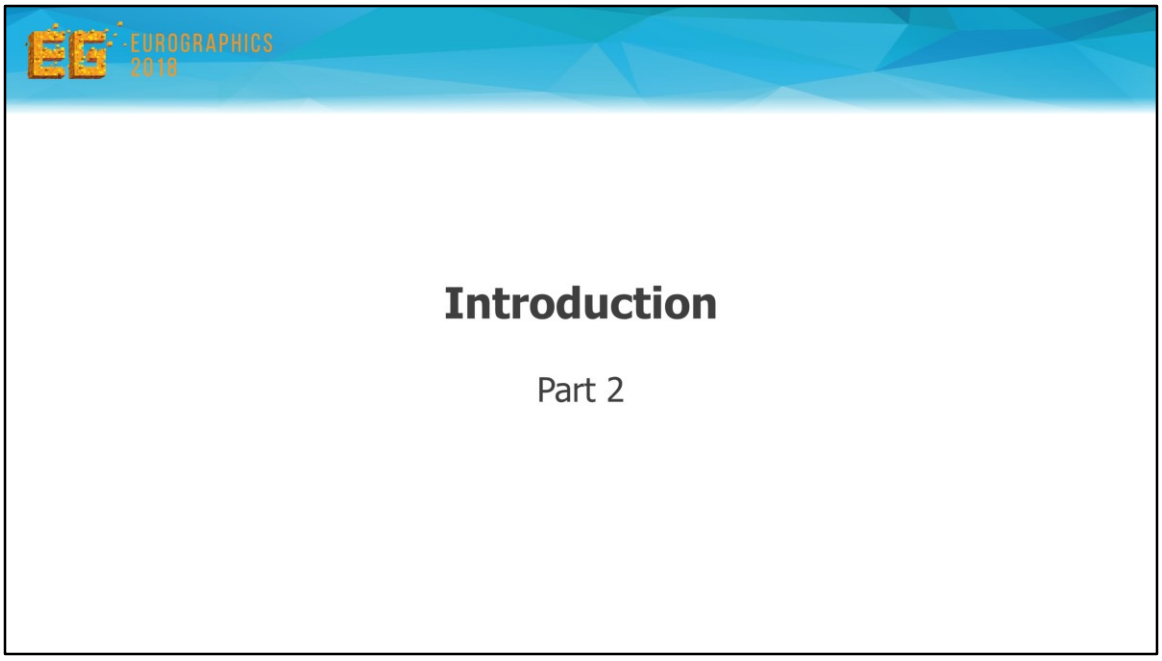
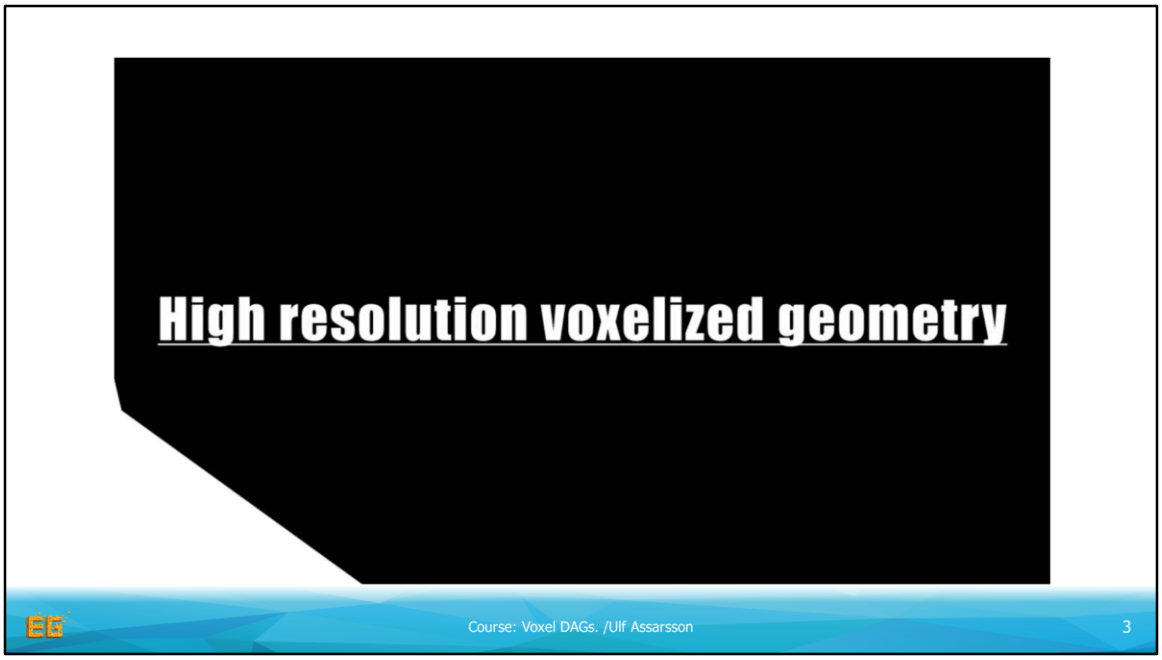

EUROGRAPHICS
2018
Introduction
Part 2

The standard DAG compression of geometry

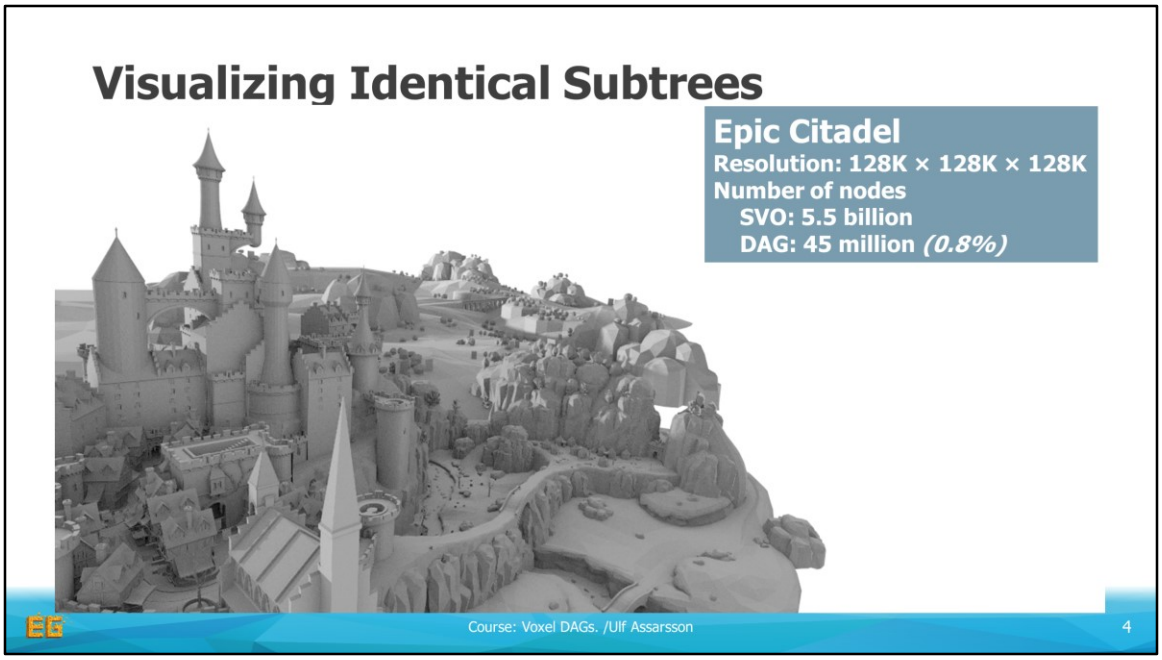
- I am:
  Ulf Assarsson,
  Chalmers University of Technology,
  Goteborg, Sweden.

Compression performance, why/when and where we get compression.
Good morning everyone and welcome to the course. My name is… I'm a prof. at.. Our research group started working on voxels and DAGs almost 8 years ago.
I will start with a movie we showed at fast forward Siggraph 2013.

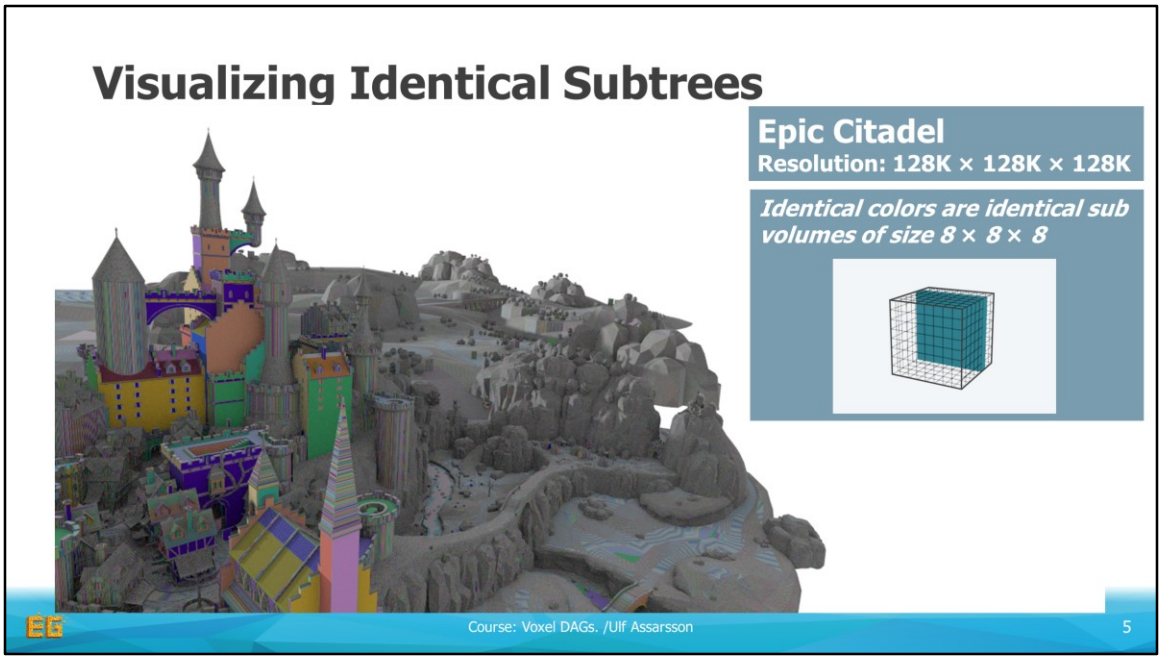


Fast forward, siggraph 2013.

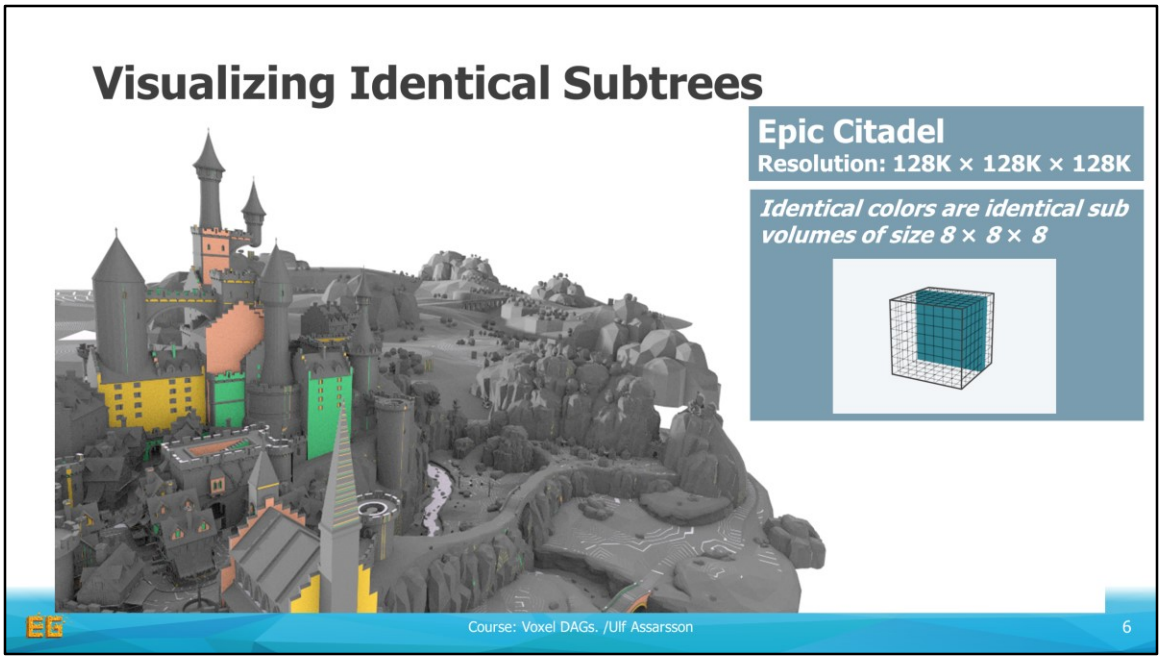
I will explain why there is the amount of subgraph-merging opportunities, and why and how we get the compression that we get.
And also talk about dynamic scenes and DAGs for free viewpoint video.



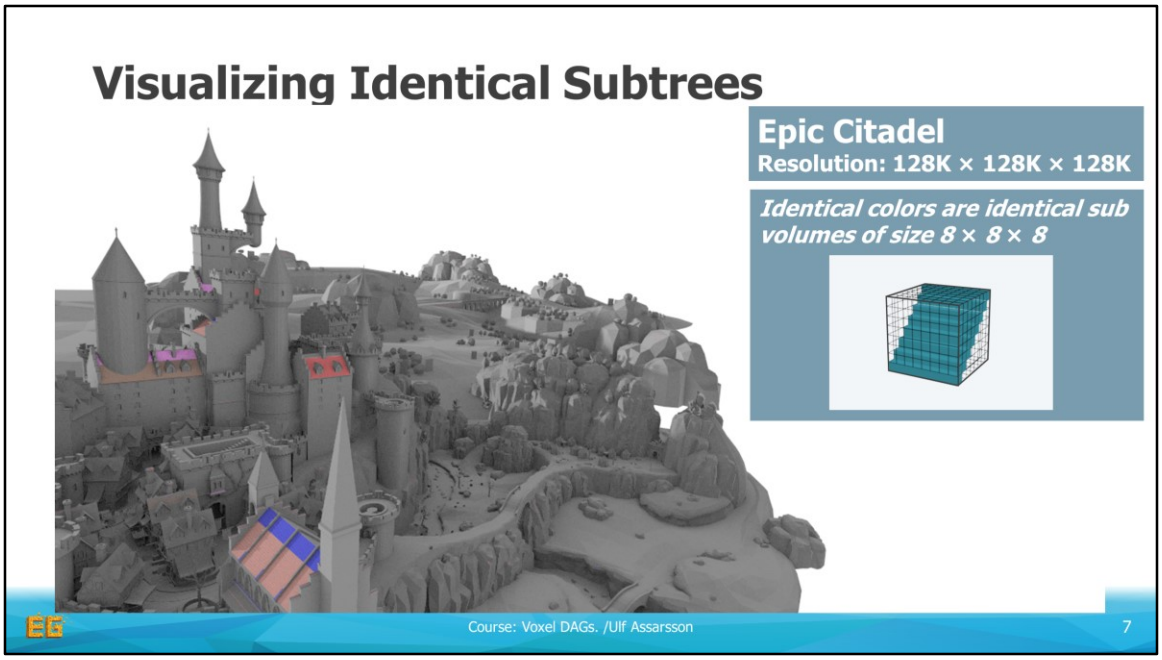
This is the epic citadel scene. SVO takes… DAG takes… so ~100x fewer nodes.



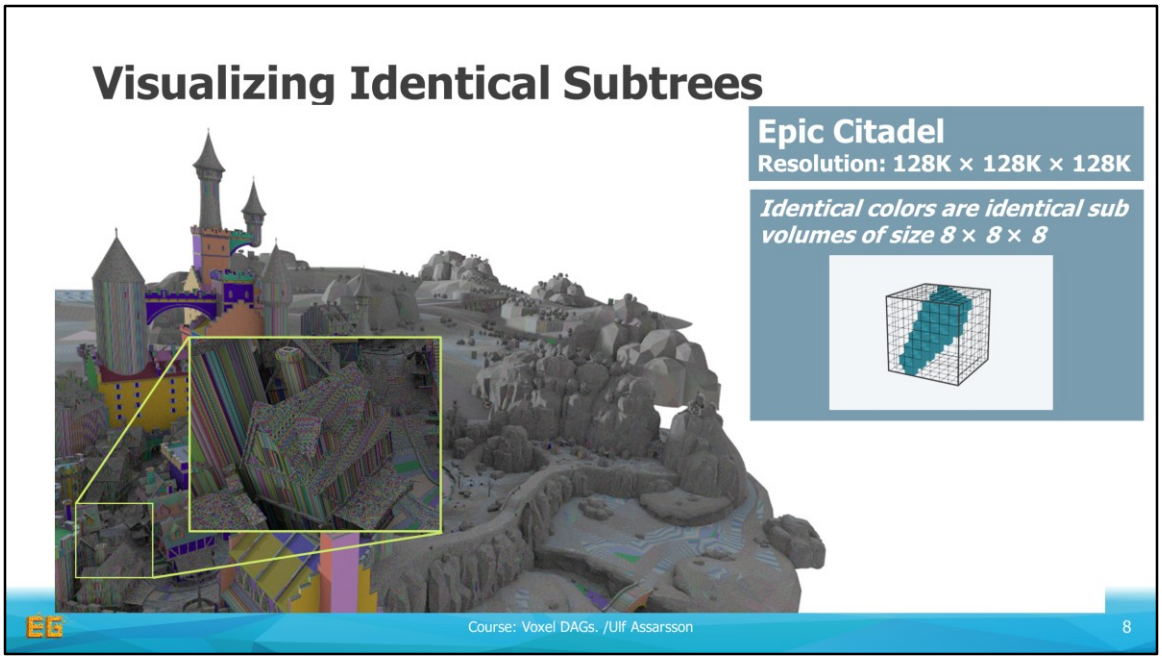
Here, we have colored some equal subvolumes of 8-cubed resolution. So the yellow surfaces are all the same 8^3-grid. The green use another identical 8^3 subvolume, etc.



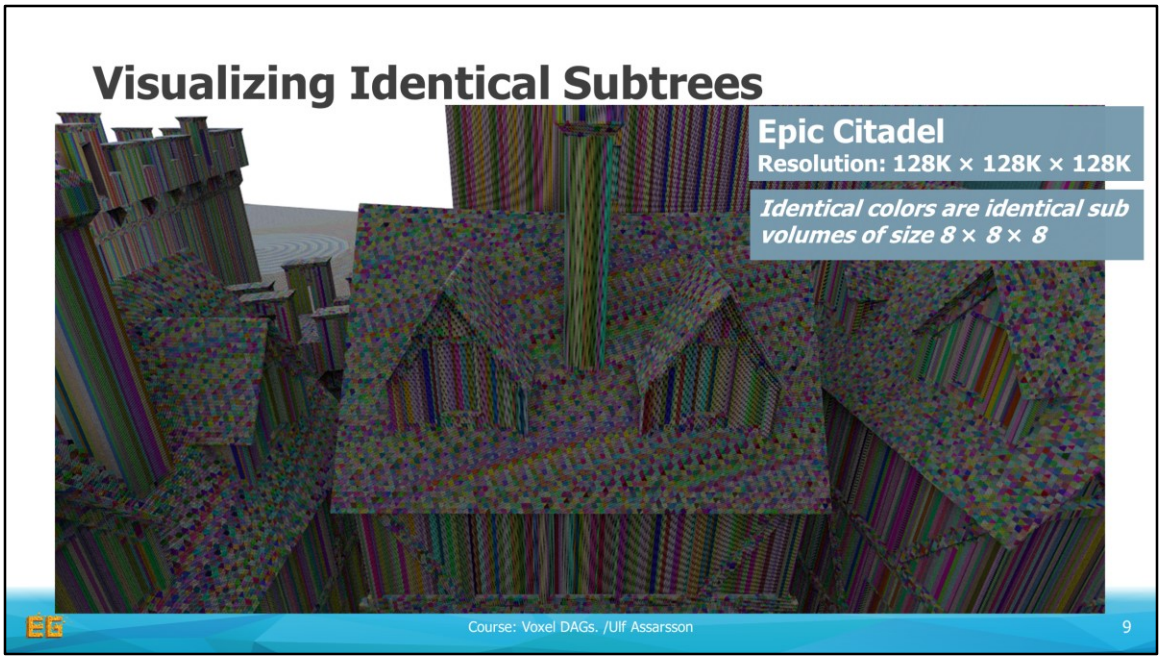
As expected, we can easily see that large, **axis-aligned** surfaces **compress very well**. The highlighted areas are described by only four different sub-trees.



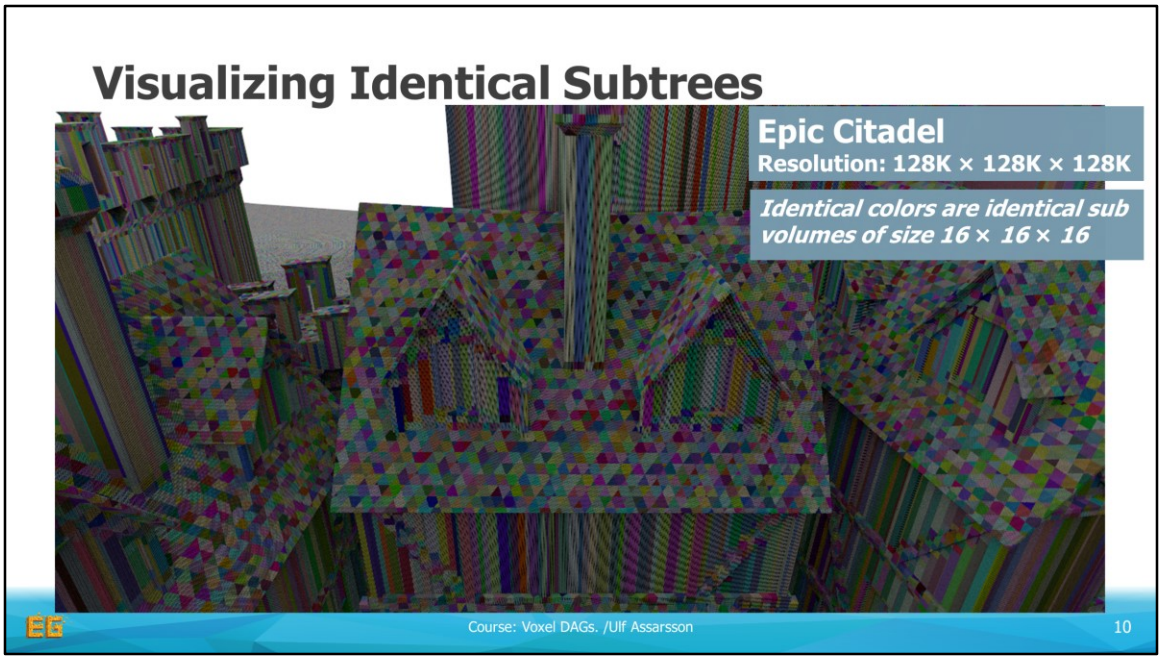
It is also clear that surfaces that **slope in only one dimension** compress very well.



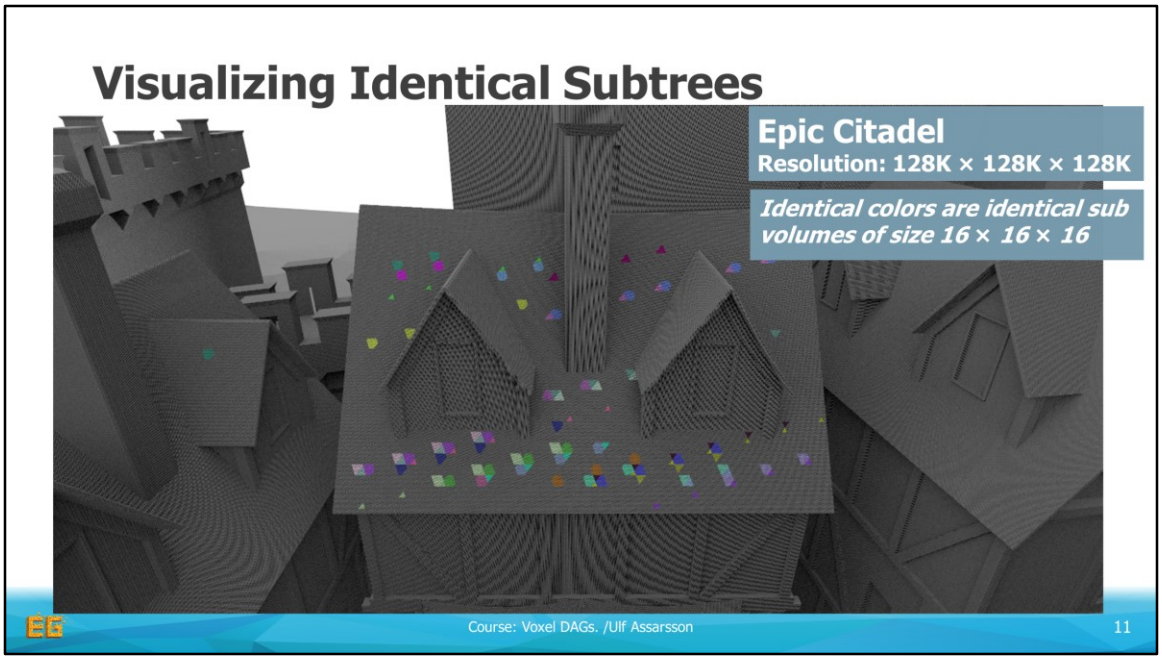
It is **less obvious** that surfaces that are **non axis-aligned** will compress.



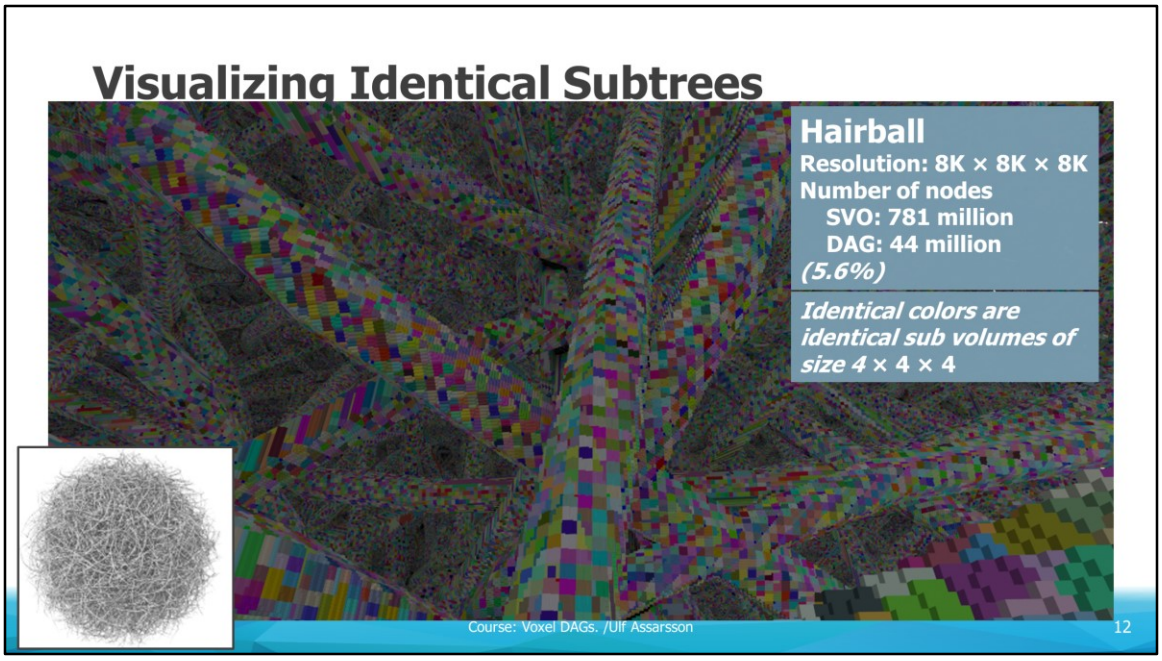
However, **zooming in** on such a surface, it is **fairly easy to see** that the roof is composed of a **repeated pattern of just a few nodes**.



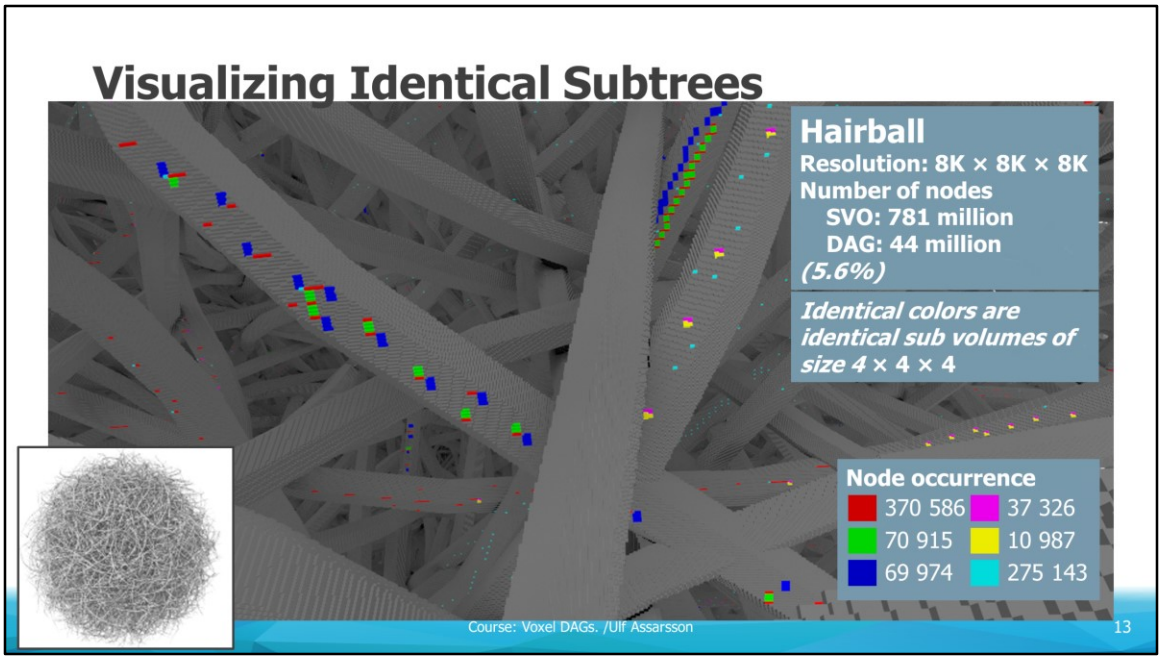
If we look at subtrees of size **16x16x16,** the **pattern** is **less obvious**…



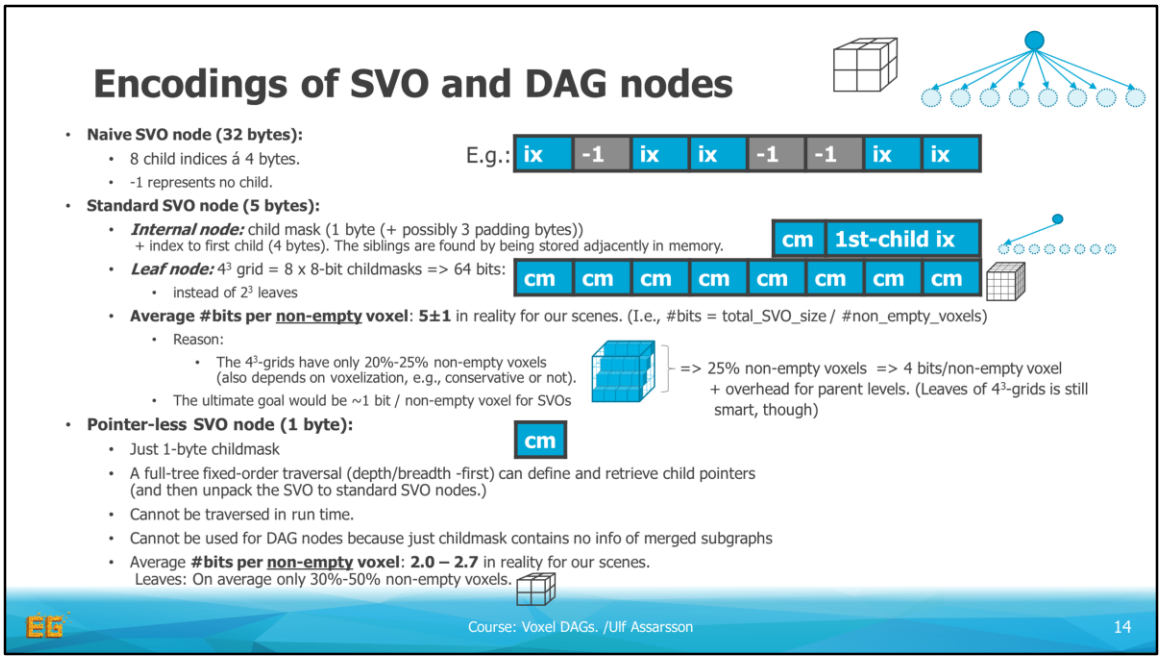
But **highlighting a only a few nodes** again, we can see that non axis-aligned surfaces indeed will compress well.



Here, we look at an **extreme close-up** of a **hair-ball**, and see **no obvious patterns**…



But **highlighting a few nodes** make it clear that the same subtrees (4x4x4) appear again, scattered over the geometry.
As an example, you can see that the red nodes in the image actually reappear 370.000 times in the scene.
However, the DAG nodes are larger than SVO nodes, so the compression in number of bytes is lower than the compression in number of nodes. We will now see why that is.



Naive SVO coding: may want to use only for debug purposes.
For typical/standard SVO-node encoding, we often land at 5-6 bits per voxel - since we use 4-byte indices!
For pointer-less SVO:s and general scenes, we cannot hope for better than 2 bits per voxel if we use 8-bit leafs and those are only filled by 50% bits of 1's.

Pointer-less good for storing on disk. But for any efficient real-time traversal, they need to be unpacked, using for instance standard SVO nodes.

Citadel 32K^3: 12,6567 set voxels per 64-bit leaf => 20%, Average 5.3 bits/voxel
Gallery 4K^3: 15,3302 set voxels per 64-bit leaf => 24%, Average 4.27 bits/voxels
Sponza: 2^3-leaves: 48% set voxels.
Gallery: 2^3-leaves: 46%,
Citadel: 2^3-leaves: 42%.
Campus: 2^3-leaves: 33%

Encodings of SVO and DAG nodes

- **Naive DAG node (32 bytes/node):**
  - 8 child indices á 4 bytes.
  - -1 represents no child. E.g.,:

ix	-1	ix	ix	-1	-1	ix	ix

- **Standard DAG node (~22 bytes/node):**
  - ***Internal node:*** Dynamic size.
    Child mask (1 byte + 3 padding bytes)
    + only indices to existing children (á 4 bytes):

cm	ix	ix	ix	ix

  - ***Leaf node:*** 4^3 grid = 8 x 8-bit childmasks:
  - Expected average node size:
    - 1 cm + 4.5 child indices á 4 bytes = **22** bytes.
    - Without padding: **19** bytes

cm	cm	cm	cm	cm	cm	cm	cm

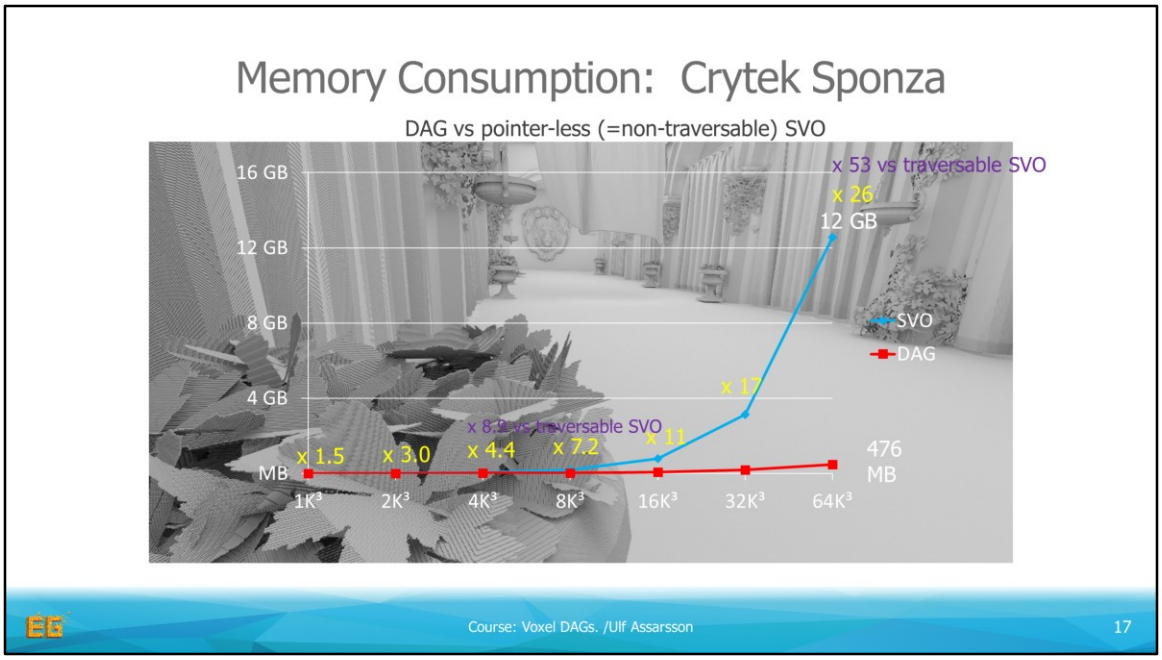
- **Pointer-compressed DAG node (~13-17 bytes/node):**
  - 1-4-byte pointers + info of pointer size (e.g., 2 bits per pointer)
  - Jaspe will provide details in next talk for clever and very good strategy.
  - Typical additional compression: ~1.5x ± 0.2
  - Pointer compression not work well for **SVO**s, since pointers constitute less than ~50% of the **SVO** data and all of them are unique

cm + ix-sizes	ix	ix	ix	ix

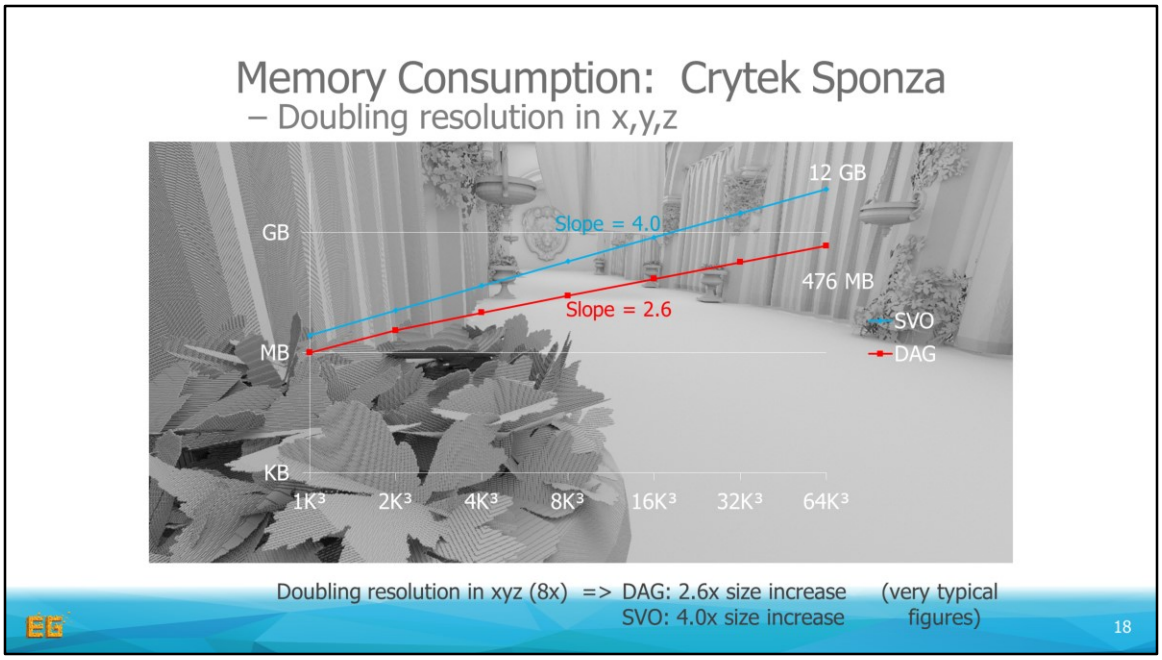
- **So, a DAG node is much larger than an SVO node**
  - This means the DAG has to rely on reducing the number of nodes drastically
  - Expected average **#bits per non-empty voxel**: **E.g., 0.06 – 1.5 bits/non-empty voxel**

Encodings of SVO and DAG nodes

- For further DAG compression, next talk explores:
  - Reflection symmetries (Jaspe)



- - The memory consumption of the SVO has increased to 12GB, and the DAG to 476MB. Both have an exponential increase in memory consumption, but the gap between the pointer-less SVO and the DAG has grown from a factor of 1.5 to a factor of 26. This shows how the memory consumption of the DAG scales better than the SVO to higher resolutions. This is even clearer in a log-plot of the same data.

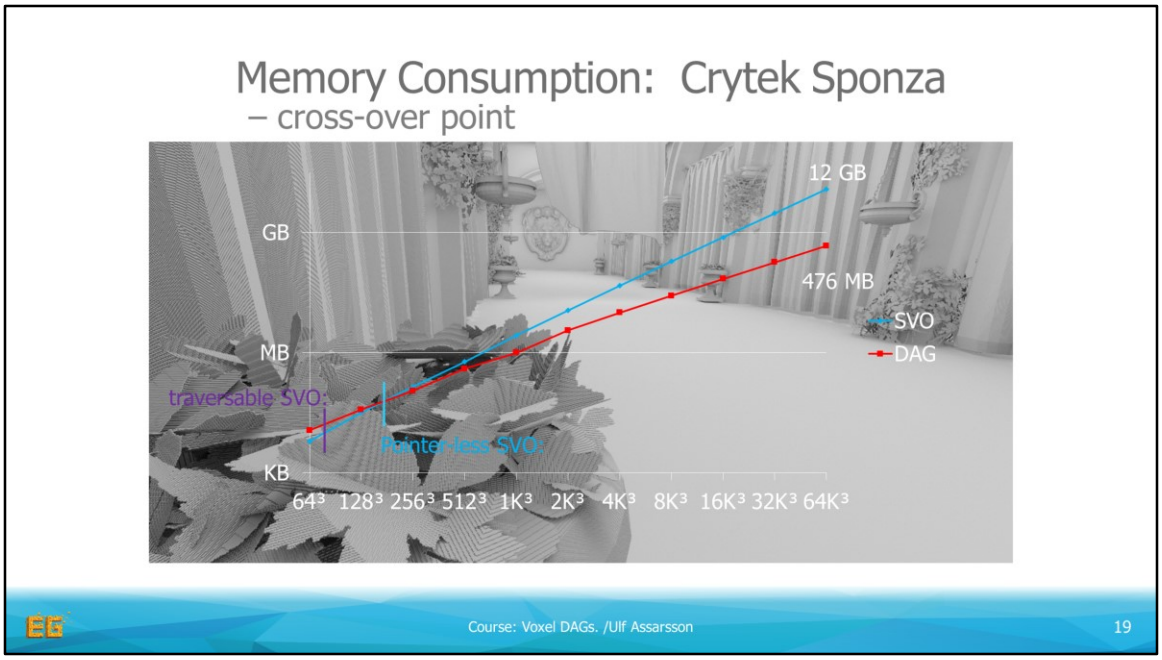


When we double the resolution in x,y,z, the resolution is increased by a factor of 8, but the memory consumption increases less – a factor 4 for SVOs and here a factor 2.6 for DAGs. The reason is that nodes will on average contain ~4 children. So the expected slope for an SVO is 4 (and it is here, as we can see), but the expected slope for the DAG is less since we expect less than 4 **individual** children on average. Next – crossover point.

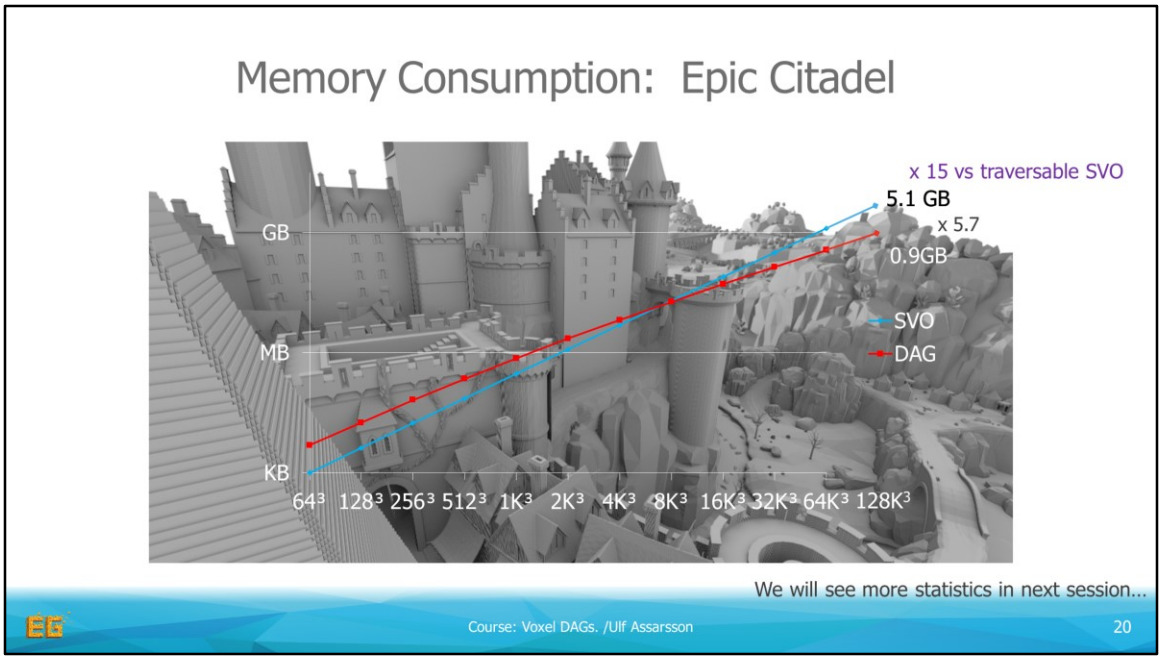
- 64^3: 4.28 bits per voxel for SVO vs 0.08 for DAG

Slopes:

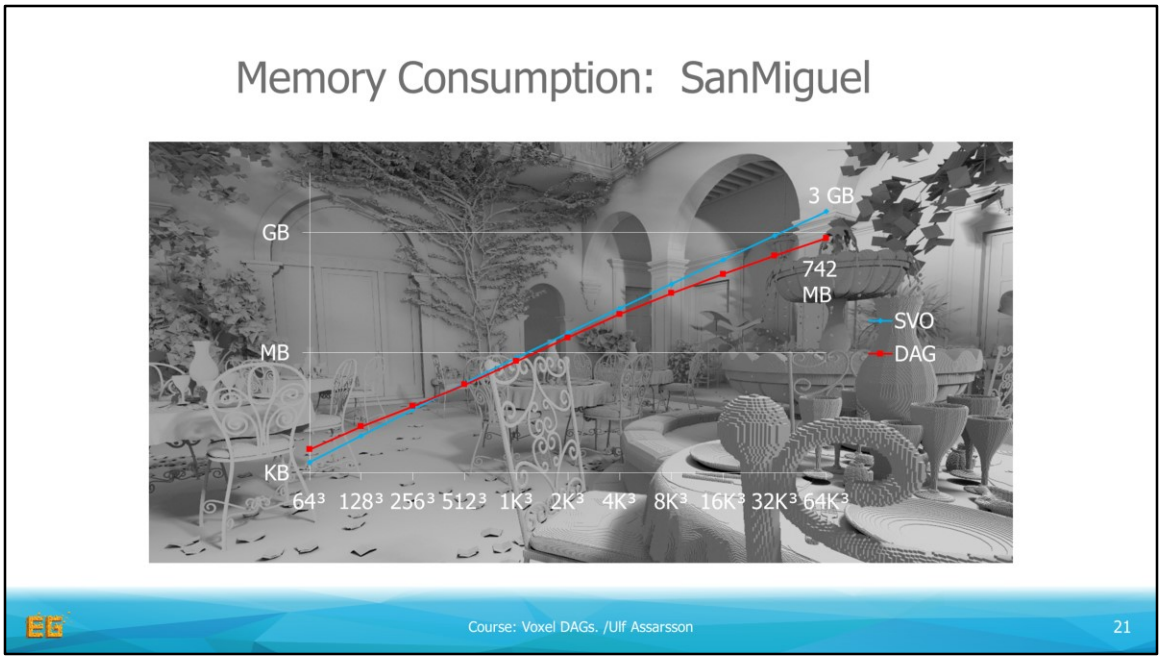
- DAG: 2.58, 2.59, 2.62, 2.45
- SVO: 4.0, 4.0,



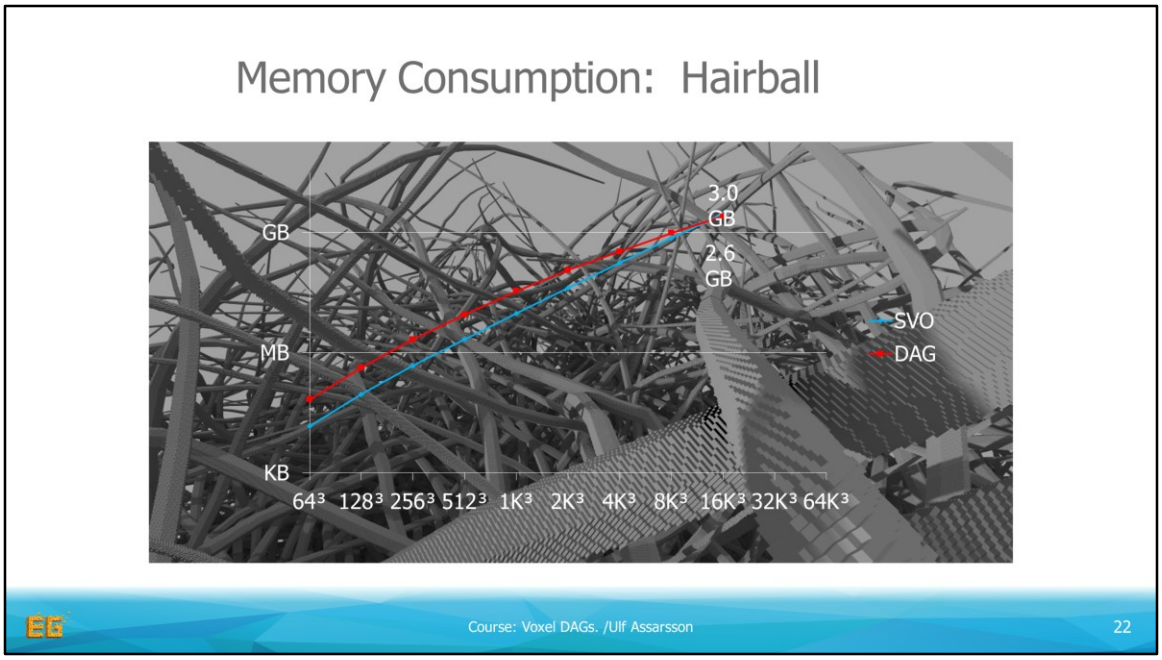
-At low resolutions the pointer-less SVO is more compact than the DAG, but for this scene the cross-over point is reached already at the 256 resolution, where the DAG is smaller, and due to the better scaling the difference increases with the resolution. The logarithmic plot shows the exponential trends very clearly. So, let's examine a few more scenes.



-The EpicCitadel contains a larger landscape and a citadel of higher geometric complexity. This scene has details on several scales, and the point of break even is not reached until the 8K resolution. The scalings looks similar to those in sponza; the DAG is scaling much better than the pointer-less SVO and we can even fit the 128K resolution voxel dag, into just below a GB of memory.



The cross-over point is at 512-cubical grids.
The SanMiguel scene is an indoor scene with lots of fine grain details. The DAG is on par to the pointer-less SVO at the 512 resolution, but once again the DAG is scaling superiorly to higher resolutions.

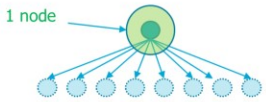


- For the really complex hairball scene, the crossover point is at 16K-cubical. This is expected. DAGs are really good for planar surfaces, and eventually, at high enough resolutions, all triangles will form larger planar regions.

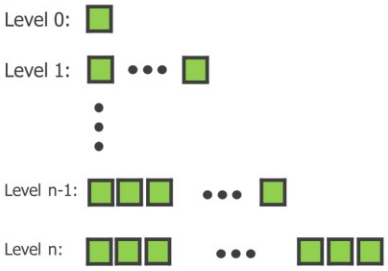
Put differently:

- The hairball is the last scene and it has a different characteristic. At lower resolutions, it is not very sparse, and it does not contain many identical volumes. The DAG and SVO shows the same scaling in the beginning, but the higher cost per node gives an offset. When we increase the resolution we start to find merging opportunities in the DAG, the memory consumption starts to scale better, and at 16K it is slightly more compact than the pointer-less SVO.
- Also remember that we are comparing our method to a pointer-less SVO, and that an SVO representation that can be efficiently raytraced will be significantly more expensive.

SVO → DAG construction



- SVO:

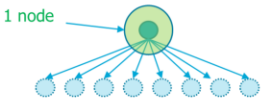


Construction algorithm:

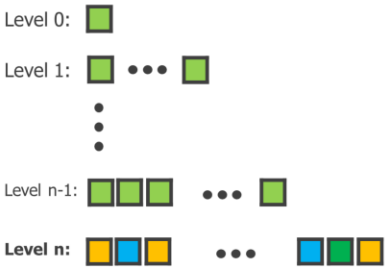
- For each level, bottom-up:
  - Sort the list of nodes
    - E.g., key = the node's bytes
  - Remove duplicates and update parents' child indices

Done!

SVO → DAG construction



- SVO:

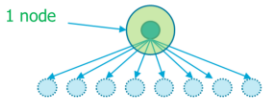


Construction algorithm:

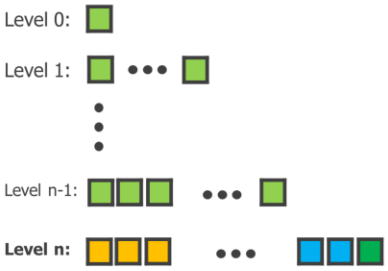
- For each level, bottom-up:
  - **Sort the list of nodes**
    - E.g., key = the node's bytes
  - Remove duplicates and update parents' child indices

Done!

SVO → DAG construction



- SVO:

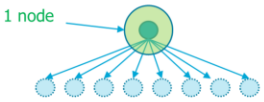


Construction algorithm:

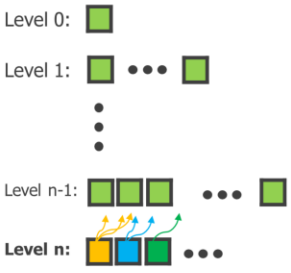
- For each level, bottom-up:
  - **Sort the list of nodes**
    - E.g., key = the node's bytes
  - Remove duplicates and update parents' child indices

Done!

SVO → DAG construction



- SVO:

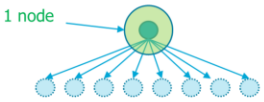


Construction algorithm:

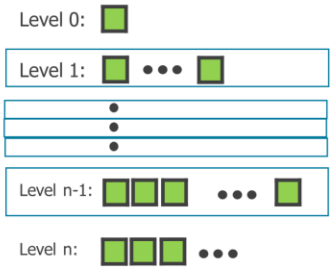
- For each level, bottom-up:
  - Sort the list of nodes
    - E.g., key = the node's bytes
  - **Remove duplicates and update parents' child indices**

Done!

SVO → DAG construction



- SVO:



Construction algorithm:

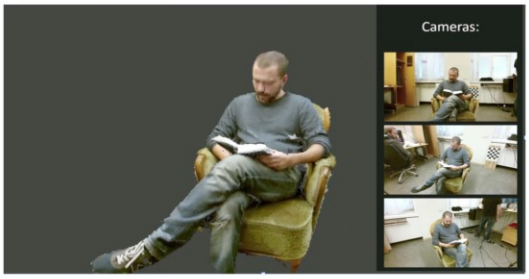
- For each level, bottom-up:
  - Sort the list of nodes
    - E.g., key = the node's bytes
  - Remove duplicates and update parents' child indices

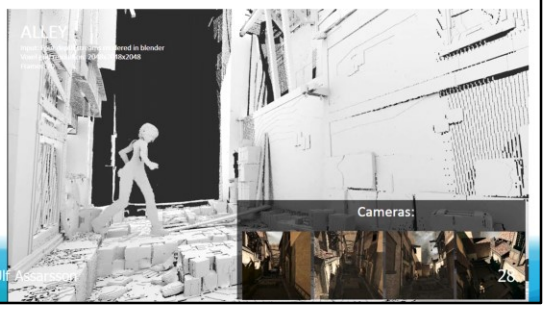
Done!

Sort	O(n log n)
Remove duplicates and update parent pointers	O(n)
Most nodes (~75%) are at the leaf level	

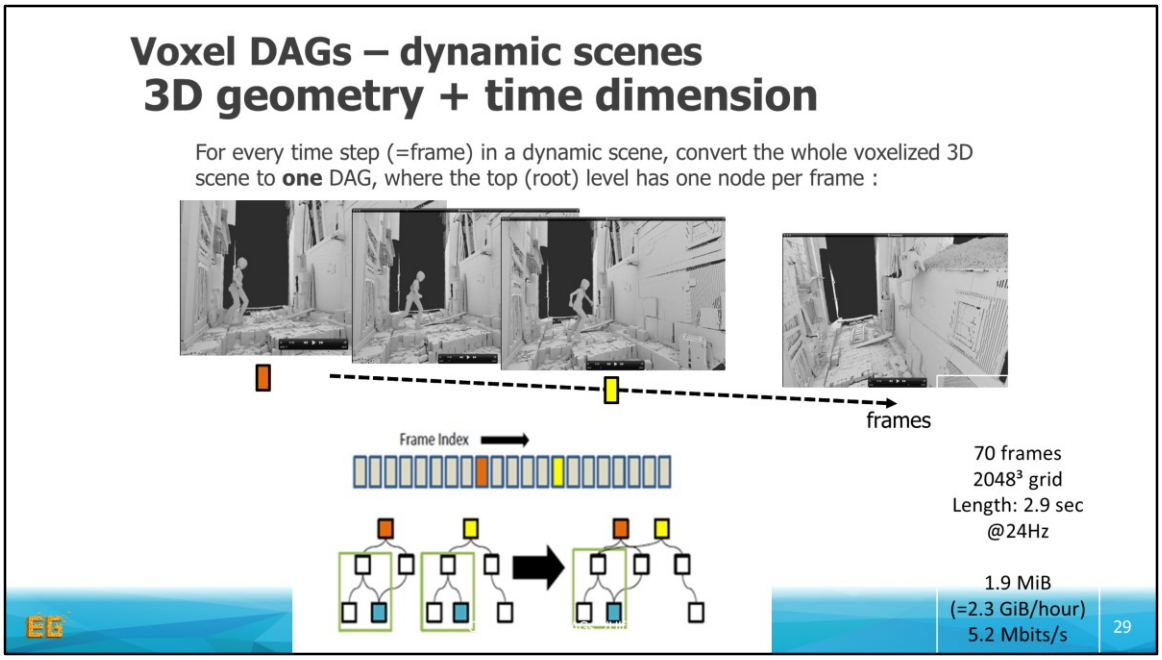
The complexity.
The GPU can sort 1B elements in ~1second. A 128K^3 scene contains some 10B voxels.
The expensive part is the voxelization. Not the SVO to DAG conversion.

Sparse Voxel DAGs: From static scenes to dynamic (moving) scenes to Free Viewpoint Video



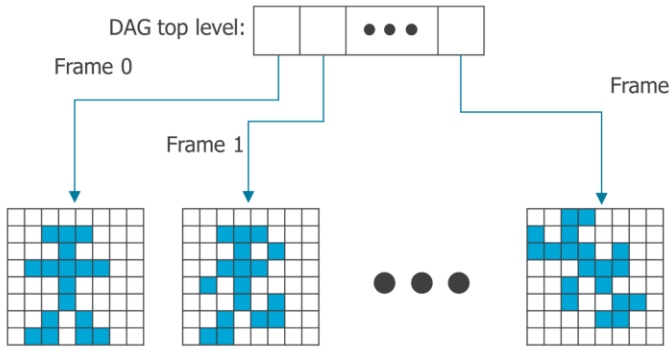


Kämpe, Rasmuson, Billeter, Sintorn, Assarsson. *Exploiting Coherence in Time-Varying Voxel Data*. I3D 2016.



We do not need to separate static and dynamic geometry. That is automatically handled to maximum efficiency.
And any coherency **between** static and dynamic geometry is also automatically found. Separating would just decrease the number of merging opportunities.

Time-varying Voxel Grid - construction



Root is a vector of one DAG per frame

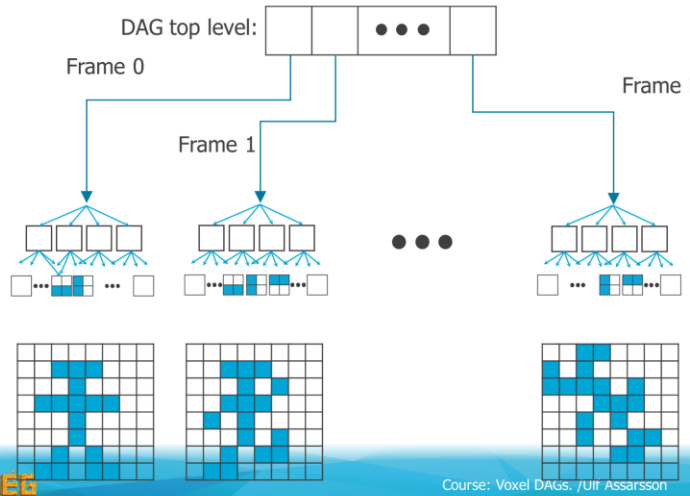
1. Convert each frame to a DAG
2. Merge equal nodes, bottom-up

Done!

Real 3D example:

Frame 0 Frame 1 Frame 2

Time-varying Voxel Grid - construction

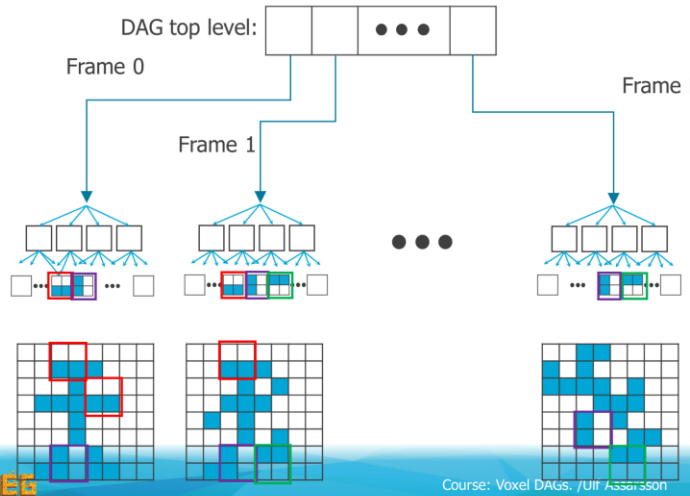


Root is a vector of one DAG per frame

1. **Convert each frame to a DAG**
2. Merge equal nodes, bottom-up

Done!

Time-varying Voxel Grid - construction

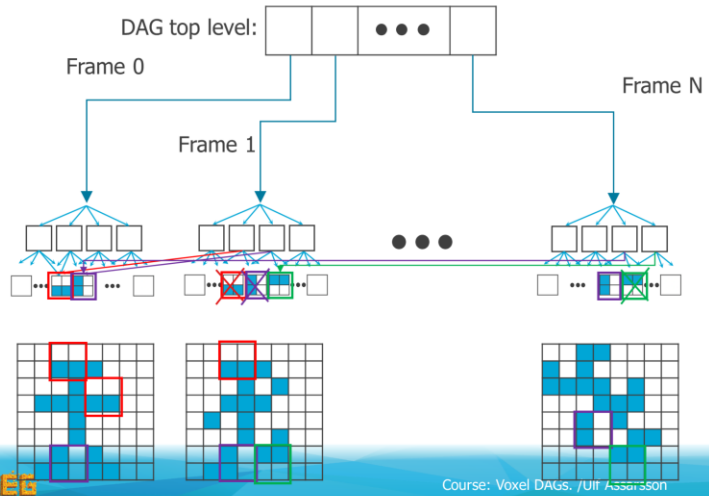


Root is a vector of one DAG per frame

1. Convert each frame to a DAG
2. **Merge equal nodes, bottom-up**

Done!

Time-varying Voxel Grid - construction

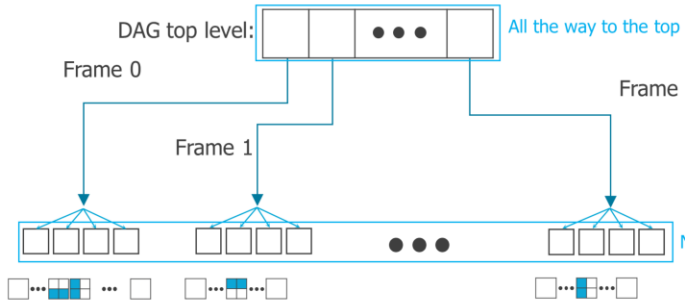


Root is a vector of one DAG per frame

1. Convert each frame to a DAG
2. **Merge equal nodes, bottom-up**

Done!

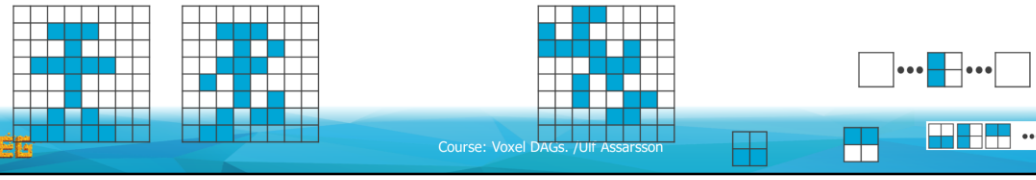
Time-varying Voxel Grid - construction

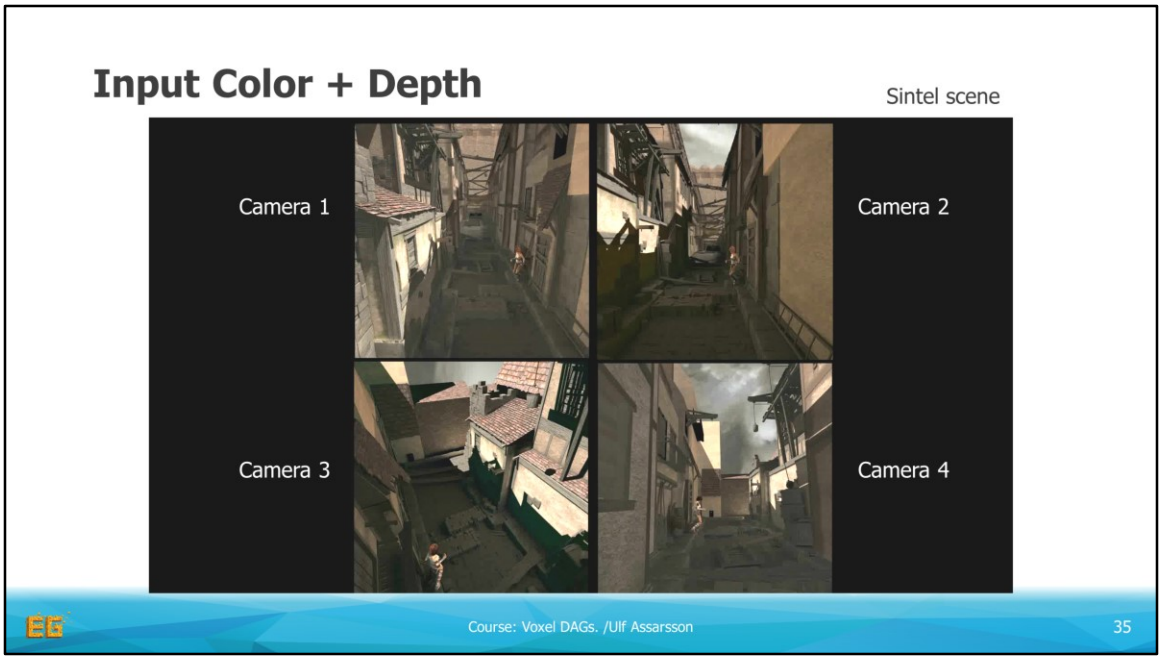


Root is a vector of one DAG per frame

1. Convert each frame to a DAG
2. **Merge equal nodes, bottom-up**

Done!



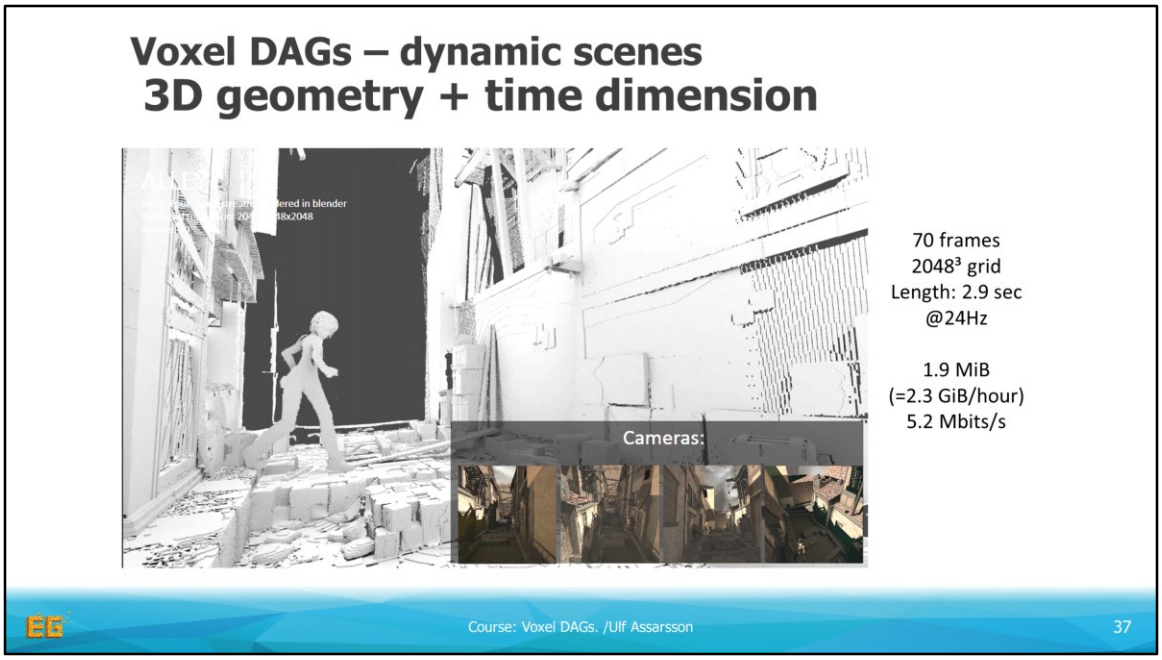


This is a snapshot from the Sintel movie. We capture and voxelize it by rendering rgb- and depth images from 4 camera positions, and then convert each frame to a DAG. So then, when we have the whole animated sequence as DAG, we can render this sequence in real time, from any arbitrary viewpoints and move the virtual camera around.

Novel Viewpoint

Kämpe, Rasmuson, Billeter, Sintorn, Assarsson. *Exploiting Coherence in Time-Varying Voxel Data*. I3D 2016.

So, here we see a virtual camera in real time, looking and moving around in the scene. The real-time rendering is pretty fast and simple.
For the colors, we simply backproject the rgb-camera streams onto the voxels.



Well, what's the point to video record something that we easilly could render in real time with triangles?
We could perhaps store physics animations of fluids or other complex soft-bodies.
But perhaps more interestingly, we could exchange these camera streams with real cameras of the physical world. => FVV.
Free viewpoint video means that you can walk around freely inside the movie during playback.

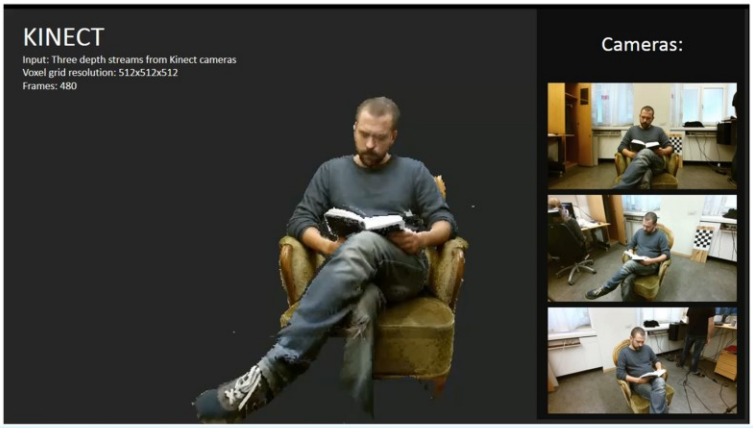
Free Viewpoint Video

Want:

1. convert a real dynamic scene to 3D graphics, 24 times/second.
2. Render scene from any viewpoint.


With 2 or more cameras, depth can be computed – like our eyes work.
Course: Voxel DAGs. /Ulf Assarsson
39

Free Viewpoint Video



480 frames
512^3 grid
20 sec
@24Hz

5.2 MiB
0.9 GiB/hour
2.1 Mbits/s

Kämpe, Rasmuson, Billeter, Sintorn, Assarsson. *Exploiting Coherence in Time-Varying Voxel Data*. i3D, 2016

Construction time (DAG-ification): ~1 minute
Free-viewpoint-video playback: real time on GTX 680