If every voxel has an attribute, like a color.
What about colors?
There are two papers on how to extend the DAGs to be used together with voxel colors coherency.


DAGs compress to around 1 bit or less per set voxel. Thus, 16-24-bit color information would be devastatingly expensive in terms of memory usage. So far, DAGs haven’t stored color information – just the geometric information. The key to the success of the DAGs was to separate colors and geometry, and so far, we have just cared about the geometry. If the DAG nodes would contain color information, that would destroy much or often all of the merging opportunities and therefore destroy any efficient compression. We have two problems...
Colors – preserving coherency

- DAGs: geometry \( \leq \) 1 bit per voxel
- 16-24 bits colors/voxel would be devastating

Two problems:
1. How can we compress the colors efficiently?
2. Connection between DAG nodes and voxel colors with fast color lookups during run-time traversal (i.e., visualization)?

1. Existing 3D color-compression algorithms give too low compression; we have \(~100,000\) times higher resolutions, but mainly surface geometry (not gas, smoke, fluids, etc).
2. Surfaces are two dimensional (in 3D-space)
   => makes sense to try using 2D image-compression algorithms (due to the 2D coherency).
   - How unfold a surface to a flat image?
   - Any distortions waste or loose precision.

Vs 3D fourier-transform-based methods.
If we have a voxelized object like this dwarf, we want to outline the voxels somehow and compress them.
We don’t want to order the voxels in a random order, because that destroys much of the spatial coherency and would be devastating for color compression.
If we unravel the voxels in the 3D grid using a coherency-preserving space-filling curve (as this Morton order or Hilbert order), we get a sequential 1D-stream with (much) spatial coherency preserved..

Then, we outline this 1D-stream onto a 2D image using a 2D-Morton order, which again preserved much of the coherency. Then, we can compress this image using standard 2D-image compression algorithms.

So, how do we traverse the voxels according to a 3D-Morton order?
Colors – preserving coherency

- Ordering voxels along a Morton curve just requires us to enumerate the SVO/DAG-node children in the corresponding order.
  - (Hilbert order is similar and easy but for DAGs requires respecting the parents' child numbers during traversal. For SVOs, the order can be coded, at SVO construction, by the child enumeration.)

Now, we have unfolded the voxel colors and can compress them. So, what does the resulting 2D image look like...?

We just store the children of each node according to a Morton order. Then, a full-tree depth-first traversal will access all voxels in a Morton order.
The image looks blocky (as expected) but it corresponds to the same block dimensions that typical block-compression algorithms prefer. So, it is at least in my opinion surprisingly good.

Dan Dolonius will present compression efficiency in a few minutes.

And he will also describe a version which can reach below 1 bit per color (not 1 bit per r,g, and b respectively, but less than 1 bit per color in total).
For an SVO, we can simply store the colors or color indices in each node. But for DAGs, that destroys all or most of the merging opportunities – especially if we use baked scenes.

So, we need to invent a method to insert information in the SVO that does not destroy the merging opportunities.

We can do that by storing, for each node, its "number of voxels in the subgraph". If we have colors at leaf levels only, that means that each node stores the number of leaf voxels in its subgraph.

Then, to compute the color index at node i, the only thing we need to do is to, for each node along the path, sum the values of all preceding siblings.

Like this (look at right side of slide):

for each path node (red), we sum all the values of the preceding siblings,

which corresponds to all the voxels in this green subgraph...

Until we have the final correct index.

This works for DAGs as well...
We don’t care if two pointers point to the same node. We just sum the value that each child pointer points to.

(Transition to next slide) What about mipmap colors? Colors at all levels and not only the leaf level?

Two problems:
1. How can we compress the colors efficiently?
2. Connection between DAG nodes and voxel colors with fast color lookups during run-time traversal?

DAGs with color indices

• For colors at leaf voxels only

SVO:

Or compute index by storing #voxels_in_subgraph\(^1\) at each node. Works for both SVOs and DAGs.

To compute color index at \(i\):
For each node along the path:
sum the values of all preceding siblings

\(^1\)Number of voxels with colors in the node's subgraph (i.e., #leaf voxels)
The voxels are ordered according to a depth-first traversal.
I will toggle these slides back and forth.

As you can see, the only things that change are that the number we store at each node now includes the voxel colors at all levels – not only the leaf level. And our node x can be at any leaf or intermediate level. And we need to add the number of path_nodes_above_node_x.

The reason for the latter is that summing all the siblings values here (green) will give the number of voxel colors in this green subgraph.

And the same for the blue and white here. But we then have missed the color stored for the root, and these two red path nodes.

So, that’s why we add this “number of path_nodes_above_node_x”, i.e., 3. And that’s it.

Transition: However, we can optimize if we want to...
However, the colors are typically accessed much more seldom than the nodes themselves, during ray traversal. So, in my research group, we prefer having the smaller DAG which boosts cache performance, and pay the slightly higher but more rare cost when the colors need to be resolved.
We lift up the prefix sums of these $v$-numbers (the number of subgraph voxels) to the child pointers.
We do that for all nodes in the whole DAG.
Next, Dan will talk about the color compression in more detail.