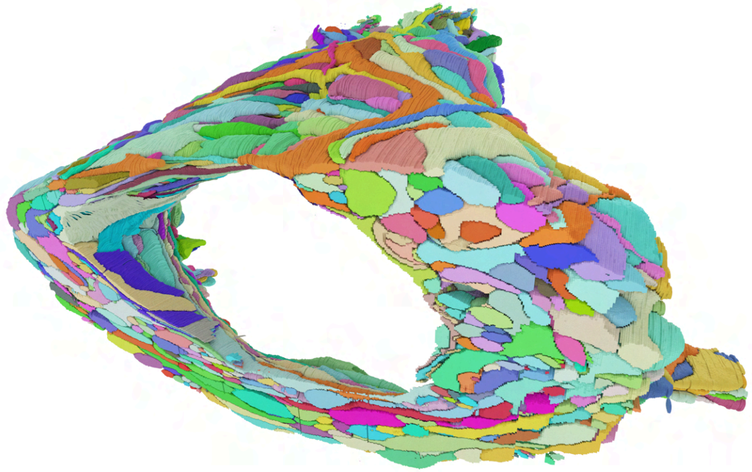
Hi, I'm Mirco and I'm presenting our paper SVDAG Compression for Segmentation Volume Path Tracing. I want to start with a quick introduction about segmentation volumes.

Segmentation Volumes

CELLS (Rosenbauer et al. 2020)



C.ELEGANS (Witvliet et al. 2021)



MOUSE (Motta et al. 2019)

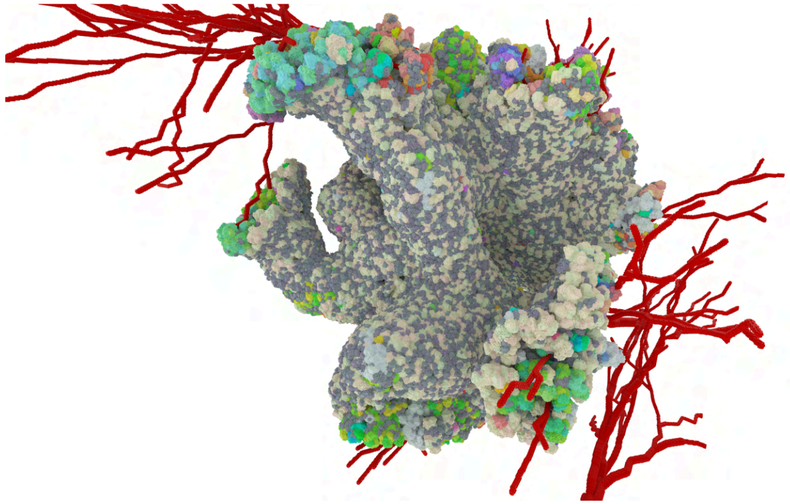
Segmentation Volume: $V(x) = l \in \mathbb{N}$

Speaker notes

Segmentation volumes are voxel data sets that are commonly used in a variety of domains like medicine, connectomics, or computational biology. These volumes store an integer label for each voxel. This segments the space into separate object regions, for example neurons in a mouse brain which you can see on the right.

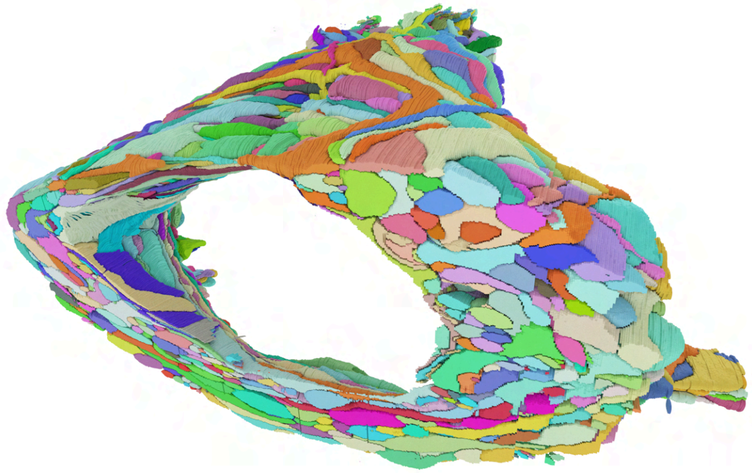
Segmentation Volumes

CELLS (Rosenbauer et al. 2020)



4 GB (32 bit/voxel)

C.ELEGANS (Witvliet et al. 2021)



113 GB (8 bit/voxel)

MOUSE (Motta et al. 2019)

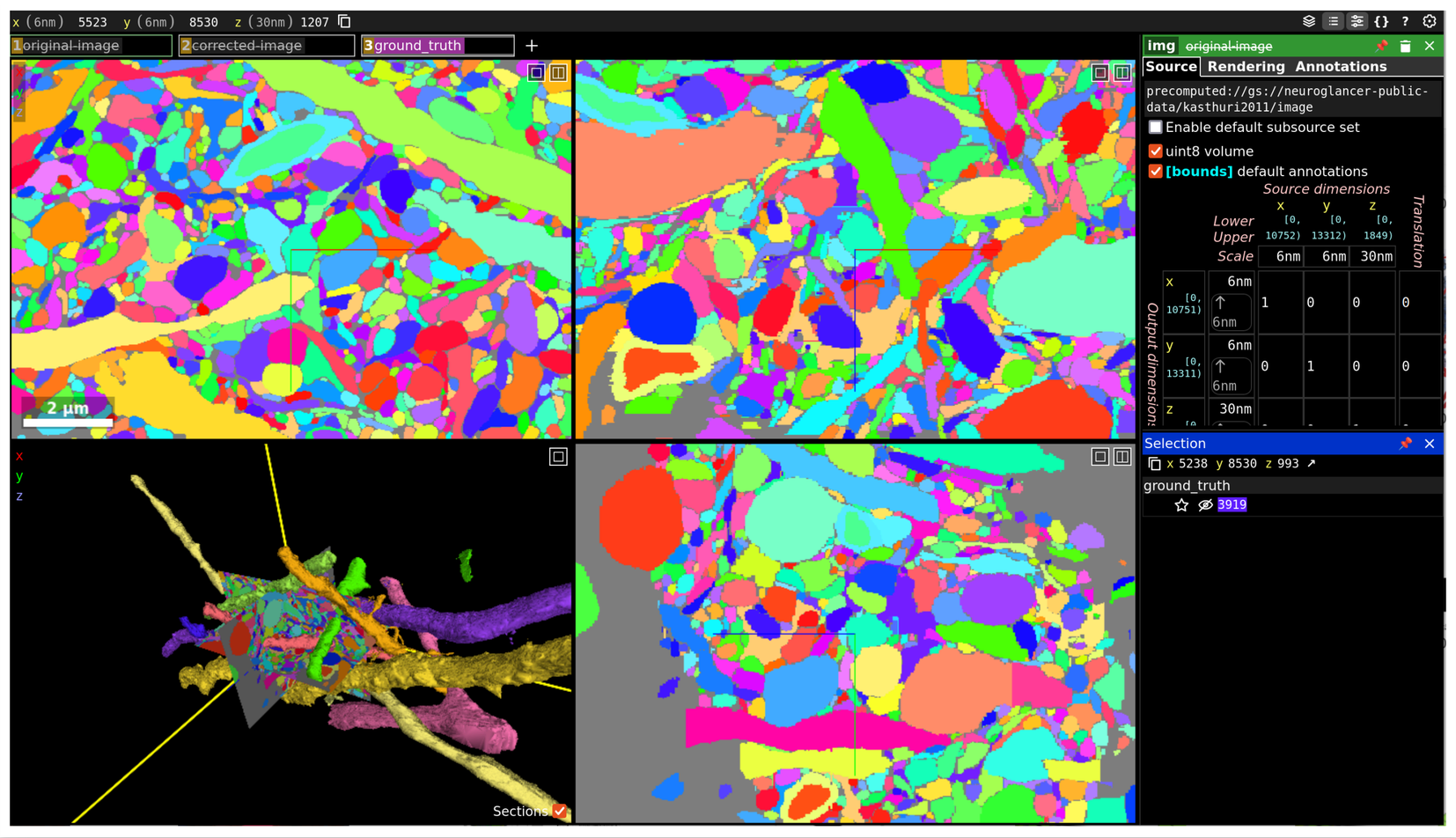
150 GB (8 bit/voxel)

Segmentation Volume: $V(x) = l \in \mathbb{N}$

Speaker notes

Segmentation volumes can become quite large, even for small label counts that allow to store volumes with few bits per voxel. These sizes easily exceed GPU memory, hindering interactive visual exploration and rendering.

Typical Rendering Workflow



Google Neuroglancer

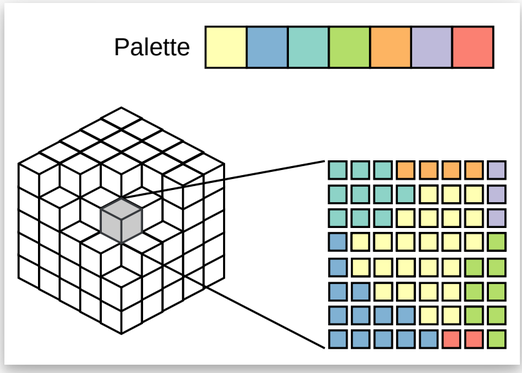
Speaker notes

Therefore, rendering workflows are typically a two stage process: First, interactive exploration is done in a low-fidelity visualization framework. Pictured here is Google neuroglancer which can display 2D slices of volumes as well as a small number of precomputed meshes of object region.

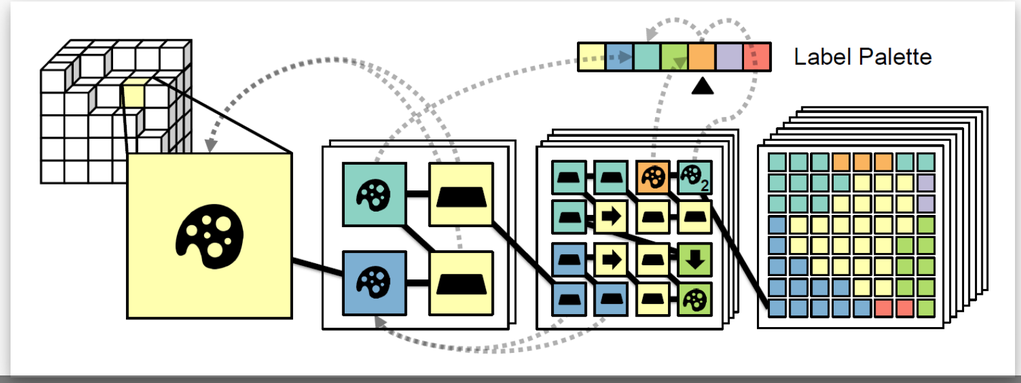
Typical Rendering Workflow

Brick-Wise Compression

- stores voxels in bricks
- label palette per brick
- $\lceil log_2(\|palette\|) \rceil$ bits per voxel



Neuroglancer Precomputed Format

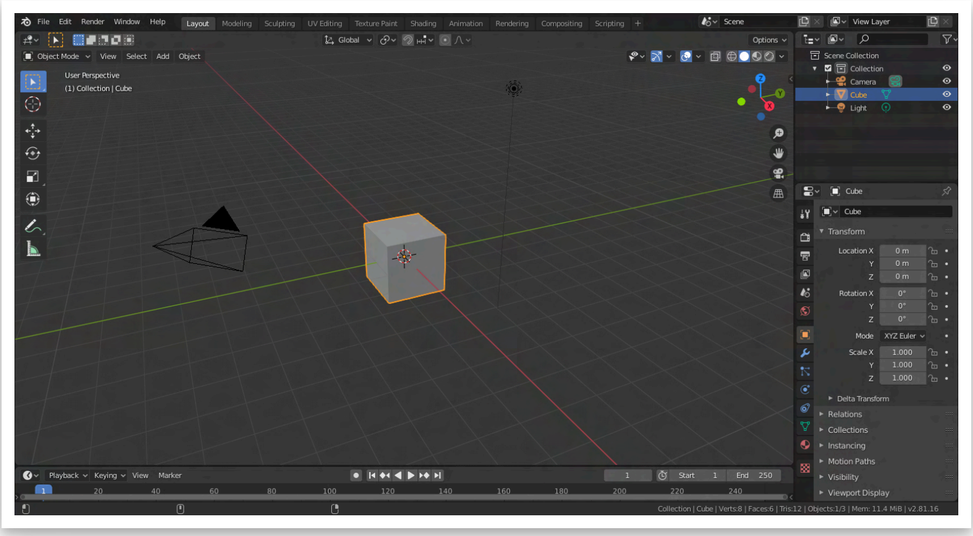


[Piochowiak*2023]
Fast Compressed Segmentation Volumes for Scientific Visualization. IEEE Vis

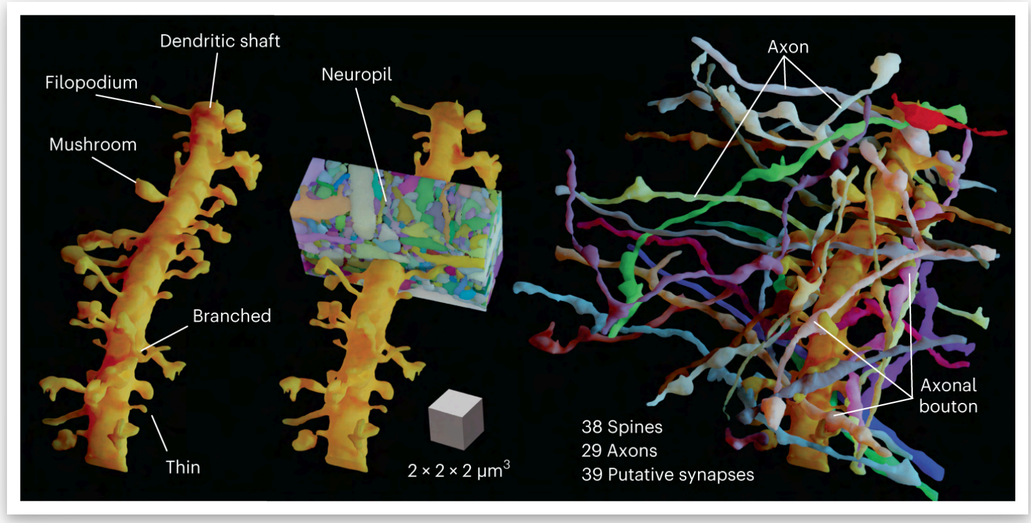
Speaker notes

Google neuroglancer uses a form a brick-wise compression: the neuroglancer precomputed format. This compression technique operates on a voxel basis: The volume is split into small bricks. For each brick, a palette of occurring voxel labels is stored. Voxels inside a brick then index into the palette using only a small number of bits. This voxel format is fast to compute and is suitable for 2D visualization methods. However, for 3D visualization, it does not directly map to raytracing hardware and typically relies on slower ray marching. More evolved brick-wise formats that achieve stronger compression additionally require larger caches for on-the-fly decompression during rendering.

Typical Rendering Workflow



Blender manual CC-BY-SA 4.0

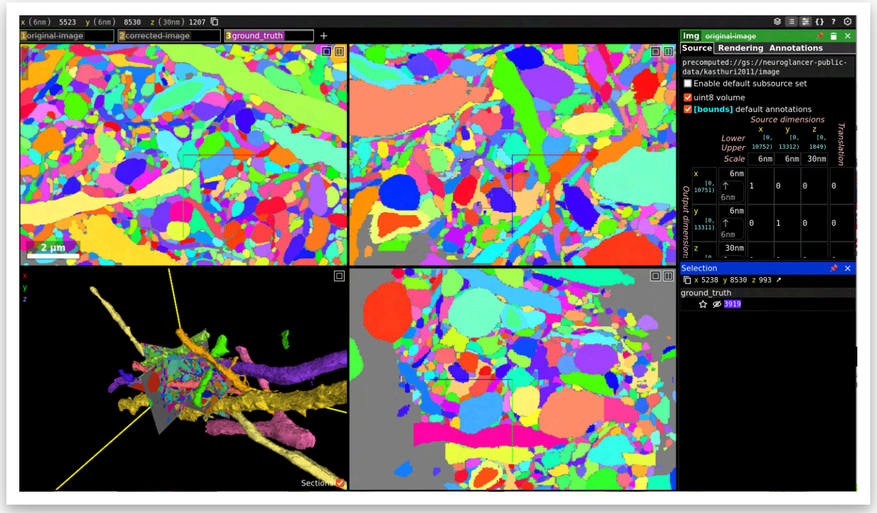


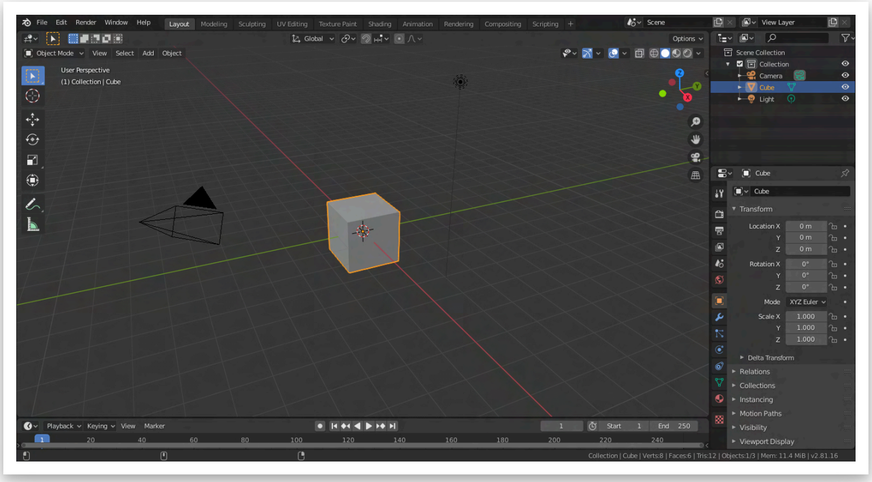
[Velicky*2023]
Dense 4D nanoscale reconstruction of living brain tissue, Nature methods

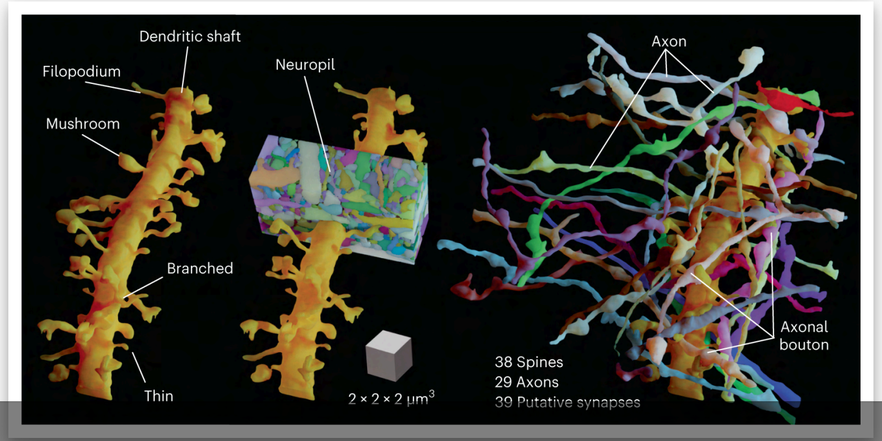
Speaker notes

Therefore, in a second step, often other external software is used to generate high quality renderings for science communication or publication purposes, for example with path tracing in Blender Cycles. To that end, the volume data has to be meshed or converted into another format first.

Typical Rendering Workflow







Speaker notes

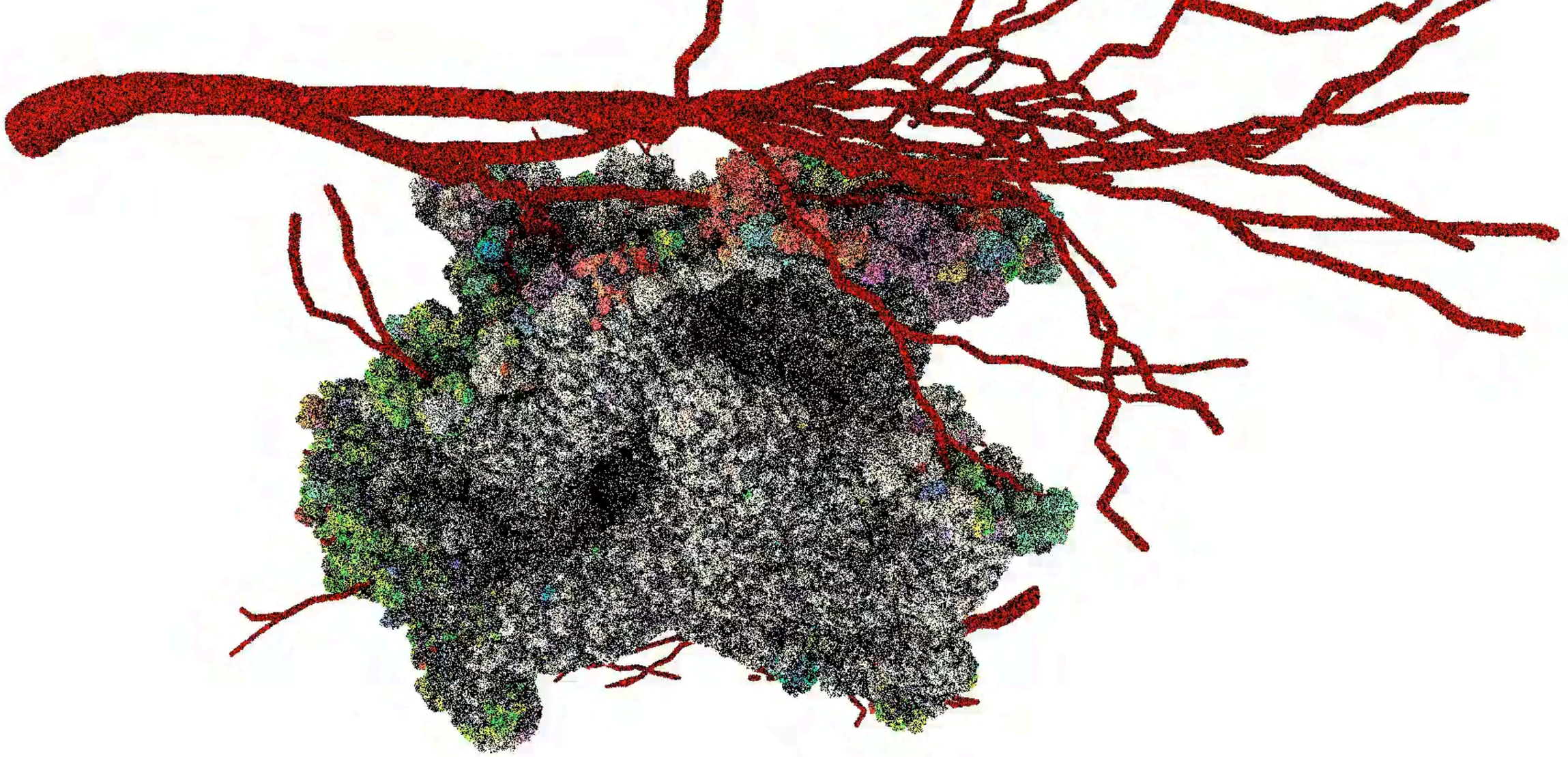
This existing process to generate high quality images is therefore typically slow: It requires switching between different data formats and software. The often CPU-bound rendering process takes minutes or hours to render one frame.

Compression for Segmentation Volumes

(our data structure)

Speaker notes

With our method, we solve these problems using a compressed data format that allows hardware accelerated GPU path tracing directly on the compressed large scale segmentation volumes. Our method offers precision and compression as in voxel-based representations as well as the rendering performance and hardware suitability of mesh based representations.

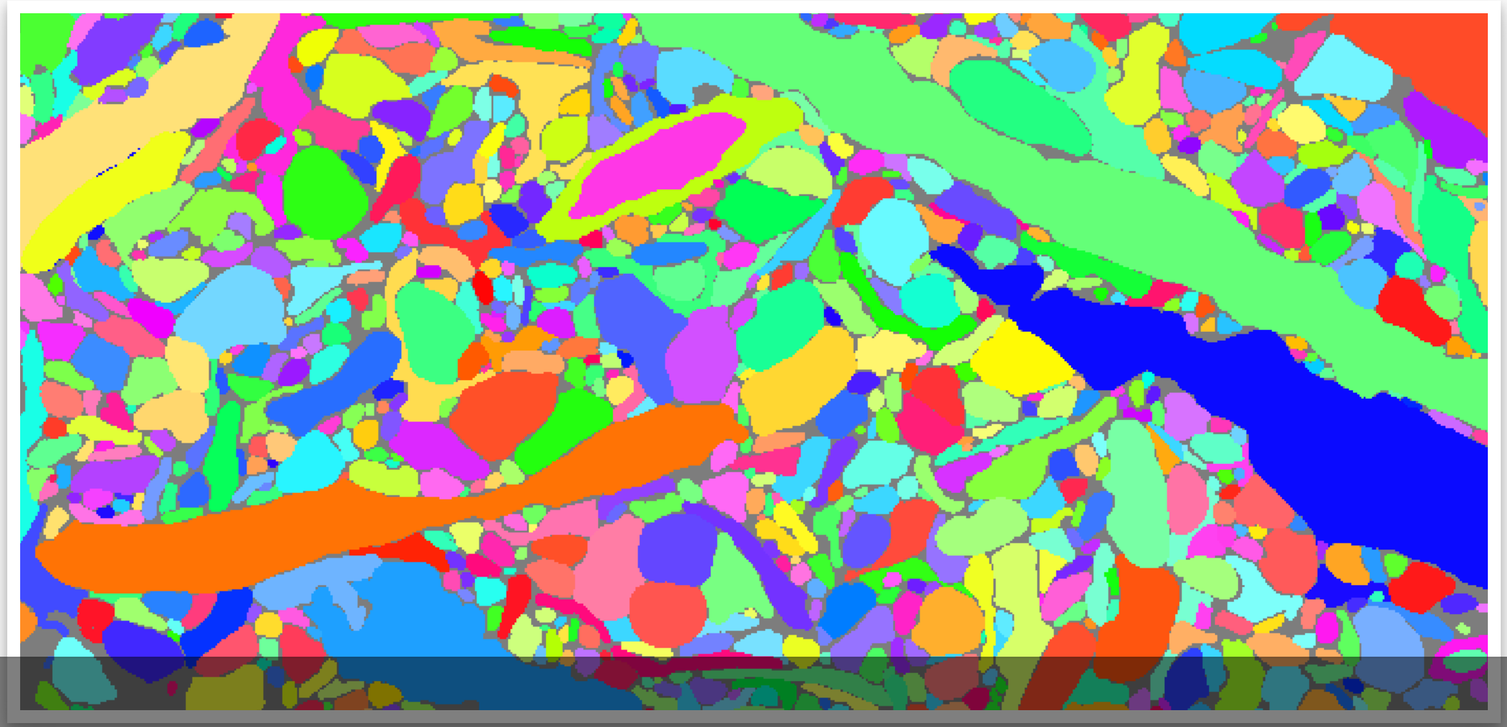


Speaker notes

Here I'm showing a video of our final framework. As you can see, we can interactively explore the segmentation volume. Once the camera is static, path tracing noise quickly vanishes and the image converges to the path-traced reference.

Compression for Segmentation Volumes - Observations

- key observation: large uniform areas with the same label

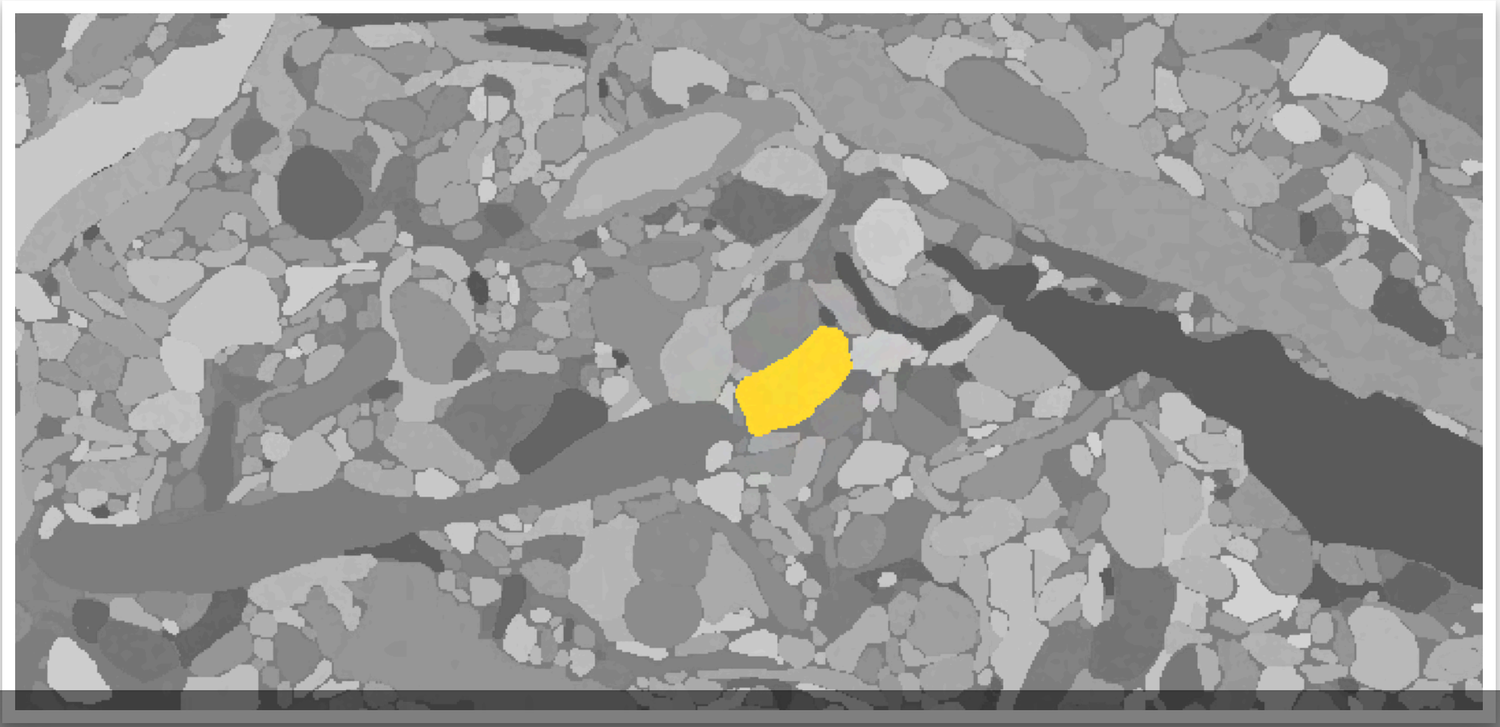


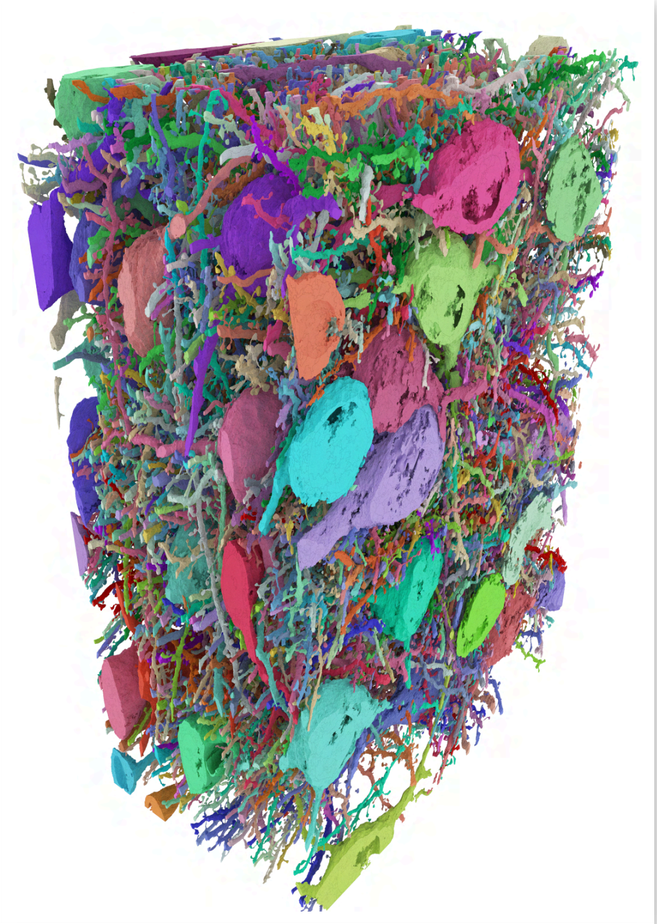
Speaker notes

How do we compress the segmentation volumes? For this, let's first take a look at a segmentation volume, on the right side in 3D and on the left side in 2D. You can clearly observe that there are large uniform areas where all voxels are assigned to the same label.

Compression for Segmentation Volumes - Observations

- key observation: large uniform areas with the same label
- compression idea: decouple label and voxel occupancy



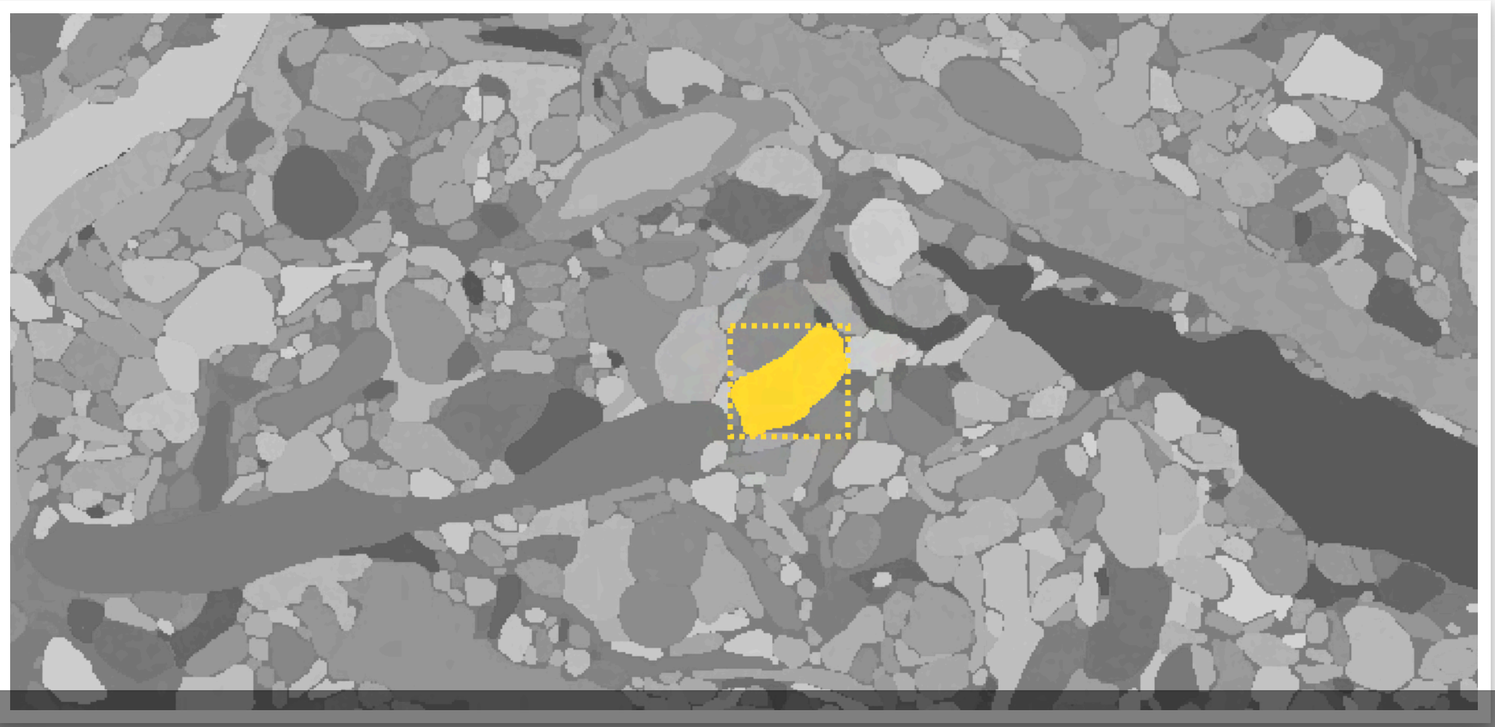


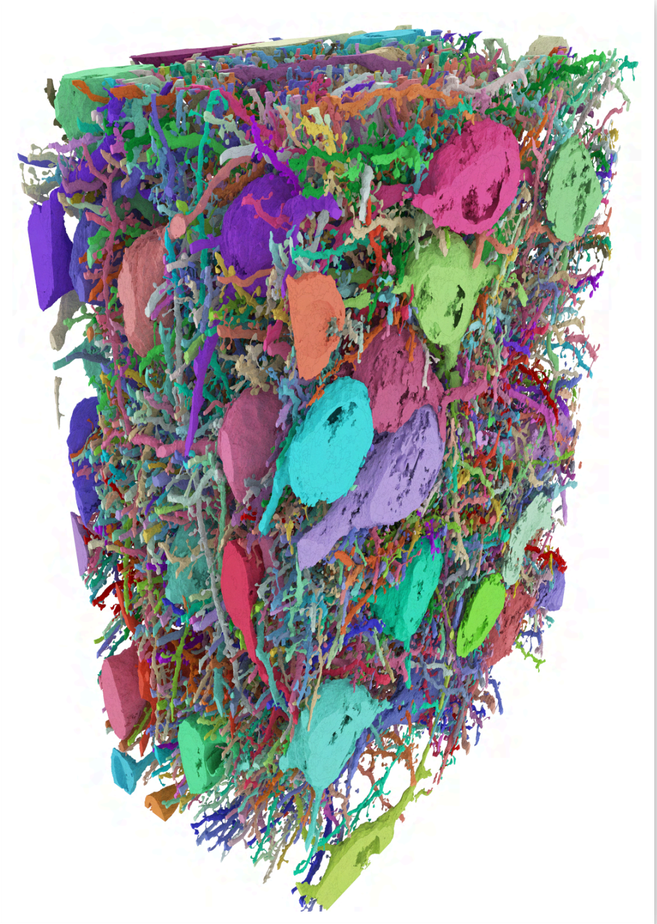
Speaker notes

Based on this observation, our compression idea is to decouple the label information from the voxel occupancy.

Compression for Segmentation Volumes - Observations

- key observation: large uniform areas with the same label
- compression idea: decouple label and voxel occupancy
  - AABB per label region



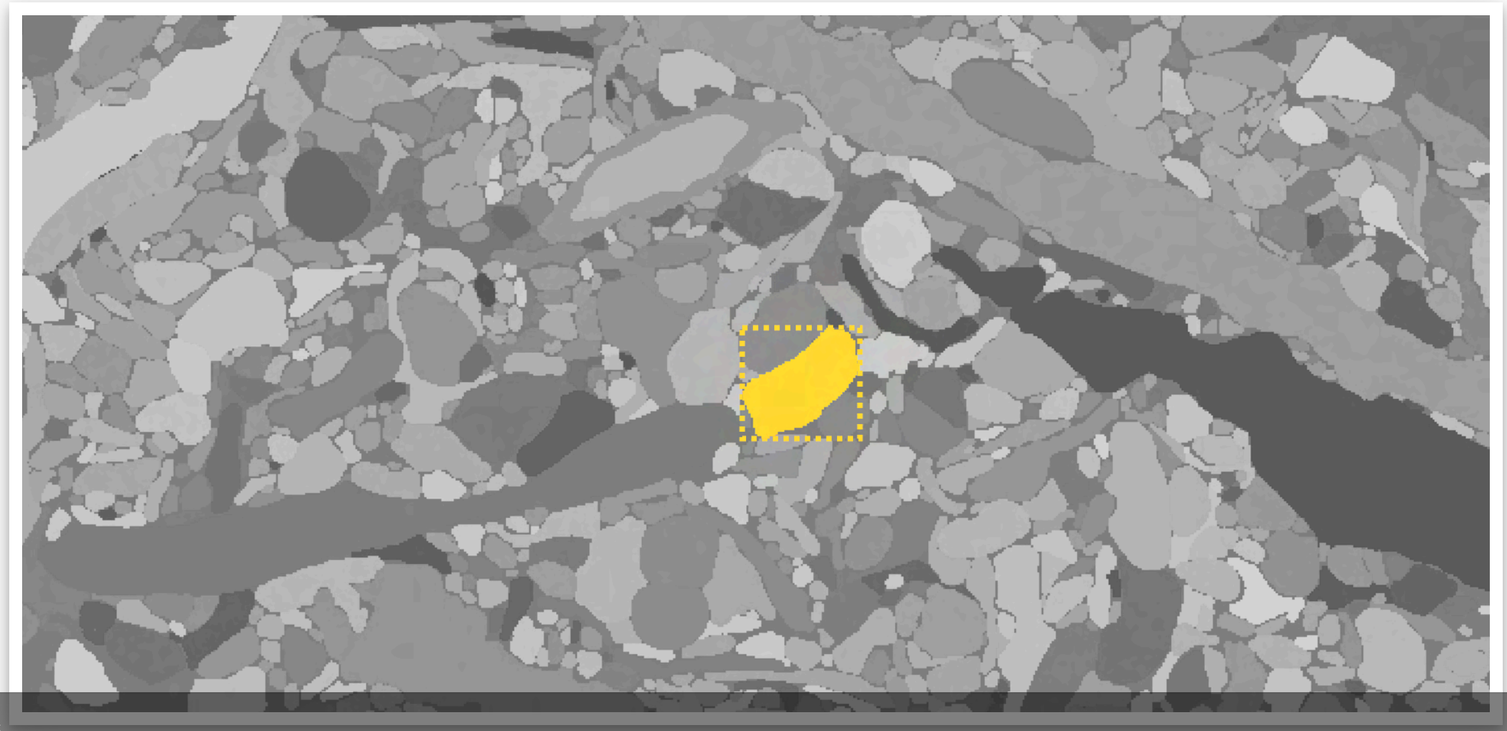


Speaker notes

Therefore, we can define an AABB per label region. All solid voxels that are assigned to this AABB now all implicitly have the same label and the label does not have to be stored for each individual voxel.

Compression for Segmentation Volumes - Observations

- key observation: large uniform areas with the same label
- compression idea: decouple label and voxel occupancy
  - AABB per label region
  - voxel occupancy becomes binary attribute inside region

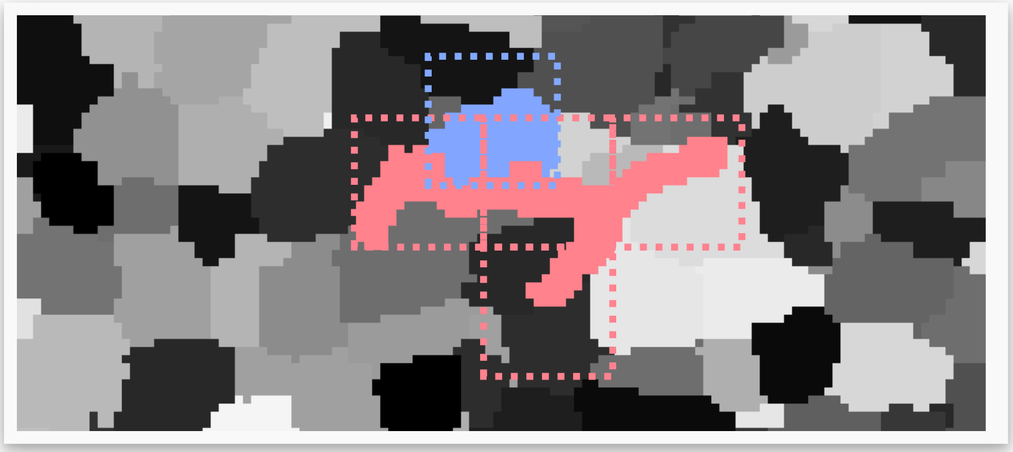


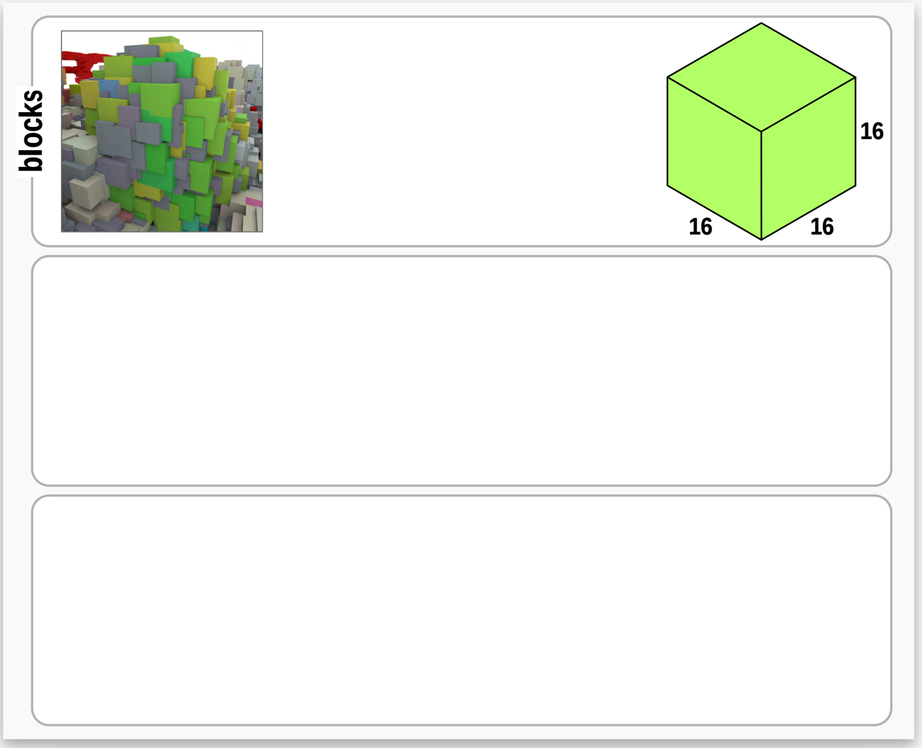
Speaker notes

This allows for much easier and stronger compression of voxel occupancy, since voxel occupancy, i.e. solid/empty, becomes a binary attribute inside the region.

Data Structure - Level 1 (blocks) ➡ decouple label from voxel occupancy

- split AABBs into blocks of fixed size (16^3)





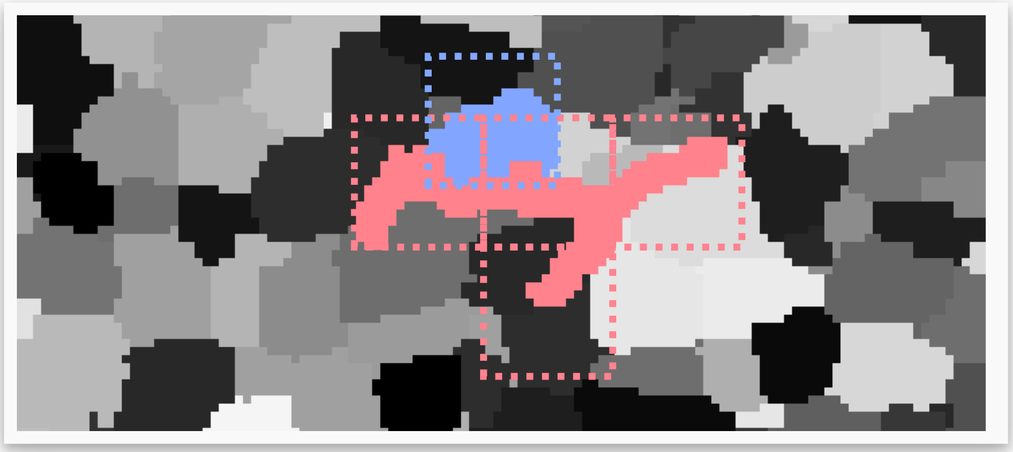
Speaker notes

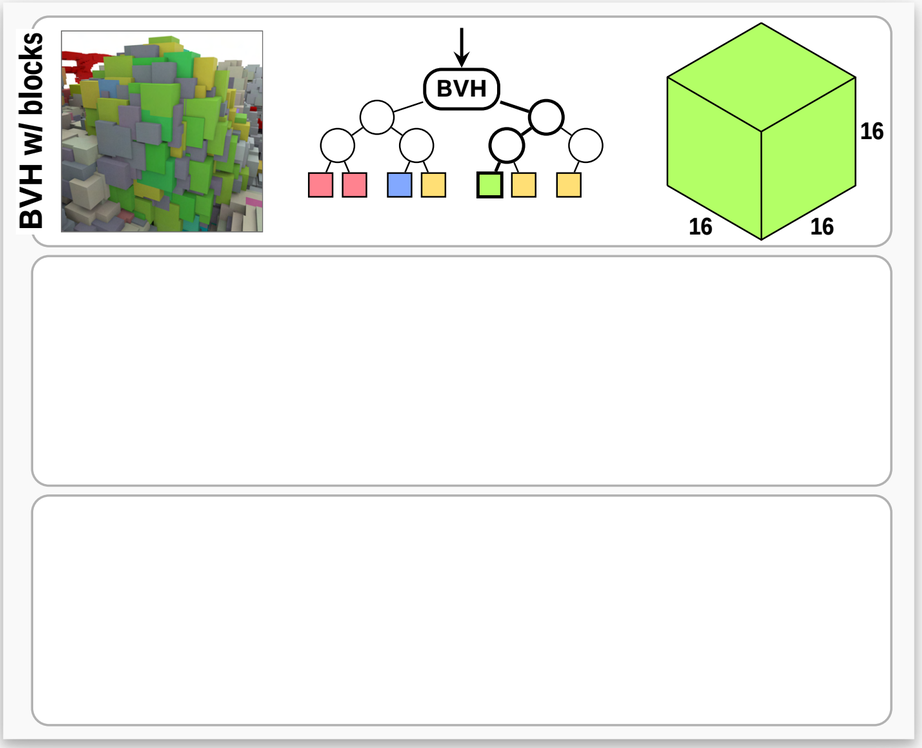
To make our compression work, we introduce a three level data structure. The goal of the first level is to decouple the label information from the voxel occupancy. Therefore, we split the AABBs into blocks of fixed size, 16 by 16 by 16.

Data Structure - Level 1 (BVH w/ blocks)

➡ decouple label from voxel occupancy

- split AABBs into blocks of fixed size (16^3)
- BVH for hardware-accelerated raytracing
  - blocks = BVH leaves, store label information



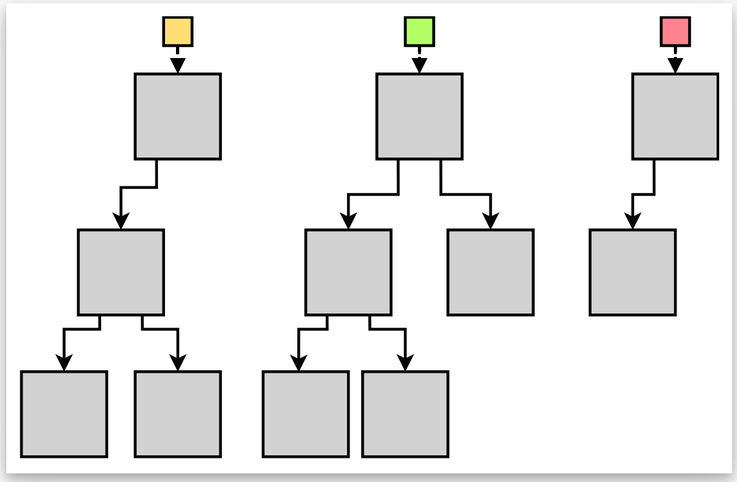


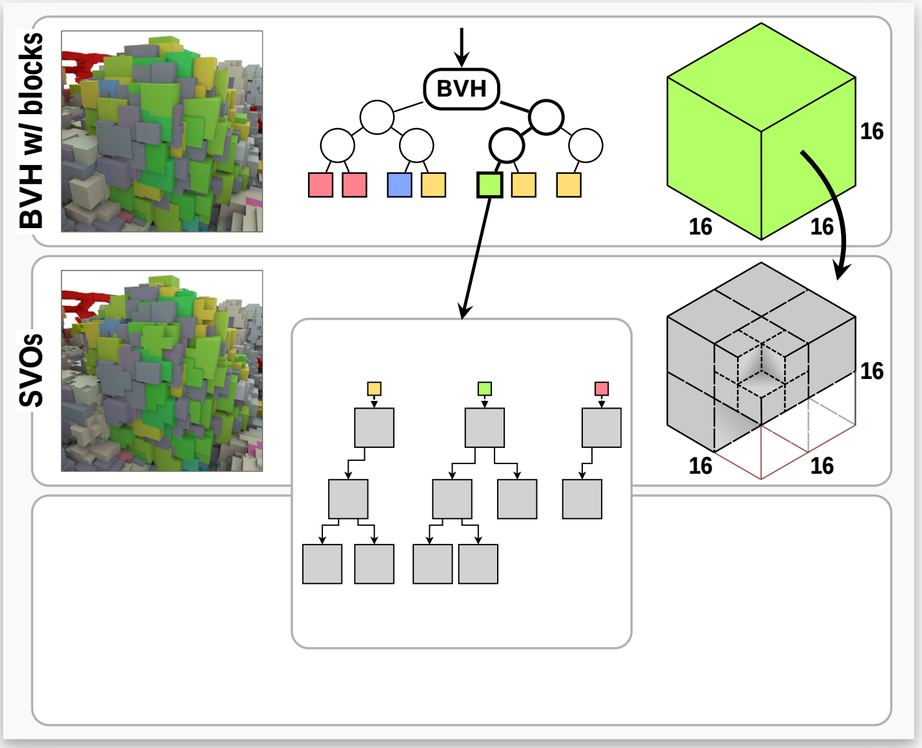
Speaker notes

For efficient rendering, we can then build a BVH for hardware-accelerated raytracing. Our blocks are the BVH leaves and store the label information. All solid voxels that are assigned to an block now all implicitly have the same label.

Data Structure - Level 2 (SVOs) ➡ encode voxel occupancy

- sparse voxel octree (SVO) [Laine*2010] in each block



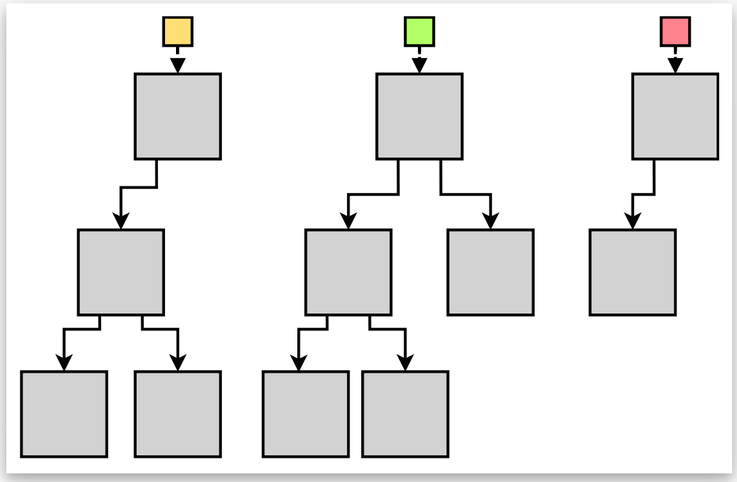


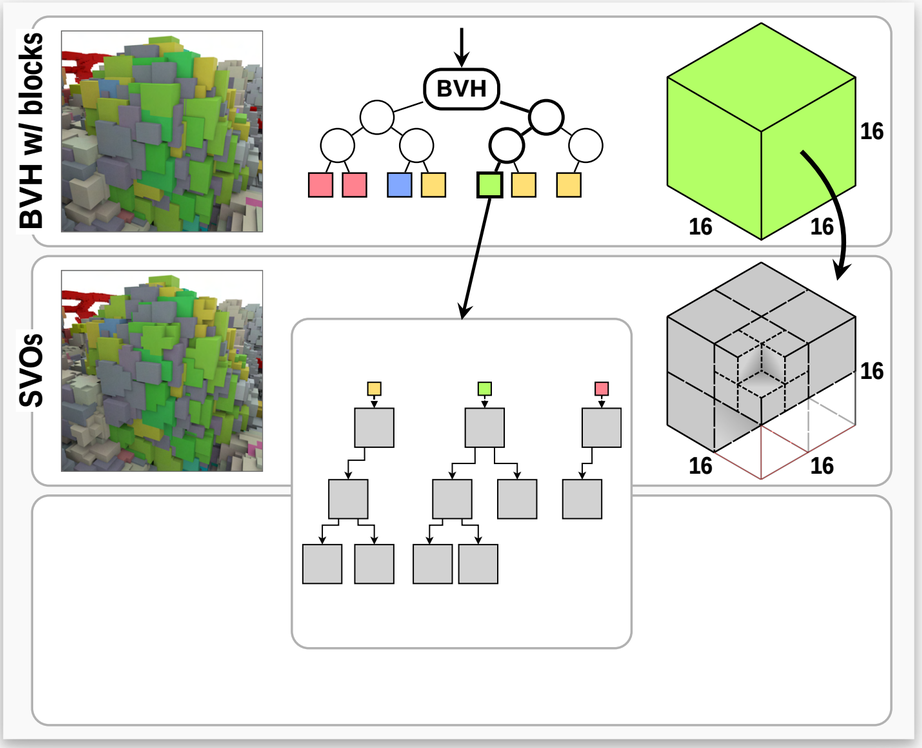
Speaker notes

Our second level of our data structure now encodes the actual voxel occupancy inside the blocks. Our first idea was to construct a SVO in each block. You can see a visualization of this on the left, where each block that contains the label information, here colored, stores this SVO.

Data Structure - Level 2 (SVOs) ➡ encode voxel occupancy

- sparse voxel octree (SVO) [Laine*2010] in each block
  - identical subtrees



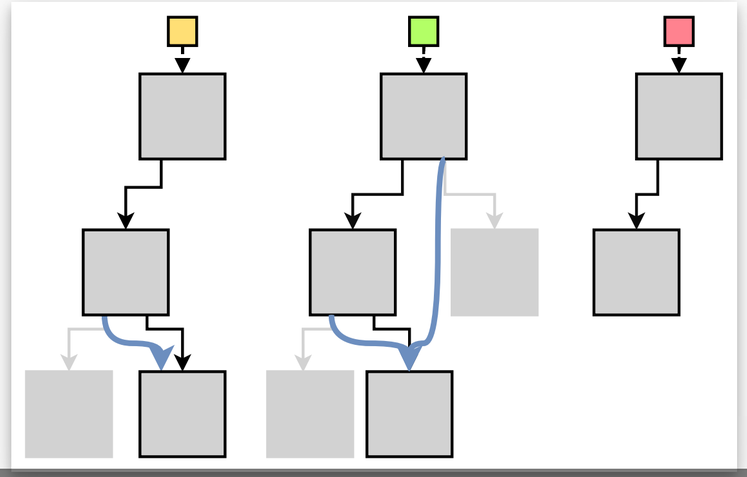


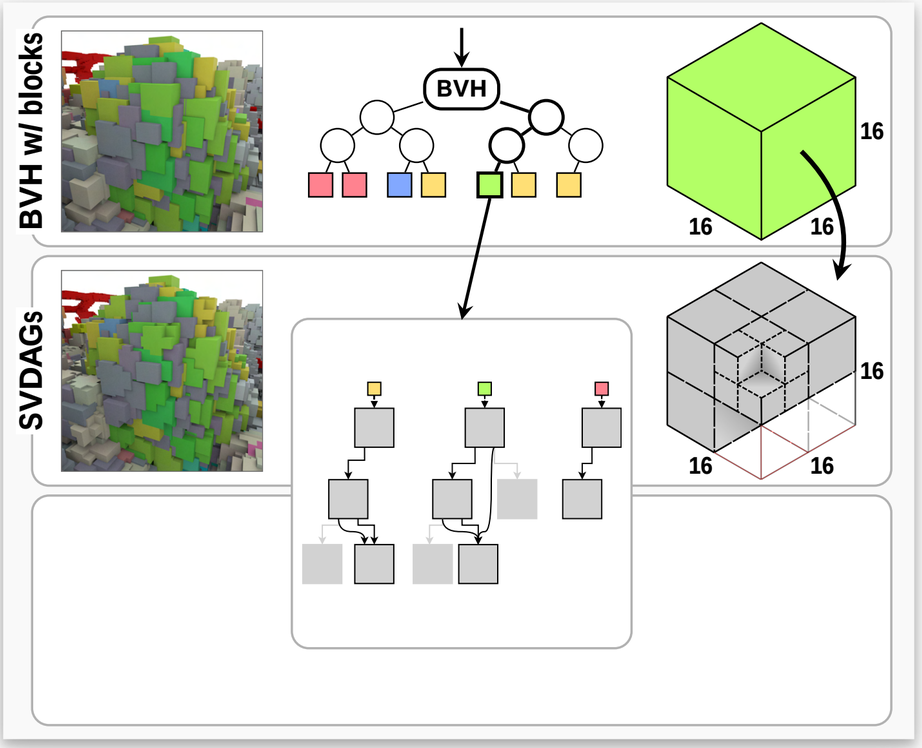
Speaker notes

Now when we take a look at the voxel occupancy inside each block, we can see that there are often identical subregions. This means, the SVO contains identical subtrees and therefore stores redundant information.

Data Structure - Level 2 (SVDAGs) ➡ encode voxel occupancy

- sparse voxel directed acyclic graph (SVDAG) [Kämpe*2013] in each block



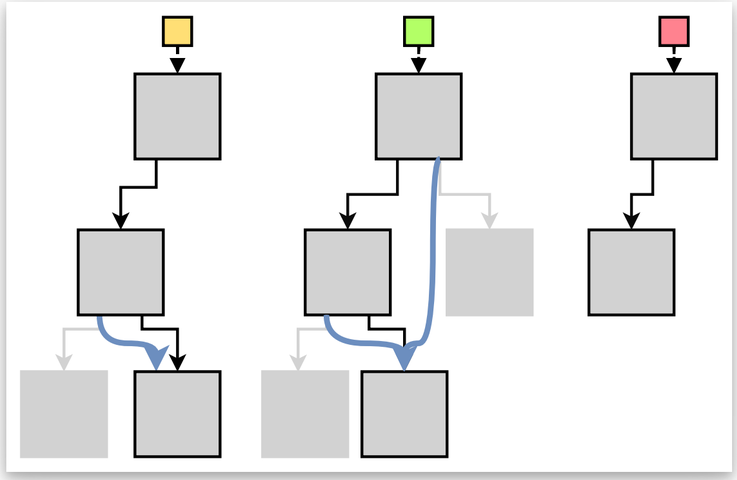


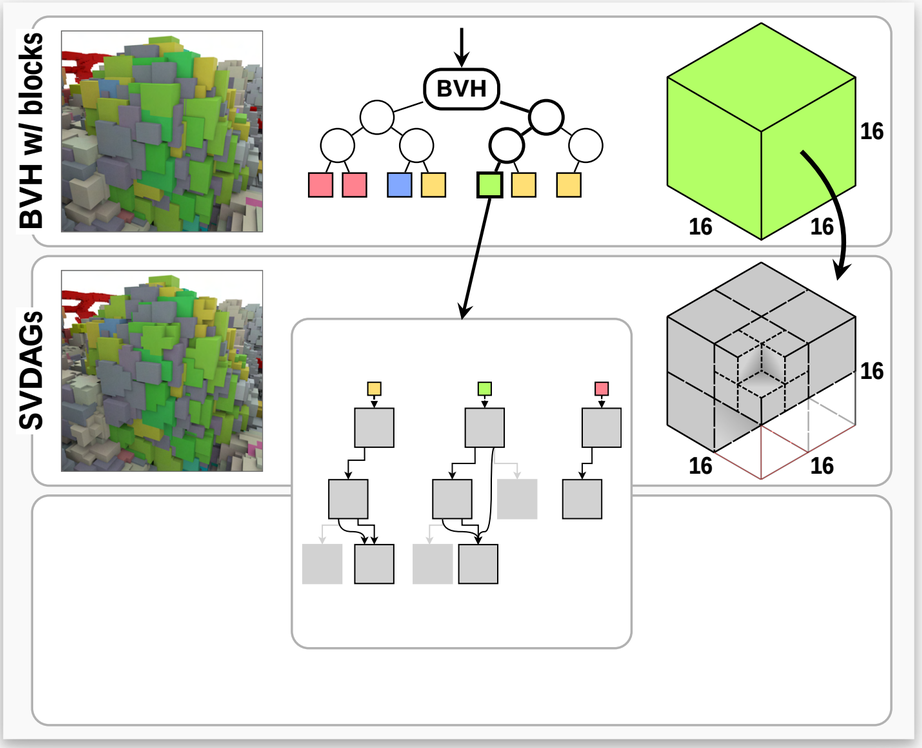
Speaker notes

Luckily, Kämpe and colleques found a solution for this by using a SVDAG instead. The idea here is that equal subtrees from the SVO are merged to transform this tree structure into a directed acyclic graph. This process is again shown here on the left with some of the now unused original SVO nodes grayed out and re-directed pointers. After this step we now have one SVDAG in each block.

Data Structure - Level 2 (SVDAGs) ➡ encode voxel occupancy

- sparse voxel directed acyclic graph (SVDAG) [Kämpe*2013] in each block
  - identical subtrees across blocks with different labels
  - (no label information stored)



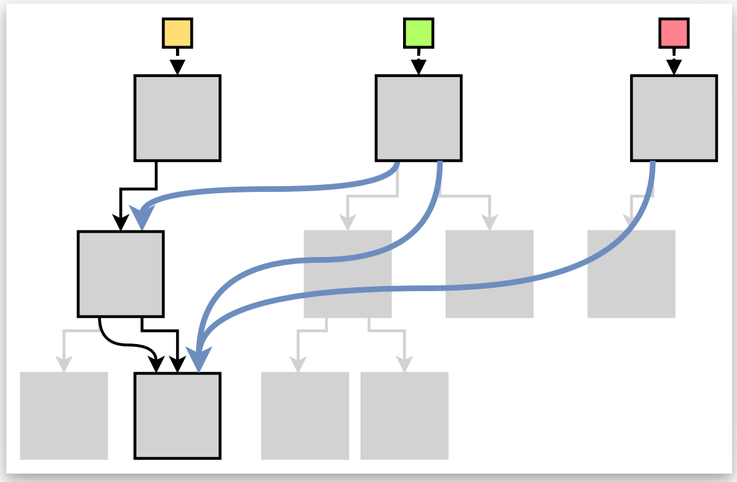


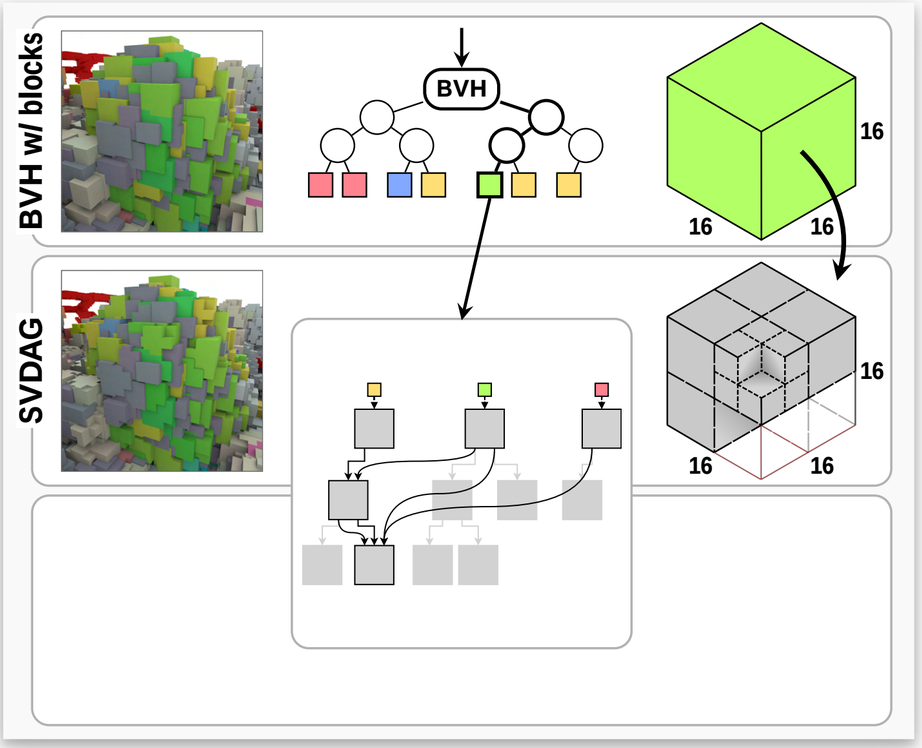
Speaker notes

Of course, there are also identical subtrees across blocks with different labels. Since we just encode the binary voxel occupancy in the SVDAGs, we can merge all these individual SVDAGs...

Data Structure - Level 2 (SVDAG) ➡ encode voxel occupancy

- single large SVDAG across all blocks



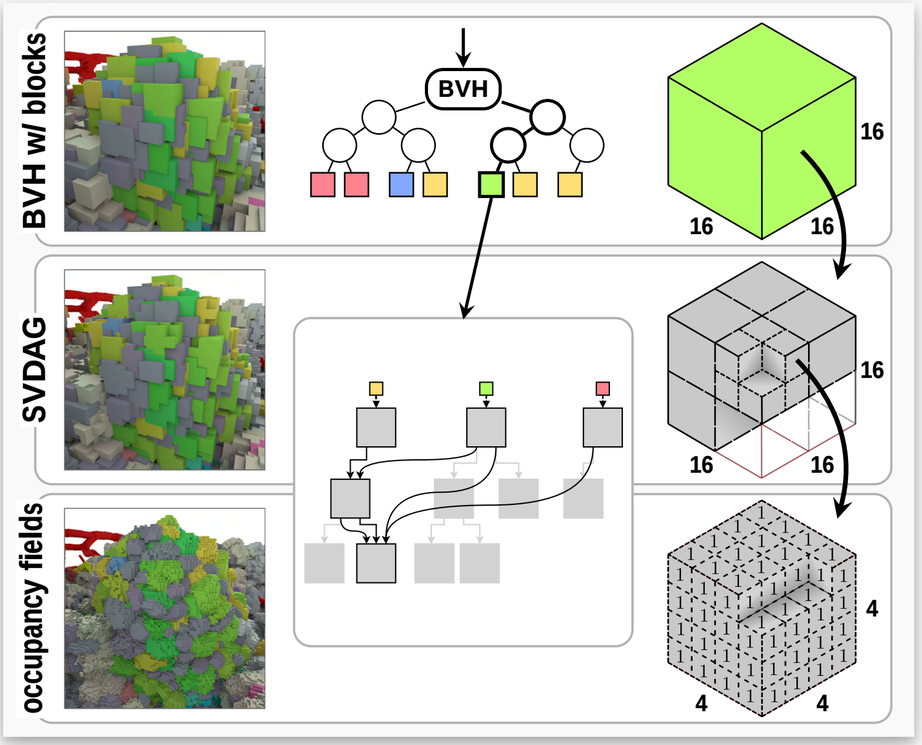


Speaker notes

into a single large SVDAG that is shared across all blocks. Again visualized here on the left.

Data Structure - Level 3 (Occupancy Fields) ➡ improve traversal performance

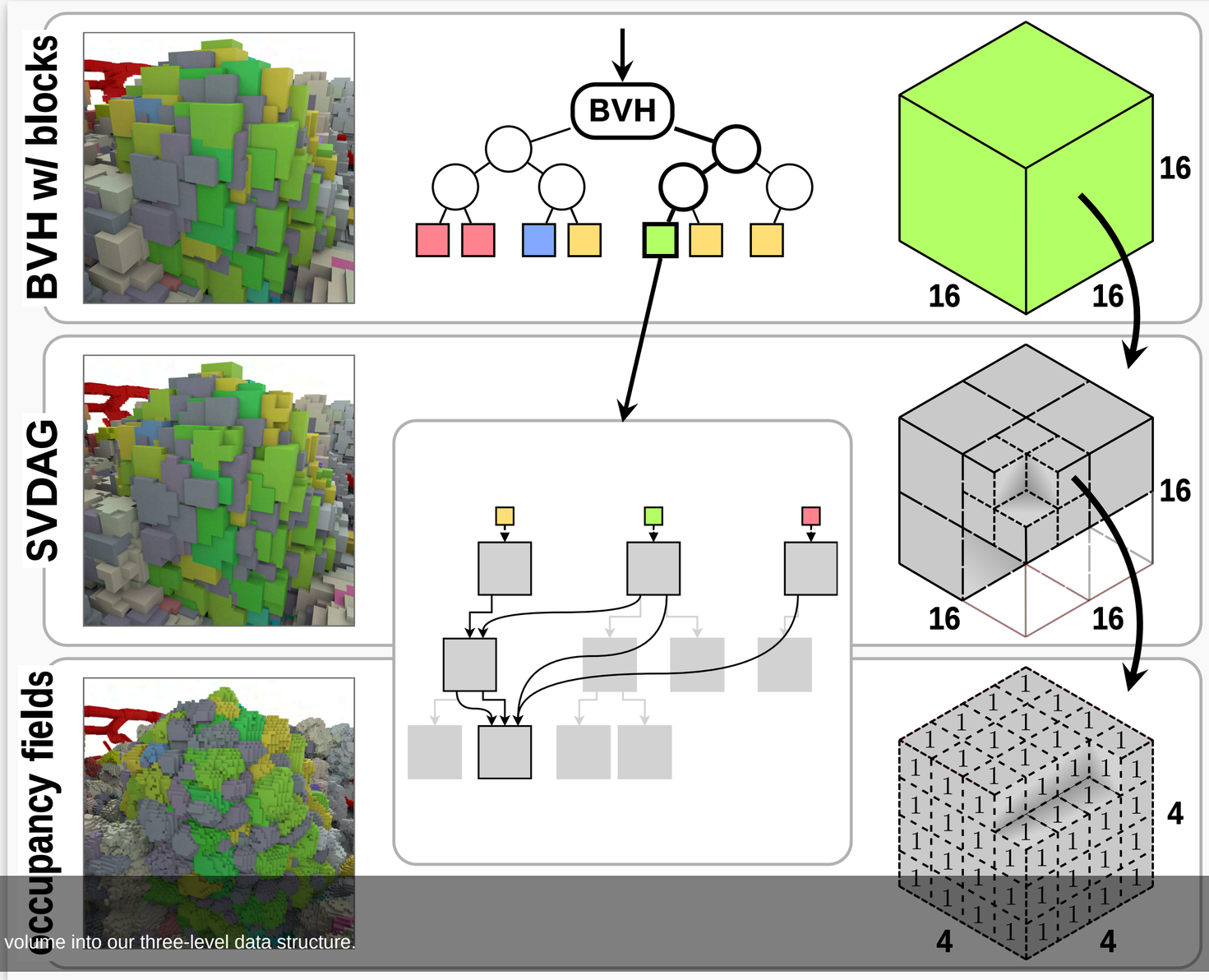
- store occupancy directly in 4^3 SVDAG leaf in bitmask
  - $4^3 = 64$ bits
- less VRAM accesses



Speaker notes

The goal of our third and last level is to improve traversal performance. As you can imagine, loading large SVDAG nodes during traversal introduces many unaligned and random VRAM accesses. We can re-define an area of 4 by 4 by 4 voxels as an SVDAG leaf node. The voxel occupancy inside a leaf can be stored directly in the leaf node using a bitmask using just 64 bits. We call this bitmask or bitfield occupancy field. With this optimization less VRAM accesses are required during traversal since we reach leaf nodes quicker with less steps.

Data Structure



Speaker notes

With that we have compressed the segmentation volume into our three-level data structure.

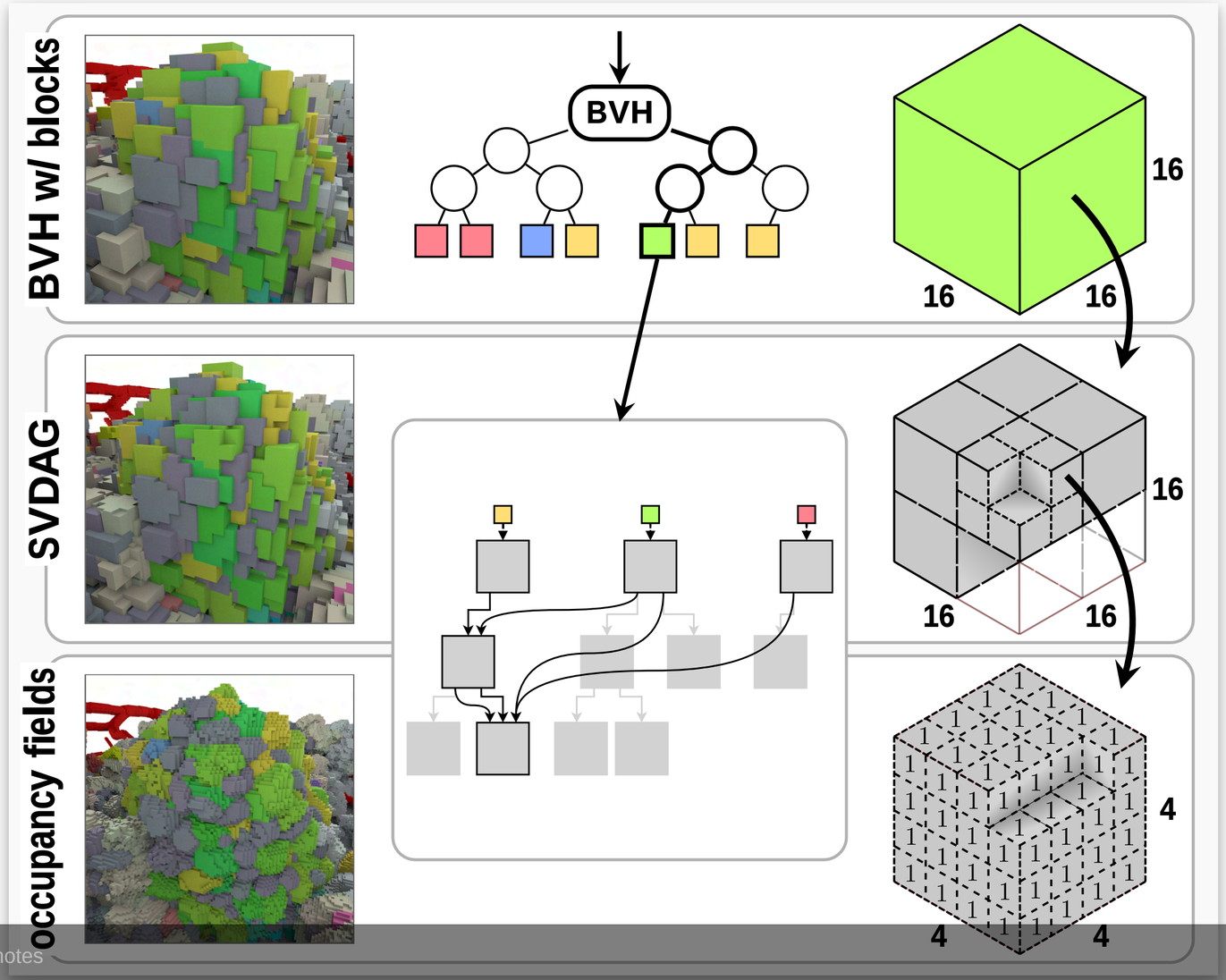
Compressed Segmentation Volume Path Tracing

(traversal of our data structure)

Speaker notes

Now let's continue with how we perform path tracing on our compressed segmentation volume. So how we traverse our data structure.

GPU Traversal

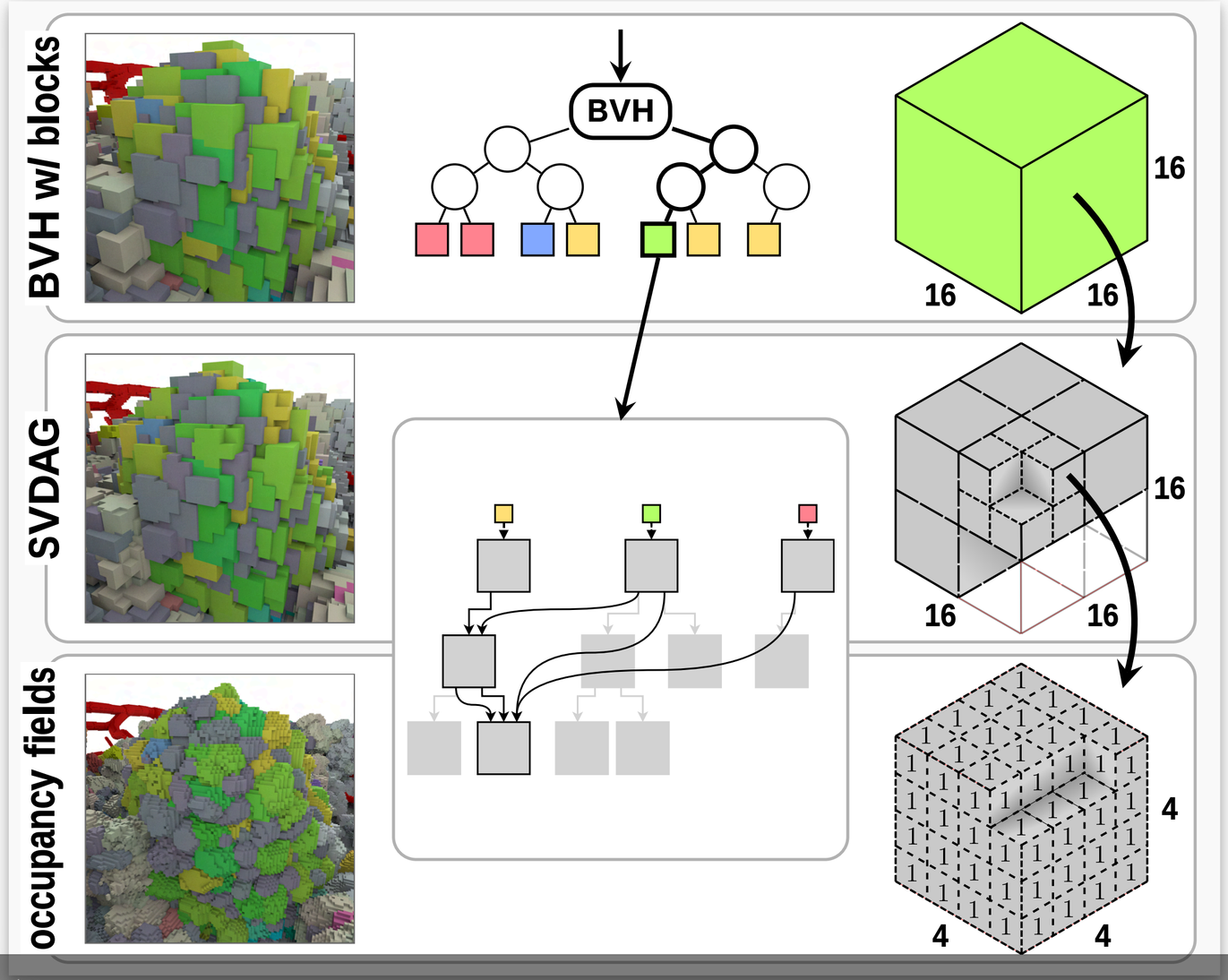


- hardware-accelerated ray tracing
  - invokes custom intersection shader

Speaker notes

You might have already noticed that this data structure maps really well to GPU hardware. This means for our first level, we can just input our blocks into a BVH builder and then use hardware-accelerated ray tracing to find intersections with the blocks. Once an block is intersected with, a custom intersection shader is invoked.

GPU Traversal



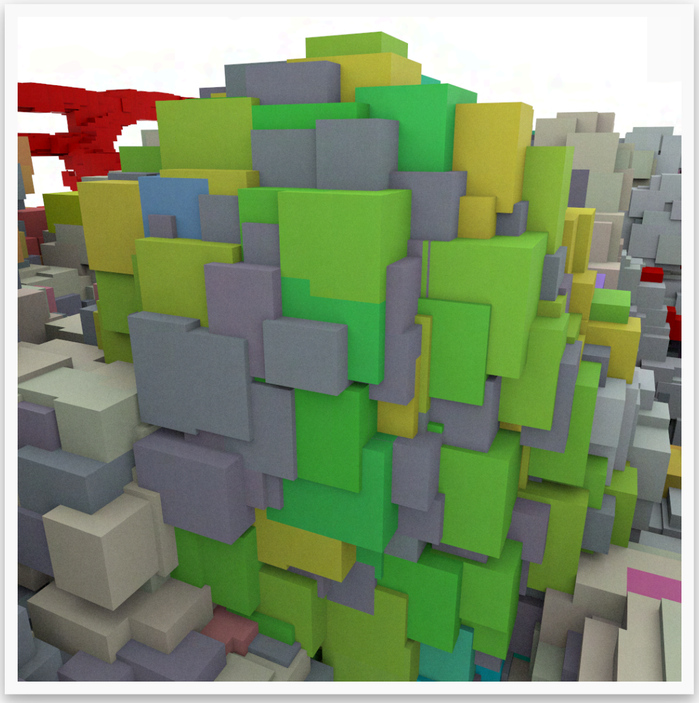
- hardware-accelerated ray tracing
  - invokes custom intersection shader

- query label and pointer to the SVDAG root node
- (multi-level) 3D-DDA traversal of SVDAG and occupancy field

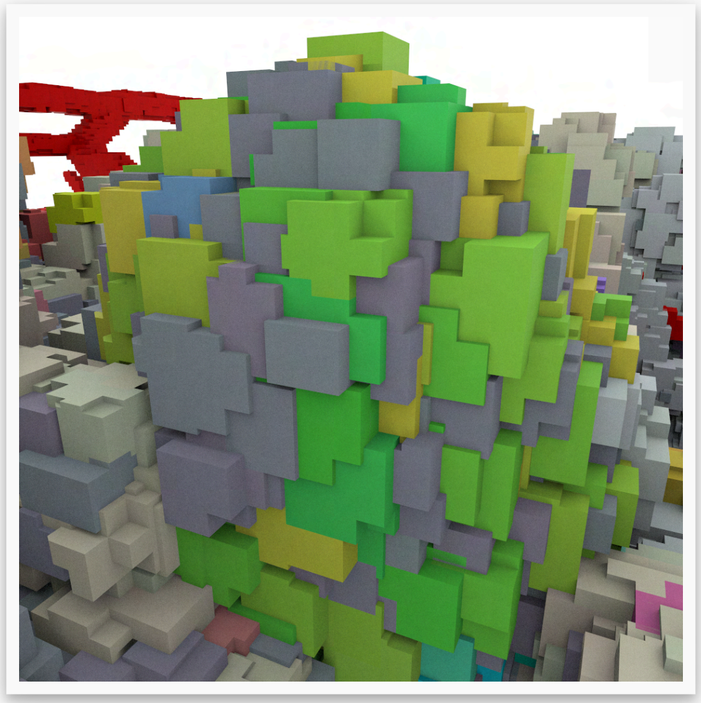
Speaker notes

In this custom intersection shader, we query the label information to load material properties and query the pointer to the SVDAG root node. Using multi-level 3D-DDA we can then manually traverse the SVDAG and the occupancy fields.

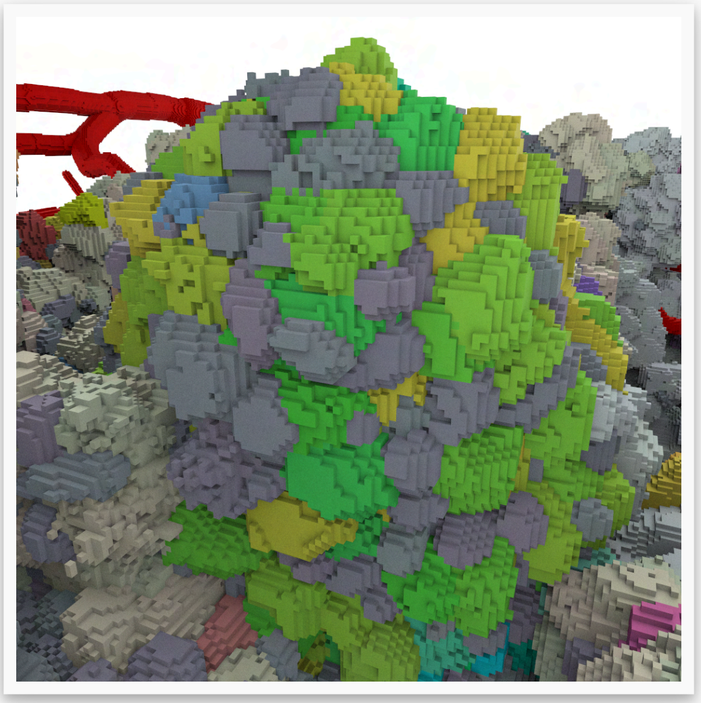
Level of Detail



block
(16^3)



SVDAG leaf
(4^3)

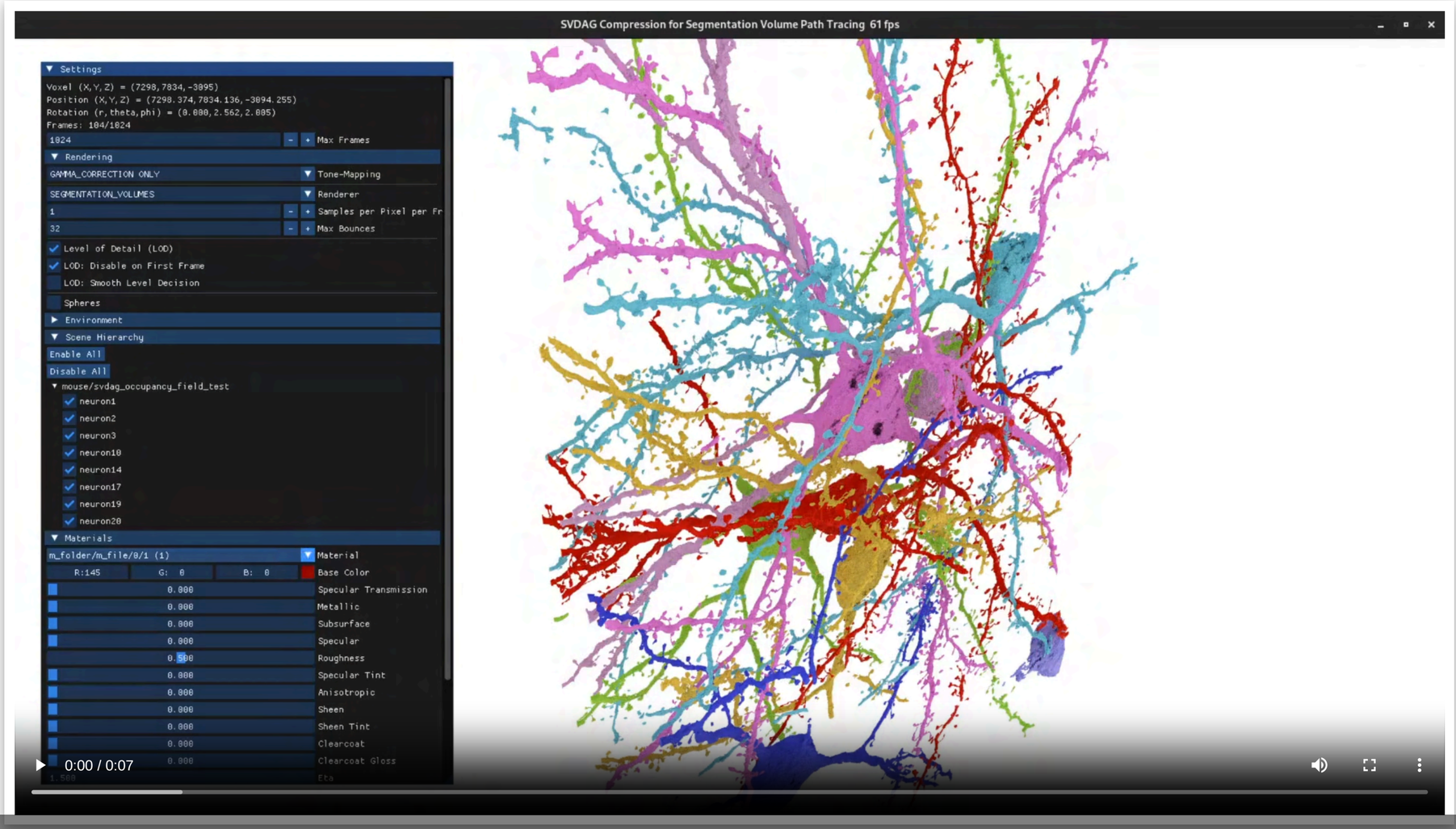


occupancy field
(1^3)

Speaker notes

Our method naturally allows for a level of detail scheme. Here on the slide you can see three possible levels of detail. On the left, we stop traversal directly after hitting a block. In the center, we traverse the SVDAG all the way down to a leaf node. And on the right, the individual voxels can be rendered by also traversing the occupancy fields. It's of course possible to dynamically choose the level of the detail based on the distance from the intersected block to the camera.

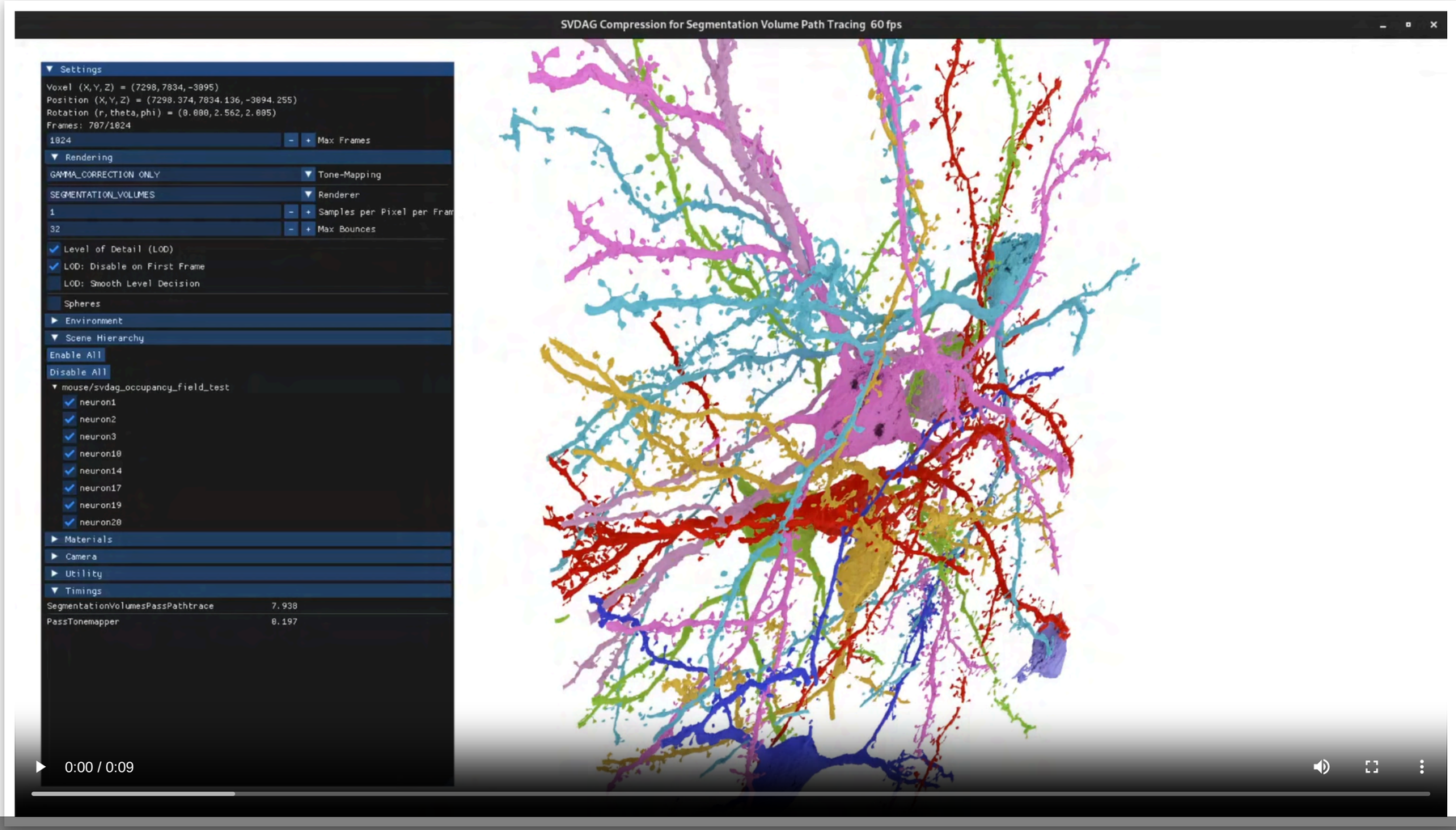
Transfer Function Editing

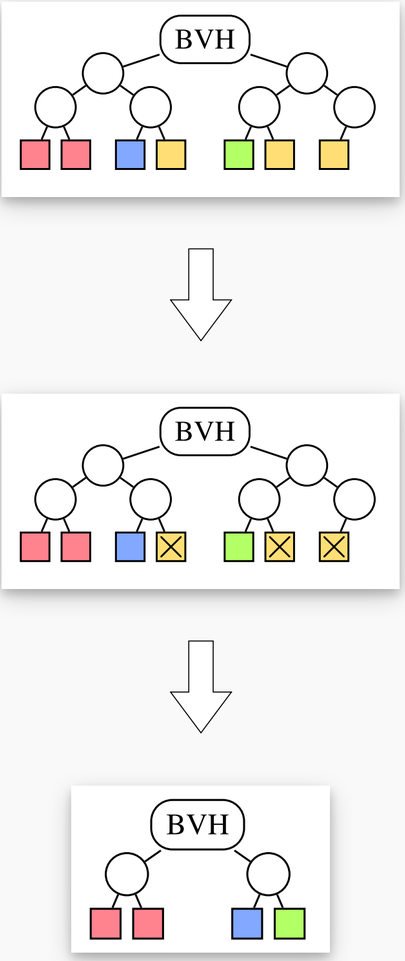


Speaker notes

Another interseting property of our method is that it allows for interactive editing of the transfer function. We can quickly adjust the material attributes corresponding to a specific label by updating the material buffer.

Transfer Function Editing





Speaker notes

When we would like to disable entire label regions, we can disable the corresponding blocks and quickly re-build the BVH to achieve an optimized traversal performance.

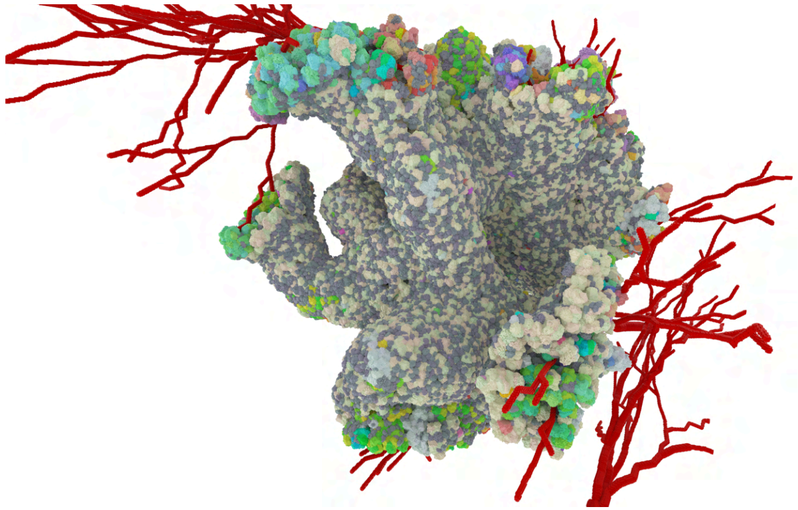
Results

Speaker notes

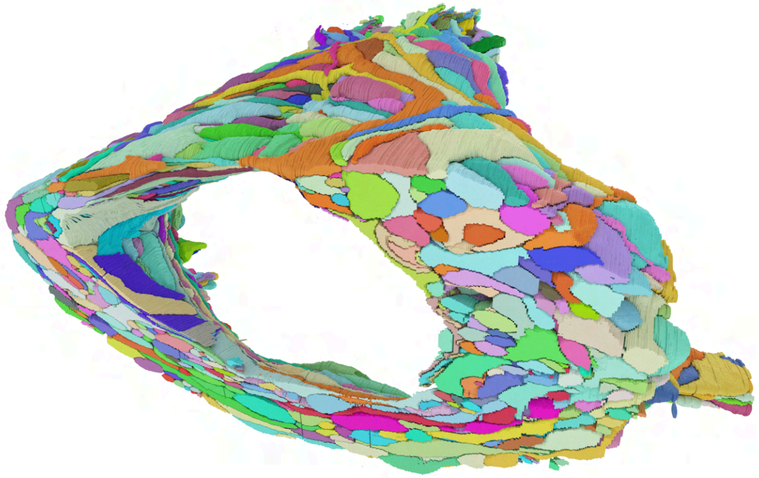
Now let's take a look at some results.

Datasets

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

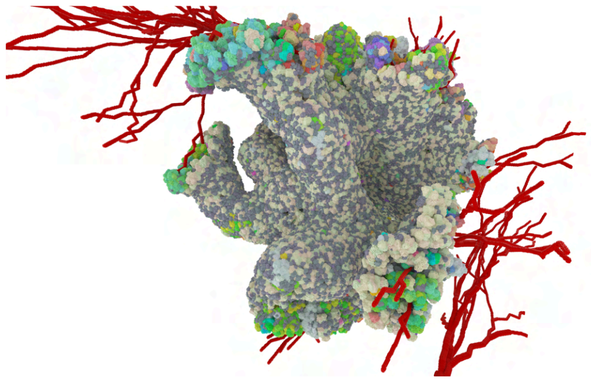
Speaker notes

We evaluated our method on these three datasets. The rather small CELLS dataset with only a raw size of 4 GB and the large CELEGANS and MOUSE datasets with up to 150 GB raw size.

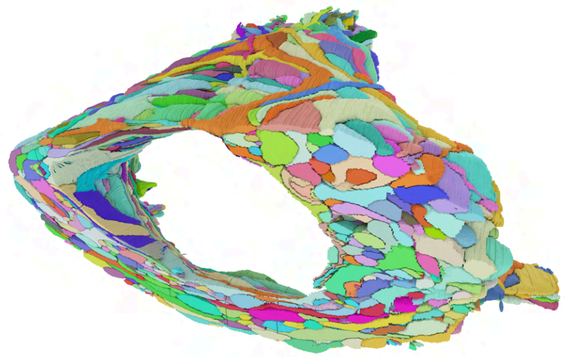
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs mem. [MB]	CR
CELLS	253.97	6.35%
C.ELEGANS	14322.37	12.7%
MOUSE	62911.29	41.95%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

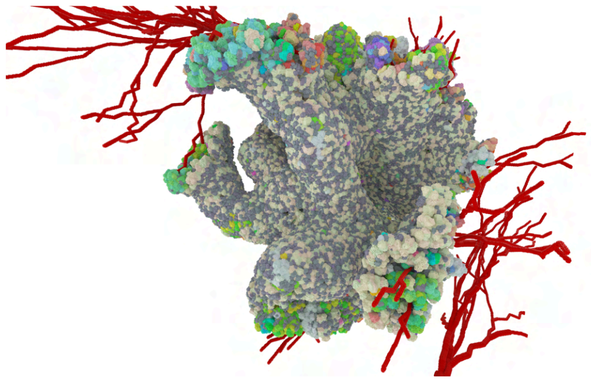
Speaker notes

Let's start with the memory and the compression rate. Remember that the reported memory is exactly the memory required for the rendering on the GPU.

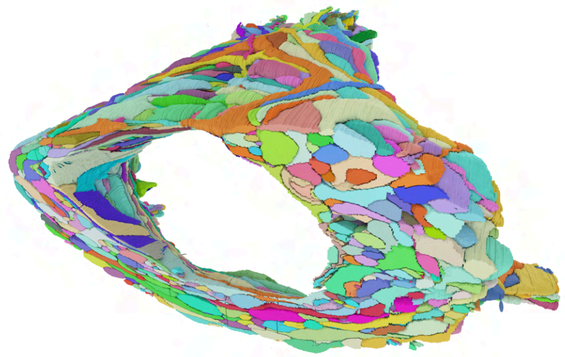
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs mem. [MB]	CR
CELLS	253.97	6.35%
C.ELEGANS	14322.37	12.7%
MOUSE	62911.29	41.95%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

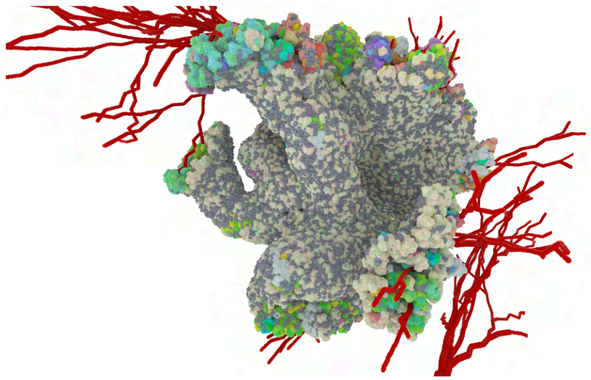
Speaker notes

The total memory consists of a comparitively small share required for the blocks and for the constructed BVH. Most memory is required for the SVO or SVDAG to encode the voxel occupancy.

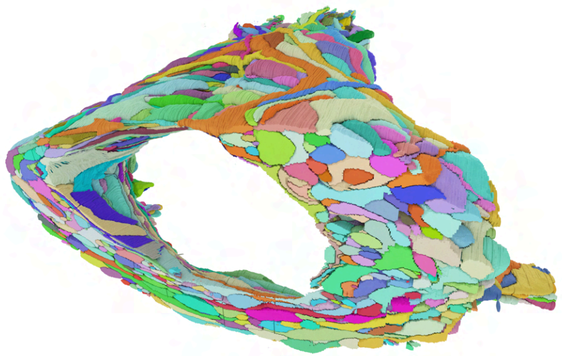
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs mem. [MB]	CR
CELLS	253.97	6.35%
C.ELEGANS	14322.37	12.7%
MOUSE	62911.29	41.95%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

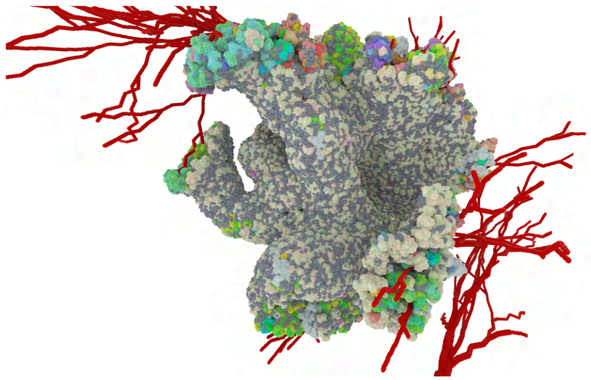
Speaker notes

With our initial idea of using SVOs in each block you can see...

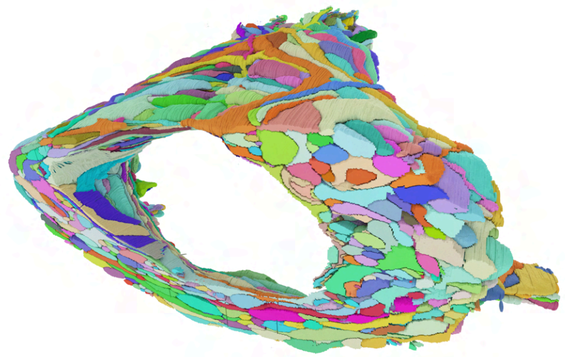
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs mem. [MB]	CR
CELLS	253.97	6.35%
C.ELEGANS	14322.37	12.7%
MOUSE	62911.29	41.95%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

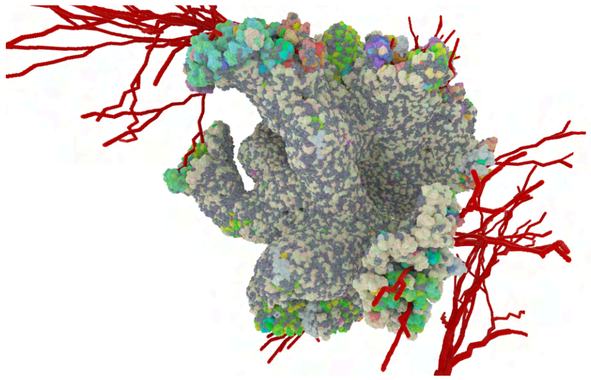
Speaker notes

...that the MOUSE cortex requires a total of 63 GB to encode the voxel occupancy which results in a poor compression rate of only 41.95%.

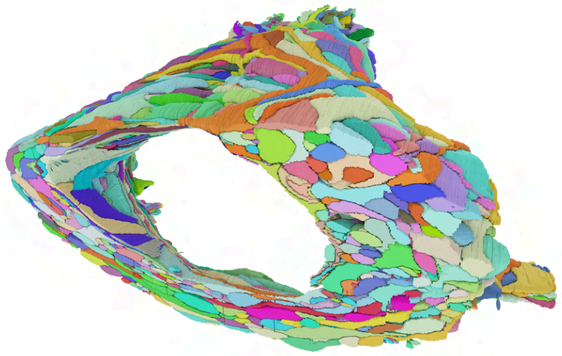
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs		blocks + BVH + SVDAG	
	mem. [MB]	CR	mem. [MB]	CR
CELLS	253.97	6.35%	86.34	2.16%
C.ELEGANS	14322.37	12.7%	2534.28	2.25%
MOUSE	62911.29	41.95%	12095.48	8.06%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

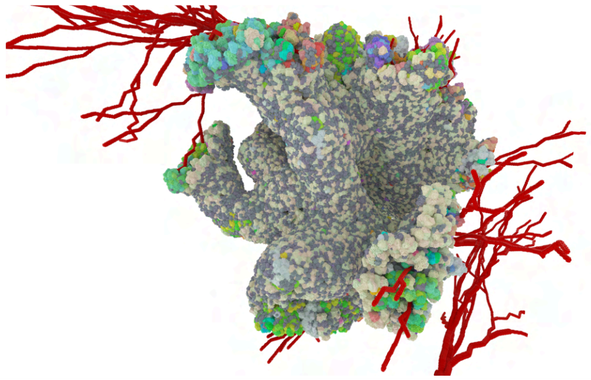
Speaker notes

When we use our large shared SVDAG instead...

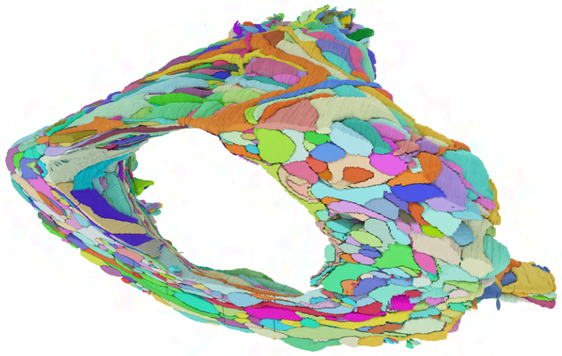
Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs		blocks + BVH + SVDAG	
	mem. [MB]	CR	mem. [MB]	CR
CELLS	253.97	6.35%	86.34	2.16%
C.ELEGANS	14322.37	12.7%	2534.28	2.25%
MOUSE	62911.29	41.95%	12095.48	8.06%

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



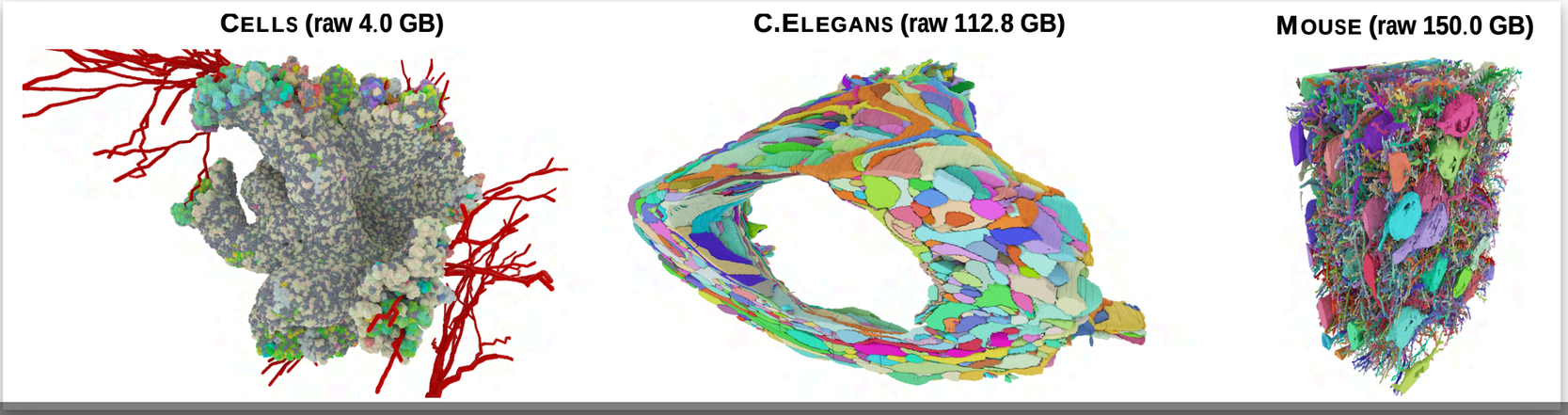
MOUSE (raw 150.0 GB)

Speaker notes

...the MOUSE cortex requires only 12 GB of memory which is an overall compression rate of 8%.

Memory (mem) and Compression Rate (CR)

	blocks + BVH + SVOs		blocks + BVH + SVDAG	
	mem. [MB]	CR	mem. [MB]	CR
CELLS	253.97	6.35%	86.34	2.16%
C.ELEGANS	14322.37	12.7%	2534.28	2.25%
MOUSE	62911.29	41.95%	12095.48	8.06%



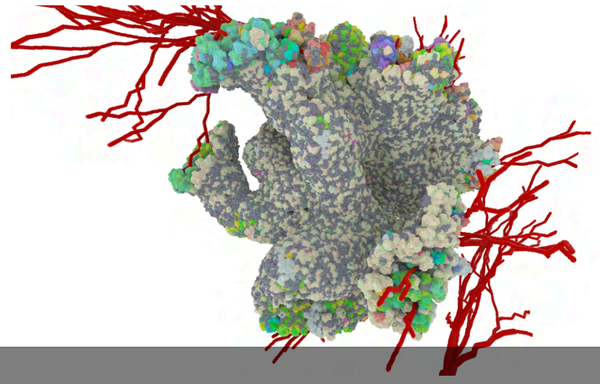
Speaker notes

And for the other two datasets we achieve even compression rates up to 2%. These volume sizes now fit into the VRAM of a modern high-end consumer GPU.

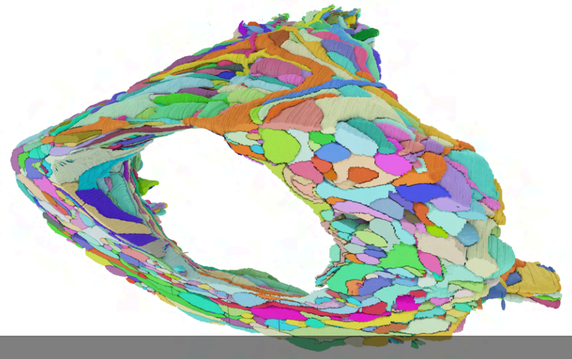
Rendering Performance

		SVOs [ms] finest LOD	SVOs [ms] coarsest LOD
1 bounce	CELLS	8.19	0.4
	C.ELEGANS	–	–
	MOUSE	–	–
32 bnc.	CELLS	11.42	0.81
	C.ELEGANS	–	–
	MOUSE	–	–

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

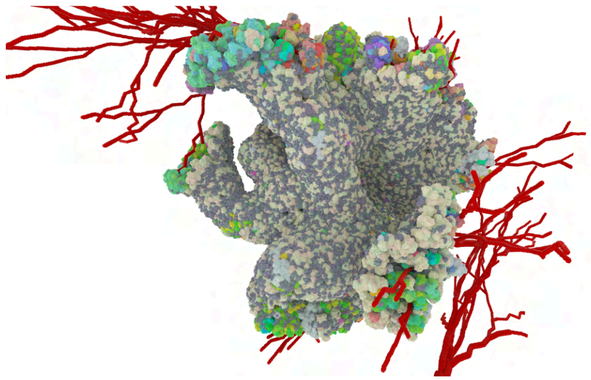
Speaker notes

Moving on to rendering performance, we show here path tracing with 1 bounce in the first row and up to 32 bounces using throughput-based Russian roulette in the second row for the three datasets. Also we report numbers either using the finest LOD, i.e. we render the individual voxels, and with the coarsest LOD, i.e. terminate traversal directly when hitting a block. Again, using our initial SVOs we can only render the CELLS dataset since the other two exceed our GPU memory.

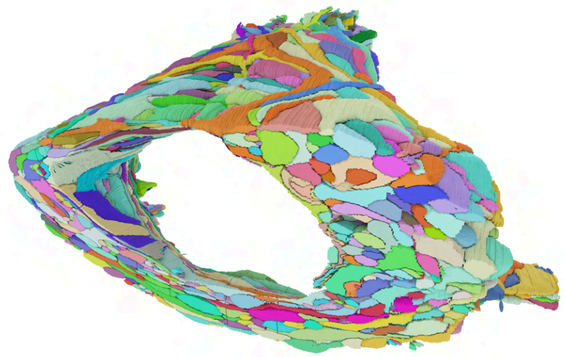
Rendering Performance

		SVOs [ms]		SVDAG [ms]	
		finest LOD	coarsest LOD	finest LOD	coarsest LOD
1 bounce	CELLS	8.19	0.4	2.95	0.51
	C.ELEGANS	–	–	5.61	0.55
	MOUSE	–	–	126.73	0.98
32 bnc.	CELLS	11.42	0.81	5.31	1.02
	C.ELEGANS	–	–	9.25	0.98
	MOUSE	–	–	189.54	2.0

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

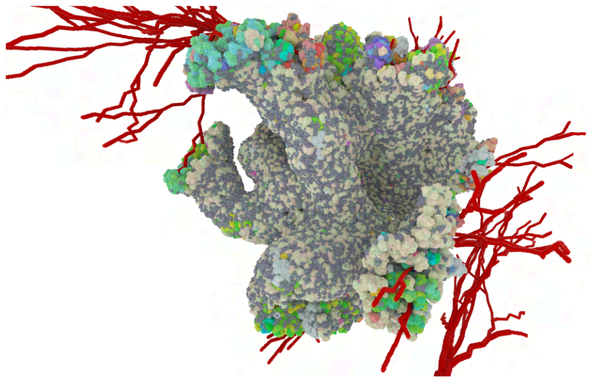
Speaker notes

When using the SVDAG...

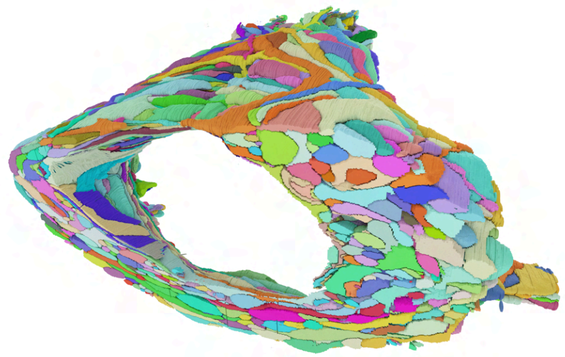
Rendering Performance

		SVOs [ms]		SVDAG [ms]	
		finest LOD	coarsest LOD	finest LOD	coarsest LOD
1 bounce	CELLS	8.19	0.4	2.95	0.51
	C.ELEGANS	–	–	5.61	0.55
	MOUSE	–	–	126.73	0.98
32 bnc.	CELLS	11.42	0.81	5.31	1.02
	C.ELEGANS	–	–	9.25	0.98
	MOUSE	–	–	189.54	2.0

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

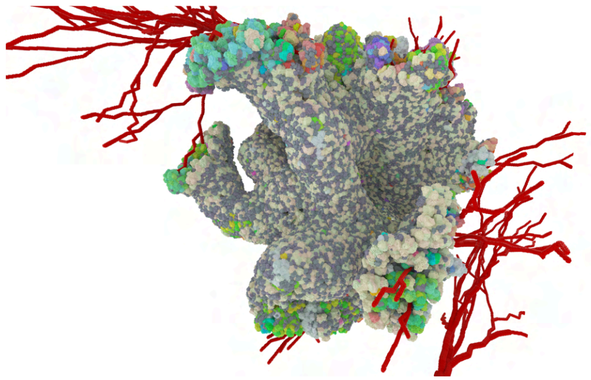
Speaker notes

...you can see that the frametime is reduced significantly.

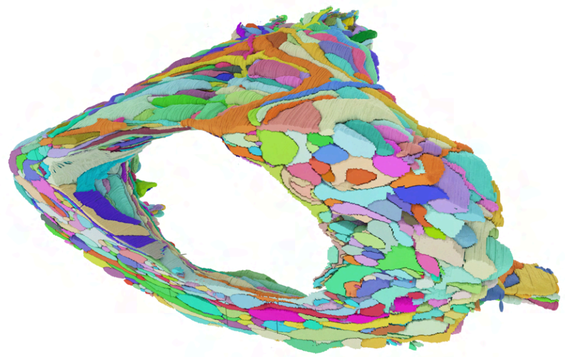
Rendering Performance

		SVOs [ms]		SVDAG [ms]	
		finest LOD	coarsest LOD	finest LOD	coarsest LOD
1 bounce	CELLS	8.19	0.4	2.95	0.51
	C.ELEGANS	–	–	5.61	0.55
	MOUSE	–	–	126.73	0.98
32 bnc.	CELLS	11.42	0.81	5.31	1.02
	C.ELEGANS	–	–	9.25	0.98
	MOUSE	–	–	189.54	2.0

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



MOUSE (raw 150.0 GB)

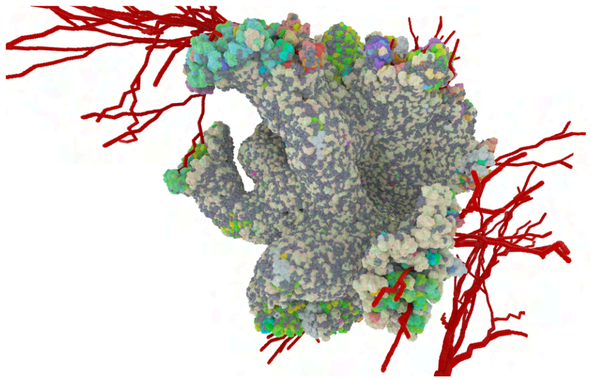
Speaker notes

The CELLS and CELEGANS datasets can be fully path traced even with up to 32 bounces with over 60 FPS on this lossless representation.

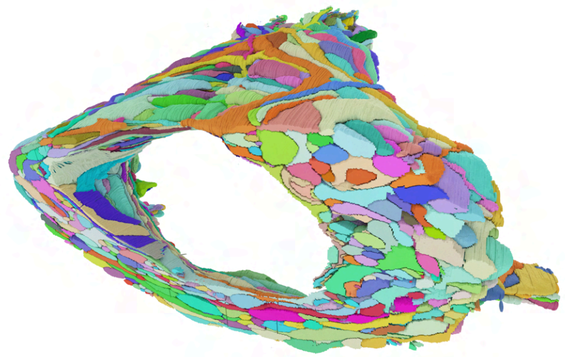
Rendering Performance

		SVOs [ms]		SVDAG [ms]	
		finest LOD	coarsest LOD	finest LOD	coarsest LOD
1 bounce	CELLS	8.19	0.4	2.95	0.51
	C.ELEGANS	–	–	5.61	0.55
	MOUSE	–	–	126.73	0.98
32 bnc.	CELLS	11.42	0.81	5.31	1.02
	C.ELEGANS	–	–	9.25	0.98
	MOUSE	–	–	189.54	2.0

CELLS (raw 4.0 GB)



C.ELEGANS (raw 112.8 GB)



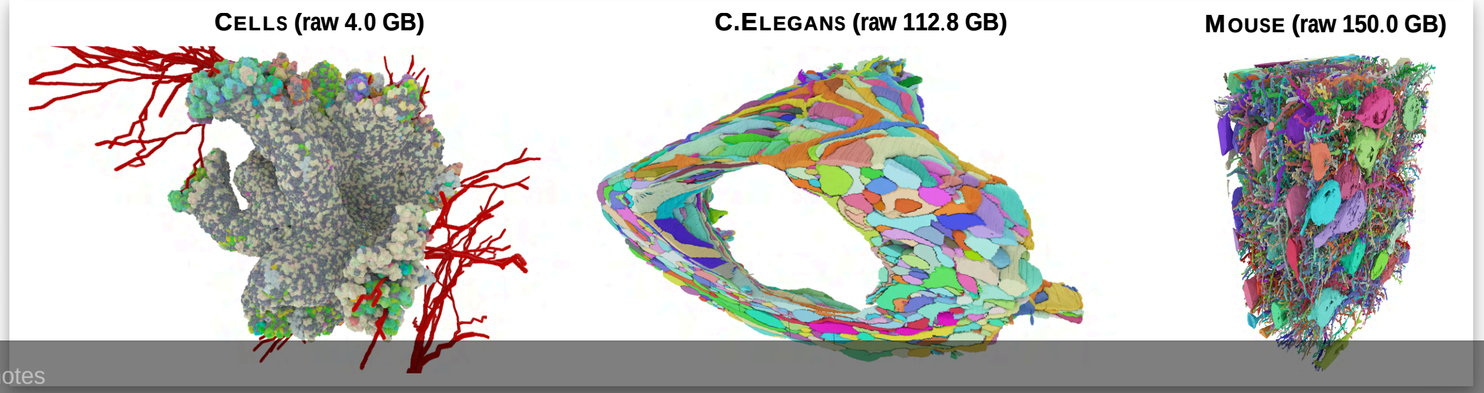
MOUSE (raw 150.0 GB)

Speaker notes

And still, the much more sparse and harder to ray trace MOUSE dataset can still be interactively path-traced to some degree.

Rendering Performance

		SVOs [ms]		SVDAG [ms]	
		finest LOD	coarsest LOD	finest LOD	coarsest LOD
1 bounce	CELLS	8.19	0.4	2.95	0.51
	C.ELEGANS	–	–	5.61	0.55
	MOUSE	–	–	126.73	0.98
32 bnc.	CELLS	11.42	0.81	5.31	1.02
	C.ELEGANS	–	–	9.25	0.98
	MOUSE	–	–	189.54	2.0



Speaker notes

By using dynamic LOD or by always using the coarsest LOD while the camera is moving, we can ensure exploration of the MOUSE dataset with more than 60 FPS. Once the camera is static, the finest LOD can be used to render the converged image. On the right side you can see the MOUSE cortex with coarsest LOD and with the finest lossless LOD. For these large datasets, from further away, you can barely see a differences. This means that using this coarser LOD during exploration is indeed a viable option.

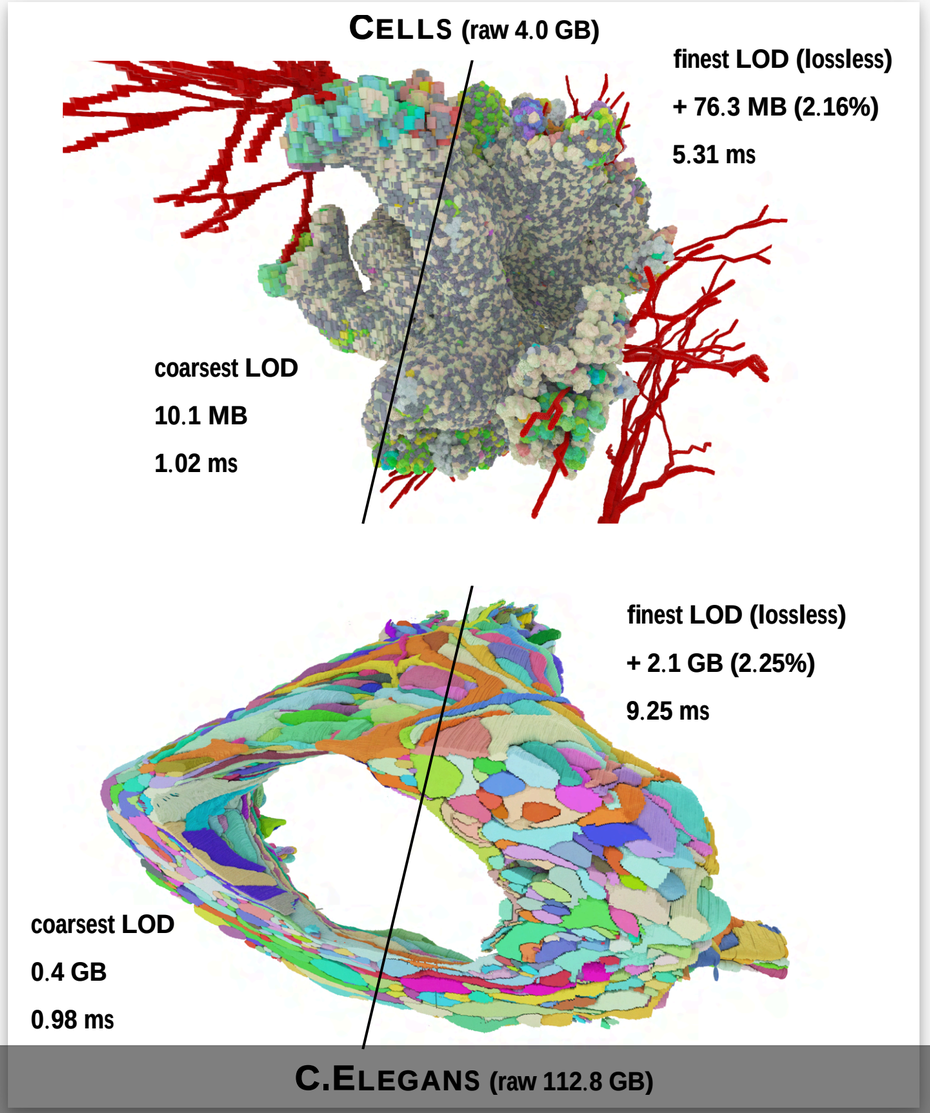
Conclusion

Speaker notes

In conclusion, we are able to unify this process of exploring and rendering large segmentation volumes into a single framework...

Conclusion

- introduced a
  - lossless SVDAG compression method

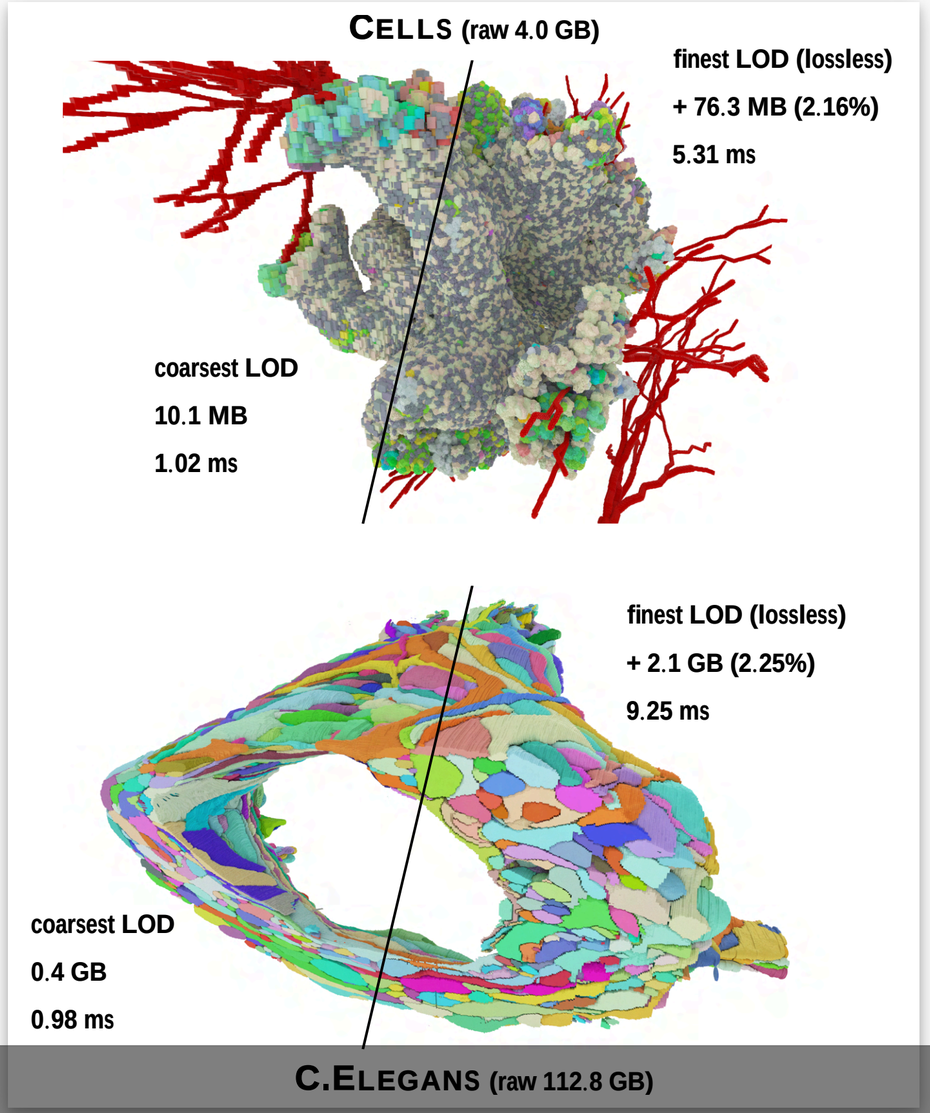


Speaker notes

... by introducing a lossless SVDAG compression method...

Conclusion

- introduced a
  - lossless SVDAG compression method
  - and a hardware-accelerated path tracing framework

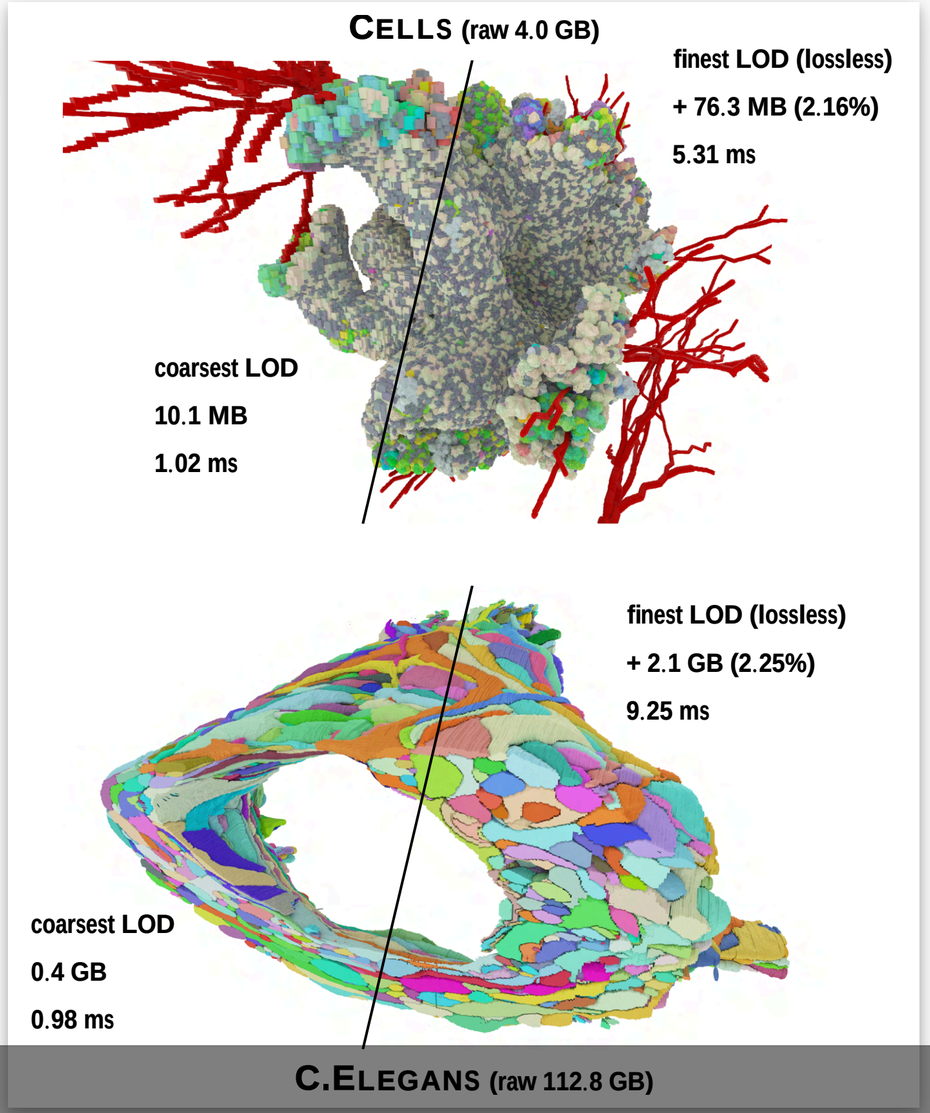


Speaker notes

... and a hardware-accelerated path tracing framework...

Conclusion

- introduced a
  - lossless SVDAG compression method
  - and a hardware-accelerated path tracing framework
- for interactive rendering of segmentation volumes



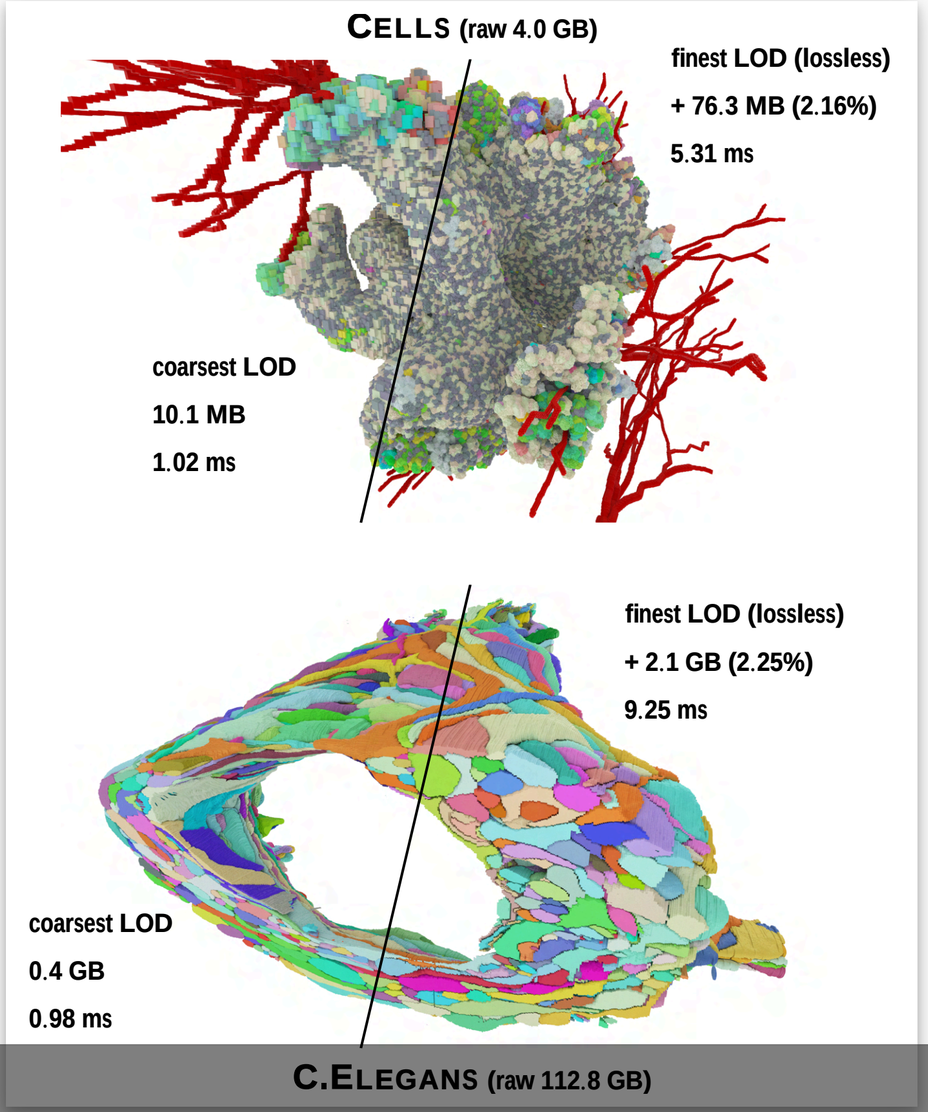
Speaker notes

that allows for interactive rendering of segmentation volumes.

Conclusion

- introduced a
  - lossless SVDAG compression method
  - and a hardware-accelerated path tracing framework
- for interactive rendering of segmentation volumes

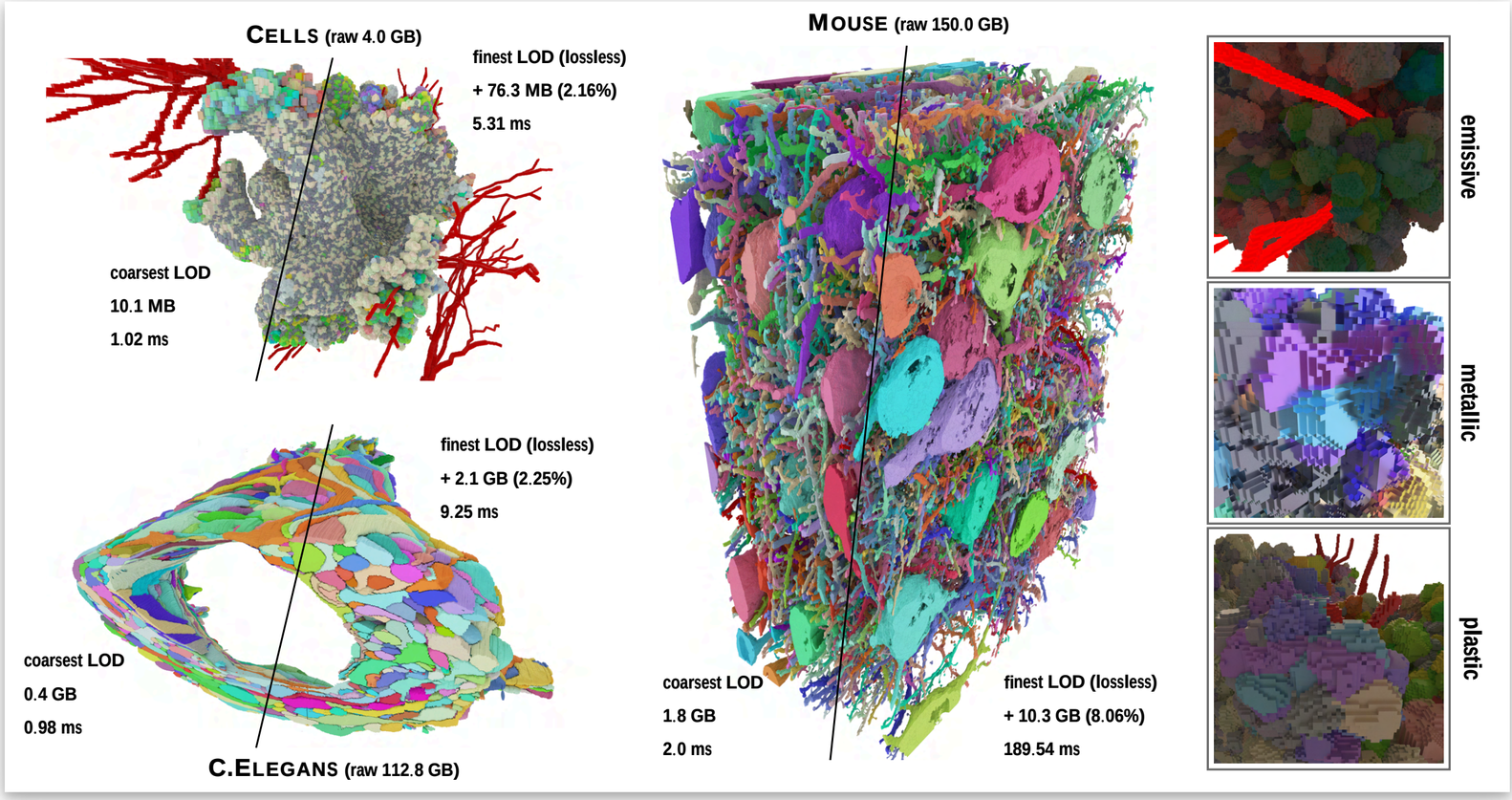
- strong compression and rendering performance



Speaker notes

We thereby achieve strong compression rates and high rendering performance.

Thank you!



Code available on GitHub:

https://github.com/MircoWerner/SegmentationVolumeCompression

Speaker notes

Our code is publicly available on GitHub. With that said, thank you for your attention!

References

- [Rosenbauer*2020]
  - ROSENBAUER, J., BERGHOFF, M., and SCHUG, A. “Emerging Tumor Development by Simulating Single-cell Events”. bioRxiv (2020). DOI: 10.1101/2020.08.24.264150
- [Witvliet*2021]
  - WITVLIET, D., MULCAHY, B., MITCHELL, J. K., et al. “Connectomes across development reveal principles of brain maturation”. Nature 596.7871 (2021), 257-261. DOI: 10.1038/s41586-021-03778-8
- [Motta*2019]
  - MOTTA, A., BERNING, M., BOERGENS, K. M., et al. “Dense connectomic reconstruction in layer 4 of the somatosensory cortex”. Science 366.6469 (2019), eaay3134. DOI: 10.1126/science.aay3134
- [Velicky*2023]
  - VELICKY, P., MIGUEL, E., MICHALSKA, J.M. et al. Dense 4D nanoscale reconstruction of living brain tissue. Nat Methods 20, 1256-1265 (2023). https://doi.org/10.1038/s41592-023-01936-6
- [Laine*2010]
  - LAINE, S. and KARRAS, T. “Efficient sparse voxel octrees”. Proc. ACM SIGGRAPH Symposium on Interact. 3D Graph. and Games. New York, NY, USA: ACM, 2010, 55-63. DOI: 10.1145/1730804.1730814
- [Kämpe*2013]
  - KÄMPE, V., SINTORN, E., and ASSARSSON, U. “High resolution sparse voxel DAGs”. ACM Transactions on Graphics 32.4 (July 2013). DOI: 10.1145/2461912.2462024

Appendix

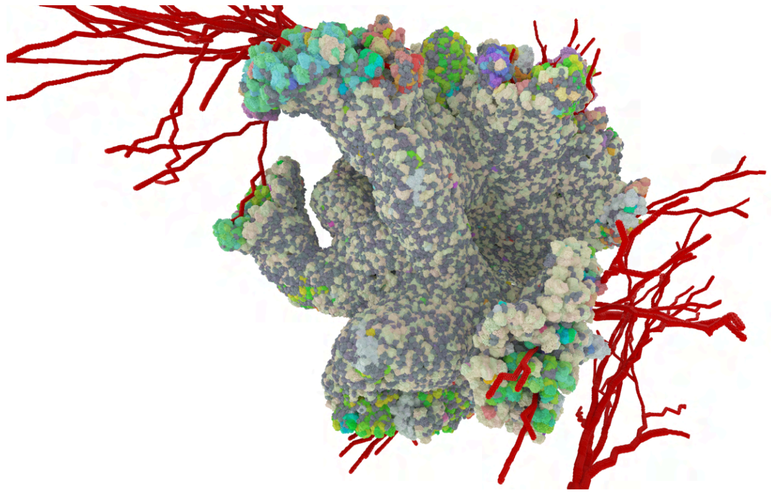
Speaker notes

...

SVDAG Node Memory Layout

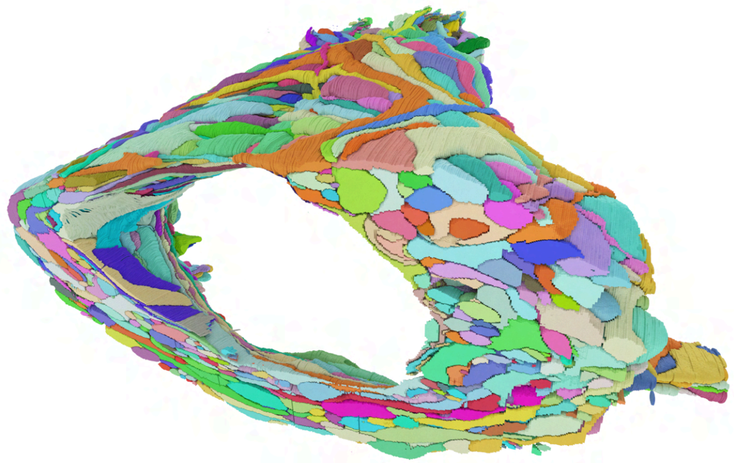
Datasets

CELLS (raw 4.0 GB)



$1000 \times 1000 \times 1000 = 1.0 \cdot 10^9$
15.39% solid
118251 labels

C.ELEGANS (raw 112.8 GB)



$661 \times 11044 \times 15448 = 112.77 \cdot 10^9$
5.45% solid
234 labels

MOUSE (raw 150.0 GB)

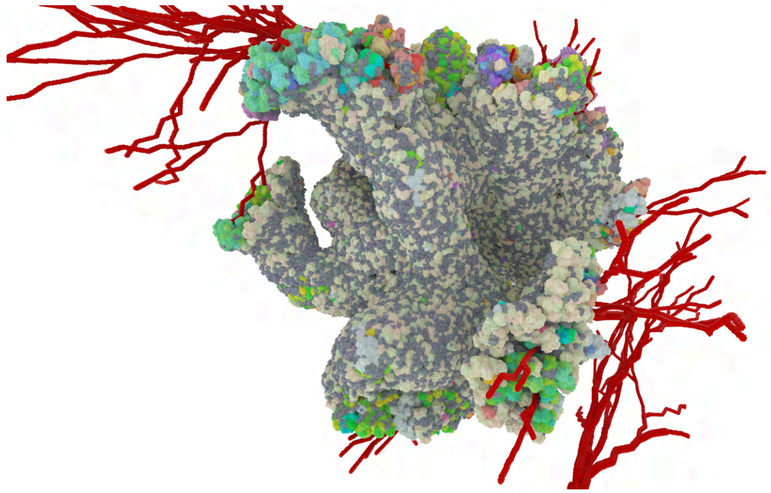
$5446 \times 8381 \times 3286 = 149.98 \cdot 10^9$
15.57% solid
96 labels

Speaker notes

We evaluated our method on these three datasets. The rather small CELLS dataset with only a raw size of 4 GB and the large CELEGANS and MOUSE datasets with up to 150 GB raw size.

Datasets

CELLS (raw 4.0 GB)

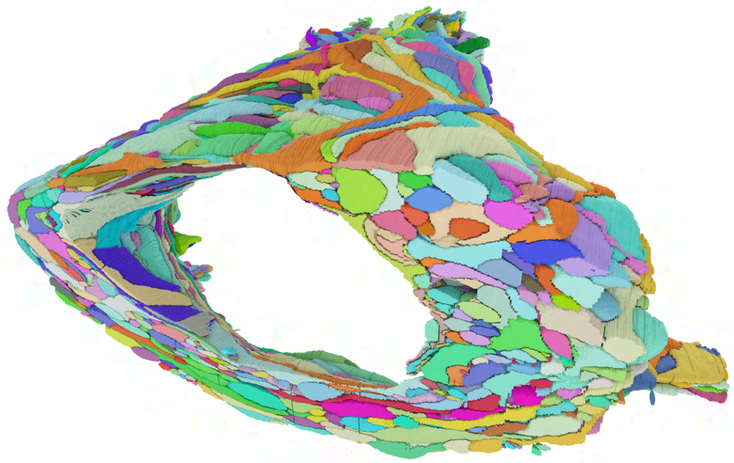


$1000 \times 1000 \times 1000 = 1.0 \cdot 10^9$

15.39% solid

118251 labels

C.ELEGANS (raw 112.8 GB)



$661 \times 11044 \times 15448 = 112.77 \cdot 10^9$

5.45% solid

234 labels

MOUSE (raw 150.0 GB)

$5446 \times 8381 \times 3286 = 149.98 \cdot 10^9$

15.57% solid

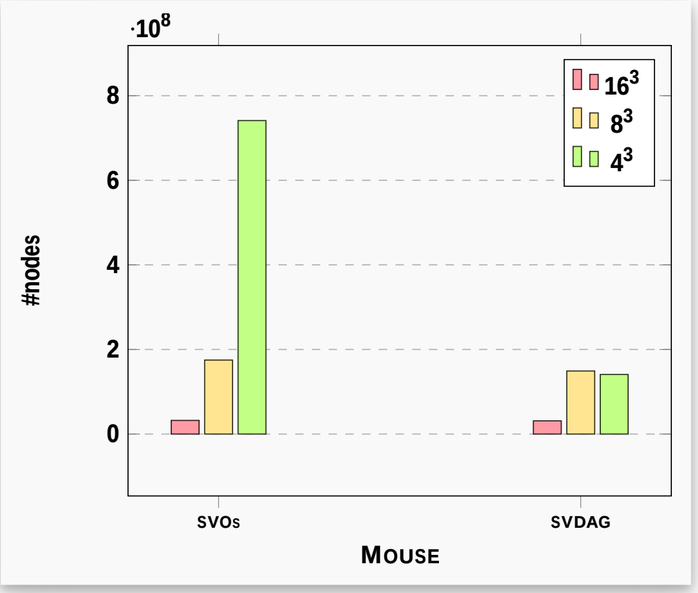
96 labels

Speaker notes

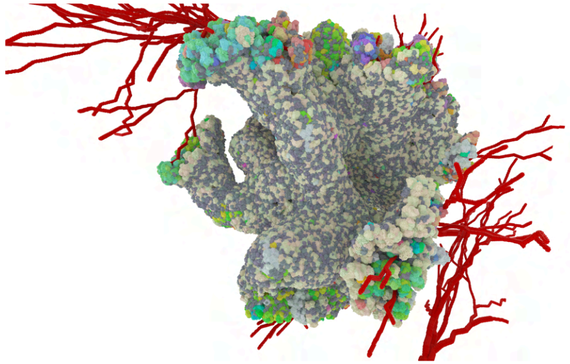
As you can see in the bottom row, the CELLS dataset contains more than 100k labels while the other two datasets contain only a few hundreds of labels. However, keep in mind, that since we decouple labels and voxel occupancy, our compression of the voxels works and scales independently from the actual number of labels.

Memory (mem) and Compression Rate (CR)

	AABBs + BVH + mem. [MB]	SVOs mem. [MB]	SVOs CR	SVDAG mem. [MB]	SVDAG CR
CELLS	5.63 + 4.42	+ 243.92	6.35%	+ 76.29	2.16%
C.ELEGANS	248.13 + 186.24	+ 13888.0	12.7%	+ 2099.91	2.25%
MOUSE	1032.11 + 786.44	+ 61092.74	41.95%	+ 10276.93	8.06%



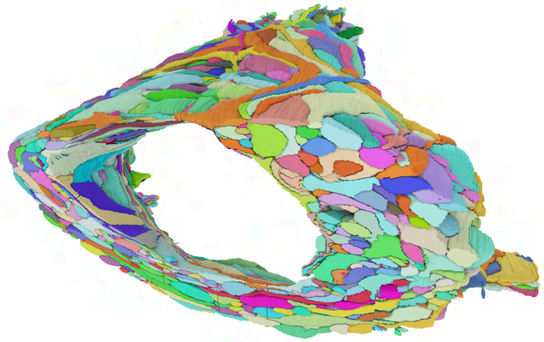
CELLS (raw 4.0 GB)



$1000 \times 1000 \times 1000 = 1.0 \cdot 10^9$

15.39% solid

C.ELEGANS (raw 112.8 GB)



$661 \times 11044 \times 15448 = 112.77 \cdot 10^9$

5.45% solid

MOUSE (raw 150.0 GB)

$5446 \times 8381 \times 3286 = 149.98 \cdot 10^9$

15.57% solid

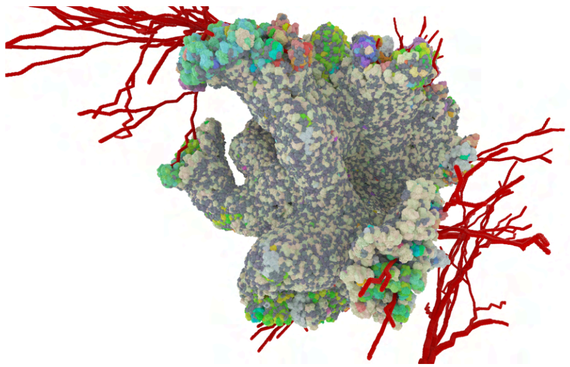
Speaker notes

On the right side here, you can see the number of nodes in each of the levels, 16^3 8^3 4^3, of the SVOs for the SVDAG respectively. Because of similar topology we can reuse a large fraction of the nodes corresponding to the 4 by 4 by 4 regions which explains the strong compression rates.

Rendering Performance [ms] using finest (coarsest) LOD

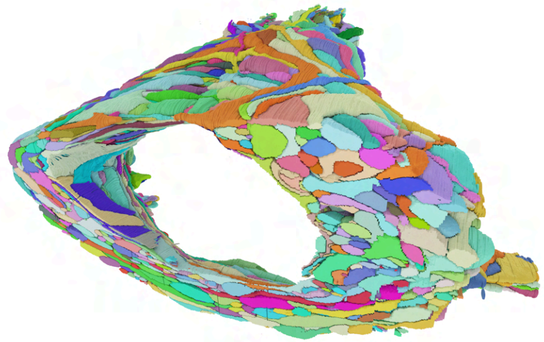
		SVOs	w/ occupancy field SVDAG	w/o occupancy field SVDAG
1 bounce	CELLS	8.19 (0.4)	2.95 (0.51)	5.87 (0.42)
	C.ELEGANS	–	5.61 (0.55)	12.75 (0.47)
	MOUSE	–	126.73 (0.98)	134.72 (0.78)
32 bnc.	CELLS	11.42 (0.81)	5.31 (1.02)	8.66 (0.85)
	C.ELEGANS	–	9.25 (0.98)	18.72 (0.83)
	MOUSE	–	189.54 (2.0)	203.65 (1.67)

CELLS (raw 4.0 GB)



$1000 \times 1000 \times 1000 = 1.0 \cdot 10^9$
15.39% solid
118251 labels

C.ELEGANS (raw 112.8 GB)



$661 \times 11044 \times 15448 = 112.77 \cdot 10^9$
5.45% solid
234 labels

MOUSE (raw 150.0 GB)

$5446 \times 8381 \times 3286 = 149.98 \cdot 10^9$
15.57% solid
96 labels

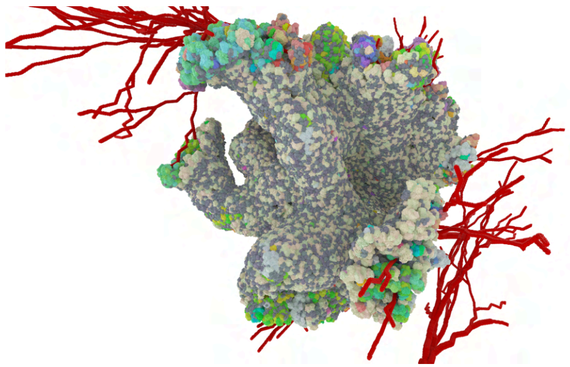
Speaker notes

Finally, let's compare the SVDAG using our proposed occupancy fields with the SVDAG without this optimization.

Rendering Performance [ms] using finest (coarsest) LOD

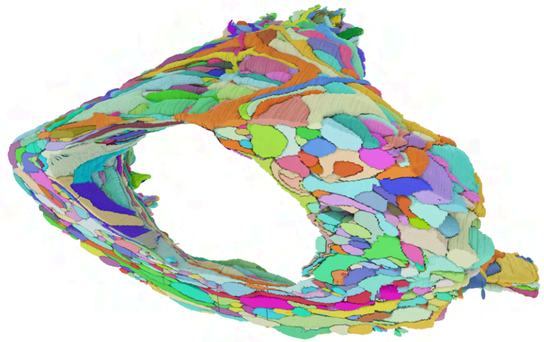
		SVOs	w/ occupancy field SVDAG	w/o occupancy field SVDAG
1 bounce	CELLS	8.19 (0.4)	2.95 (0.51)	5.87 (0.42)
	C.ELEGANS	–	5.61 (0.55)	12.75 (0.47)
	MOUSE	–	126.73 (0.98)	134.72 (0.78)
32 bnc.	CELLS	11.42 (0.81)	5.31 (1.02)	8.66 (0.85)
	C.ELEGANS	–	9.25 (0.98)	18.72 (0.83)
	MOUSE	–	189.54 (2.0)	203.65 (1.67)

CELLS (raw 4.0 GB)



$1000 \times 1000 \times 1000 = 1.0 \cdot 10^9$
15.39% solid
118251 labels

C.ELEGANS (raw 112.8 GB)



$661 \times 11044 \times 15448 = 112.77 \cdot 10^9$
5.45% solid
234 labels

MOUSE (raw 150.0 GB)

$5446 \times 8381 \times 3286 = 149.98 \cdot 10^9$
15.57% solid
96 labels

Speaker notes

Notice, that our occupancy fields signifintly reduce frametime by reducing the number of slow and random VRAM accesses.

Compression Timings

	CELLS	C.ELEGANS	MOUSE
SVOS (AABB)			
CPU [s]	5.97	905.26	4127.16
SVDAGS (TYPE)			
CPU [s]	2.24	117.92	683.45
SVDAG (SHARED)			
CPU [ms]	408.49	33089.85	147721.0
GPU [ms]	101.96	2865.33	–

Table 5: Timings for each step to compress segmentation volumes: Computing AABBs and their SVOs from lists of solid voxel positions per label, merging them to SVDAGS (TYPE), and subsequently merging these partial SVDAGs to one SVDAG (SHARED). We provide a GPU implementation of the last step for fast merging of visualized types during rendering start-up or runtime.

CSGV [Piochowiak*2023] (memory)

	blocks + BVH + SVOs		blocks + BVH + SVDAG	
	mem. [MB]	CR	mem. [MB]	CR
CELLS	253.97	6.35%	86.34	2.16%
C.ELEGANS	14322.37	12.7%	2534.28	2.25%
MOUSE	62911.29	41.95%	12095.48	8.06%

CSGV mem. [MB]		CSGV CR	
15.46	(+544)	0.39%	(13.99%)
736.981	(+768)	0.65%	(1.33%)
1910.17	(+768)	1.27%	(1.79%)

Table 2: Memory and compression rate of CSGV [PD24]. Theoretical compression rate when including a GPU cache size to decompress visible regions to fitting levels of detail during rendering in ().

CSGV [Piochowiak*2023] (rendering performance)

		SVOs [ms]		SVDAG [ms]		CSGV
		finest LOD	coarsest LOD	finest LOD	coarsest LOD	
1 bounce	CELLS	8.19	0.4	2.95	0.51	23.21 (2.28)
	C.ELEGANS	–	–	5.61	0.55	87.18 (16.56)
	MOUSE	–	–	126.73	0.98	124.89 (8.47)
32 bnc.	CELLS	11.42	0.81	5.31	1.02	25.64 (2.29)
	C.ELEGANS	–	–	9.25	0.98	97.63 (17.06)
	MOUSE	–	–	189.54	2.0	136.16 (10.55)