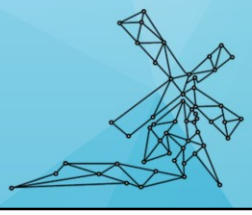
DAG Construction

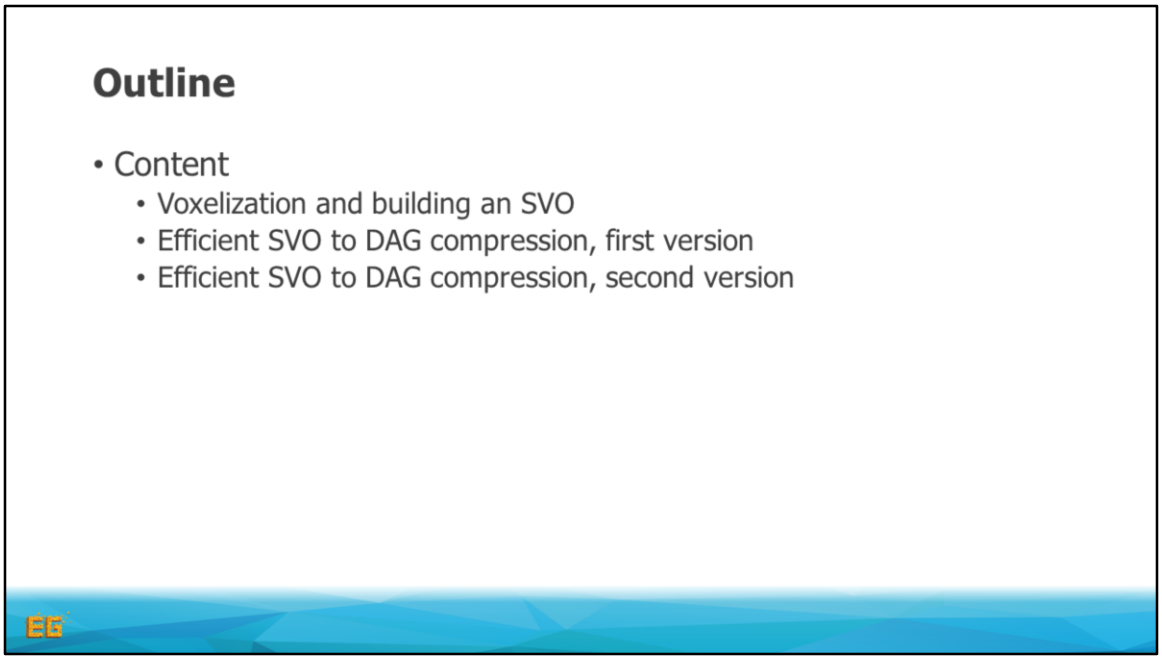
Erik Sintorn

Chalmers University of Technology, Sweden

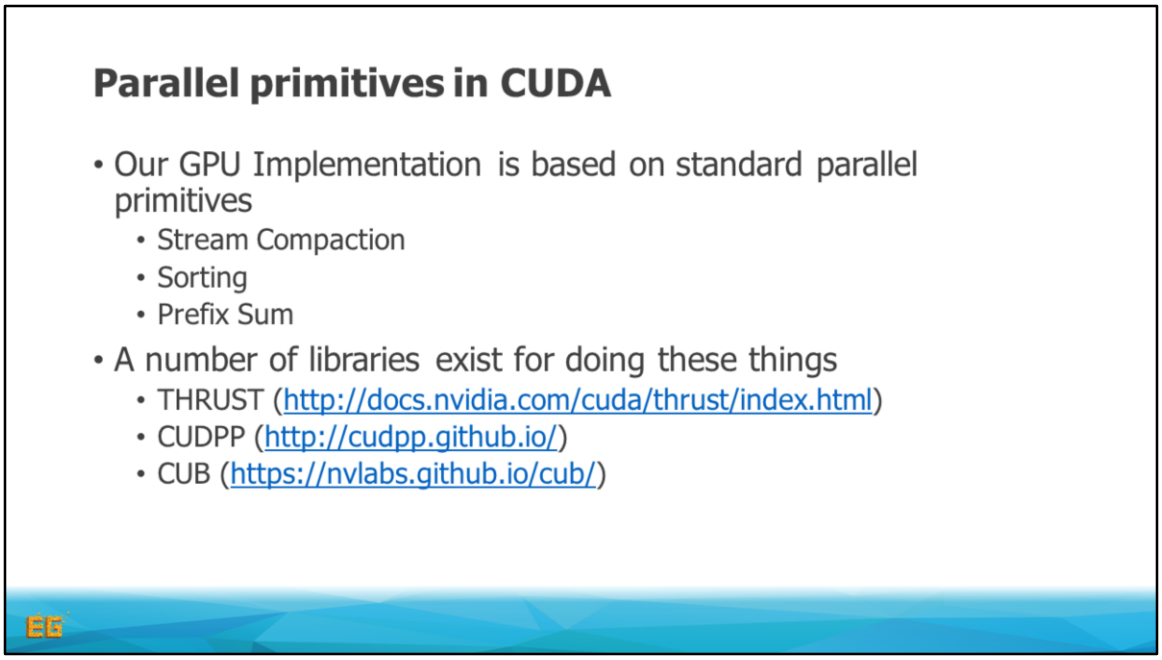
EUROGRAPHICS 2018



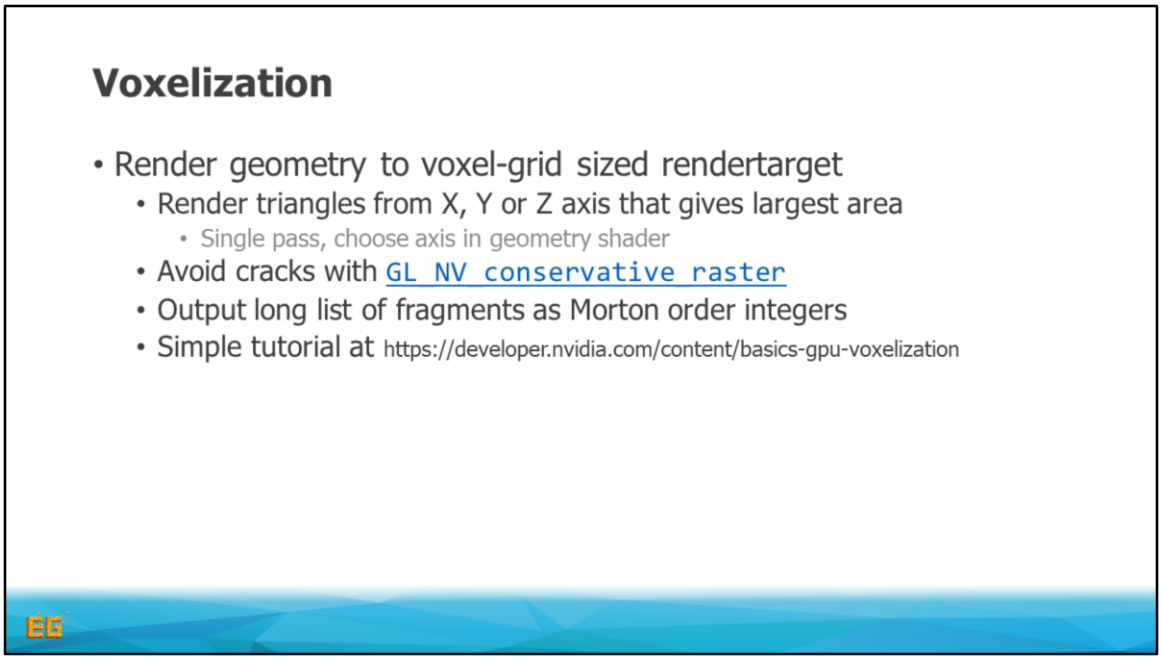
Welcome back
Before the break, we went through the theory.
These next two talks will be on more practical things.
I will start by talking about how we actually build these DAGs on GPUs, using parallel primitives.



So, the outline of the talk is that:
I will first talk briefly about voxelization
And then about how to create an SVO from the voxelization
Then I will explain how we *used to* build a DAG from the SVO
And then I will go through an alternative way of doing this, which is quite a lot simpler and faster.



Before I start though,
I will talk a lot about these parallel primitives.
These are things like: sorting an array, stream compaction and prefix sum.
Stuff that must be fast, and that you do not want to write yourself.
There are a number of libraries available for this:
THRUST actually ships with CUDA, but the last time I tried it it performed very badly,
So I now use CUB.
Either of these libraries should work though.



The first step is to voxelize our scene.
We do this in a single renderpass, with a viewport set up to match the resolution of the final DAG.
For each triangle we choose to project it as seen from the X, Y or Z axis, depending on which gives the largest area.
Also, to avoid cracks, we need to use some sort of conservative rasterization.
This is pretty easy these days, with the GL_NV_conservative_raster OpenGL extension.
In the fragment shader, we will simply turn the fragments position into the grid coordinates, and output it to a long list as a *Morton code.*
There are many tutorials on voxelization on the internet, and here's a link for one of them.

Build SVO in CUDA

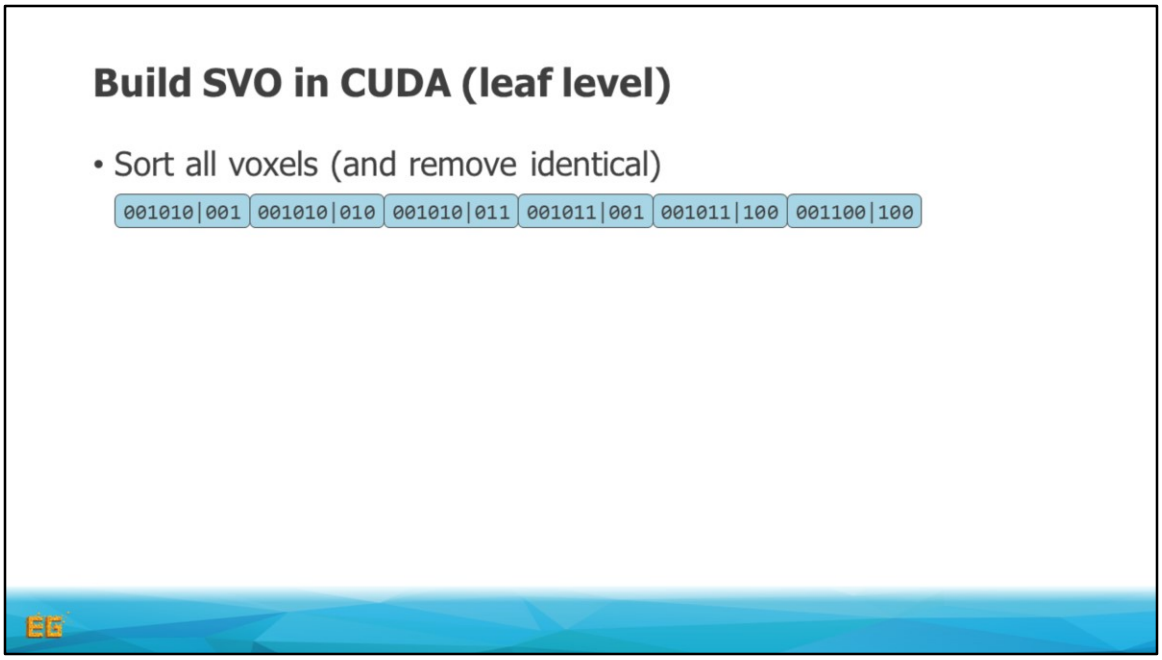
- Given the Morton cords, v_i, of all set voxels
  - Parent coord is $(v_i \gg 3)$
  - Child index is $(v_i \& 7)$

parent coord

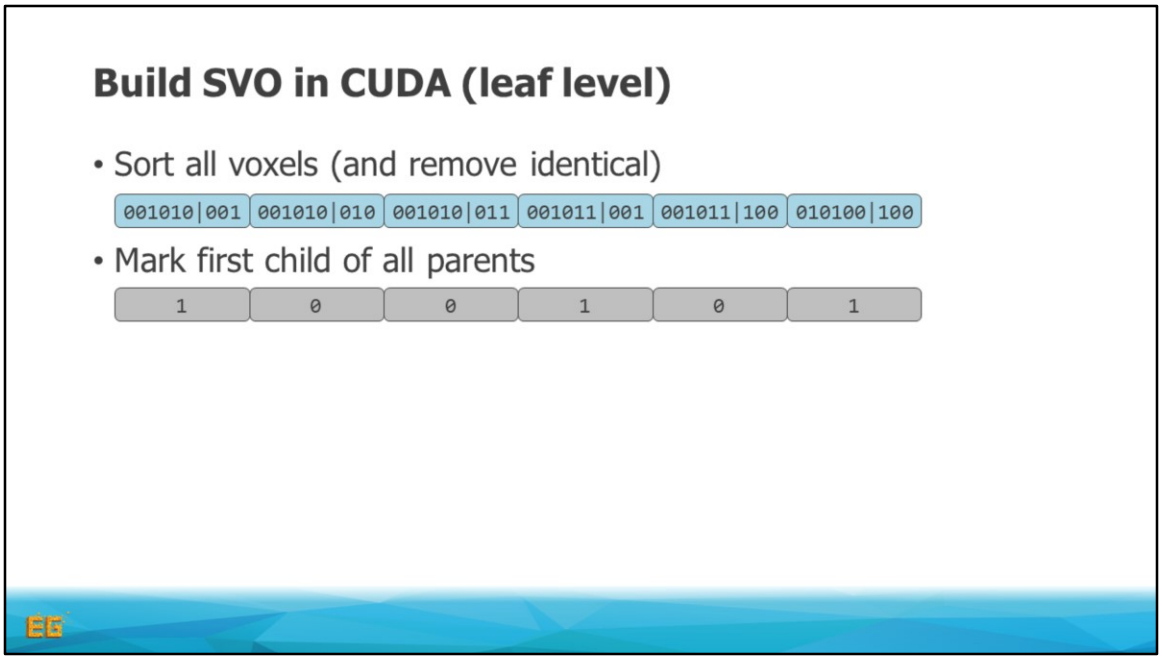
$$v_i = x_N y_N z_N \ldots x_1 y_1 z_1 x_0 y_0 z_0$$

child index

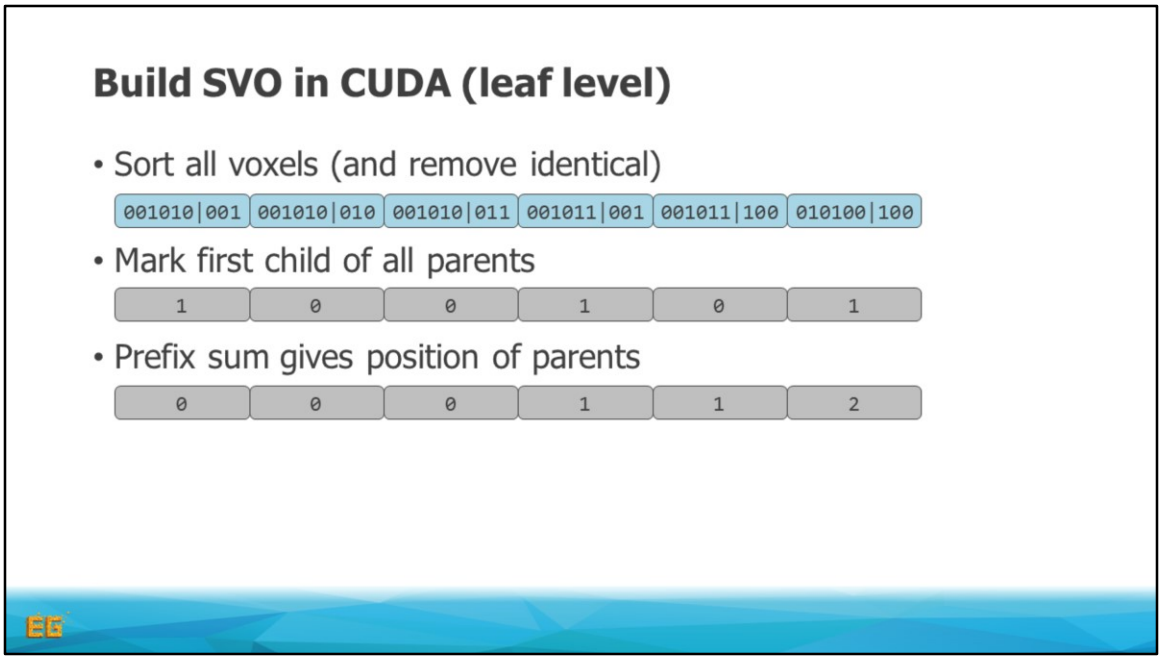
We output the voxel's position as a Morton code, because that makes it very easy mask of the three lowest bits and
<click>
Then the upper bits are the coordinates of the leaf node,
And the lower bits are the index of the voxel within its parent.



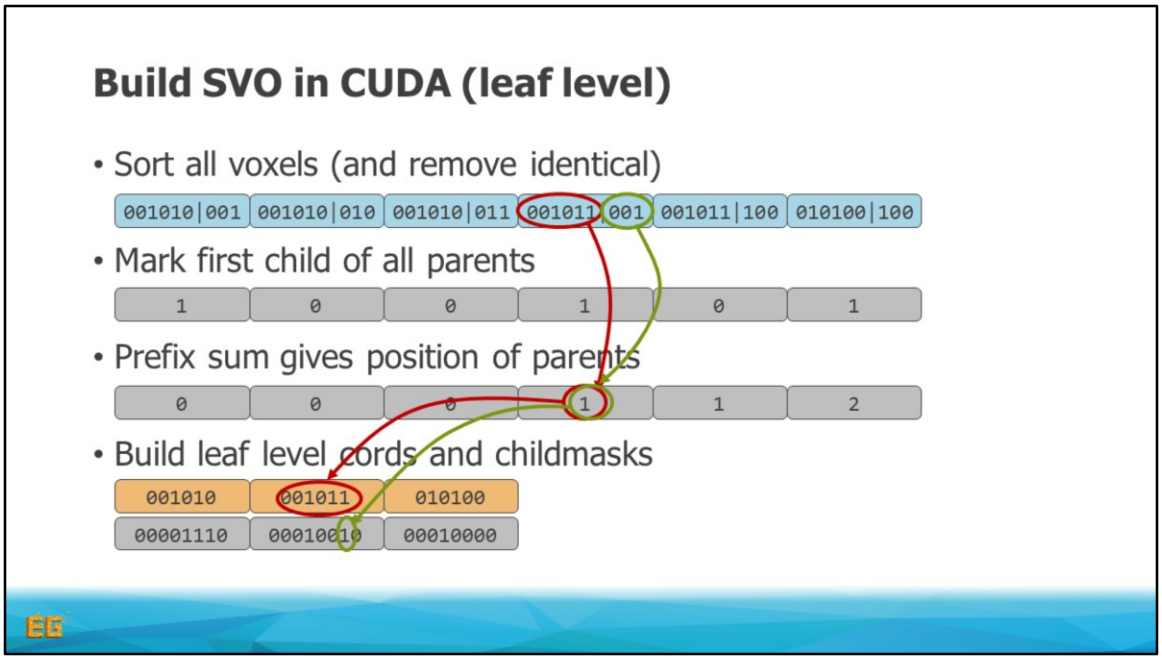
Now we have a long list of voxels, and we want to build an SVO.
We start by sorting the list, and removing all identical nodes.
This is stuff that a library will handle efficiently for us.



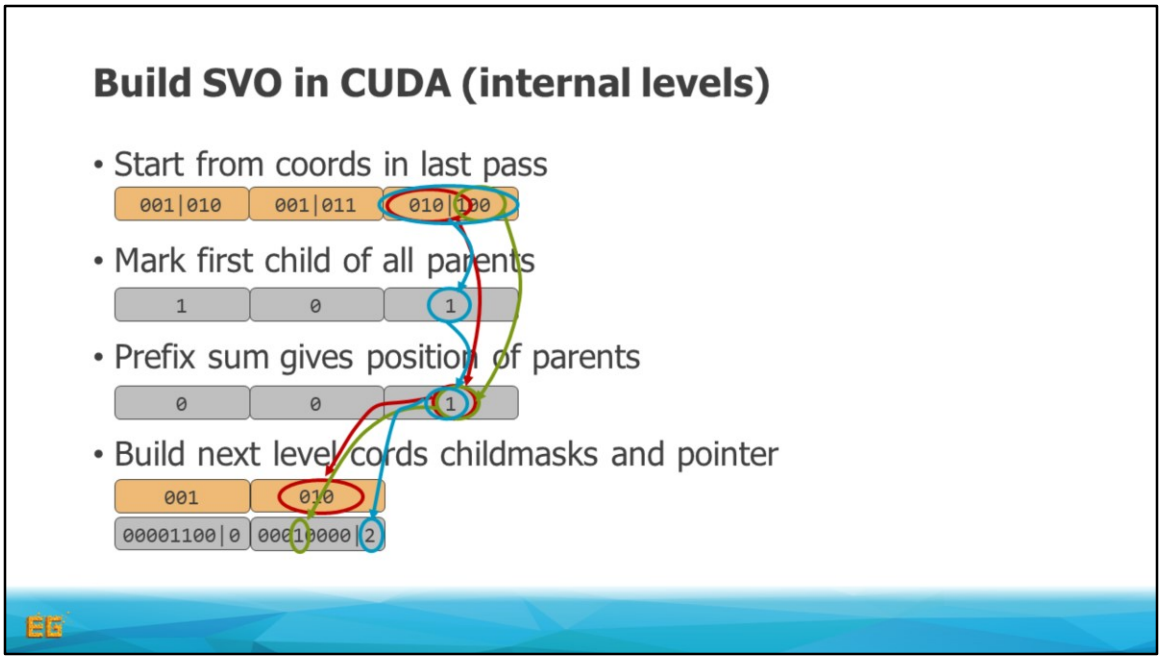
Then, we mark the *first* voxel of each leafnode. We do this by running through all the voxels, and checking the voxel to the left.
If it has the same parent: We write a zero.
If it has a different parent, this is the first child, so we write a one.



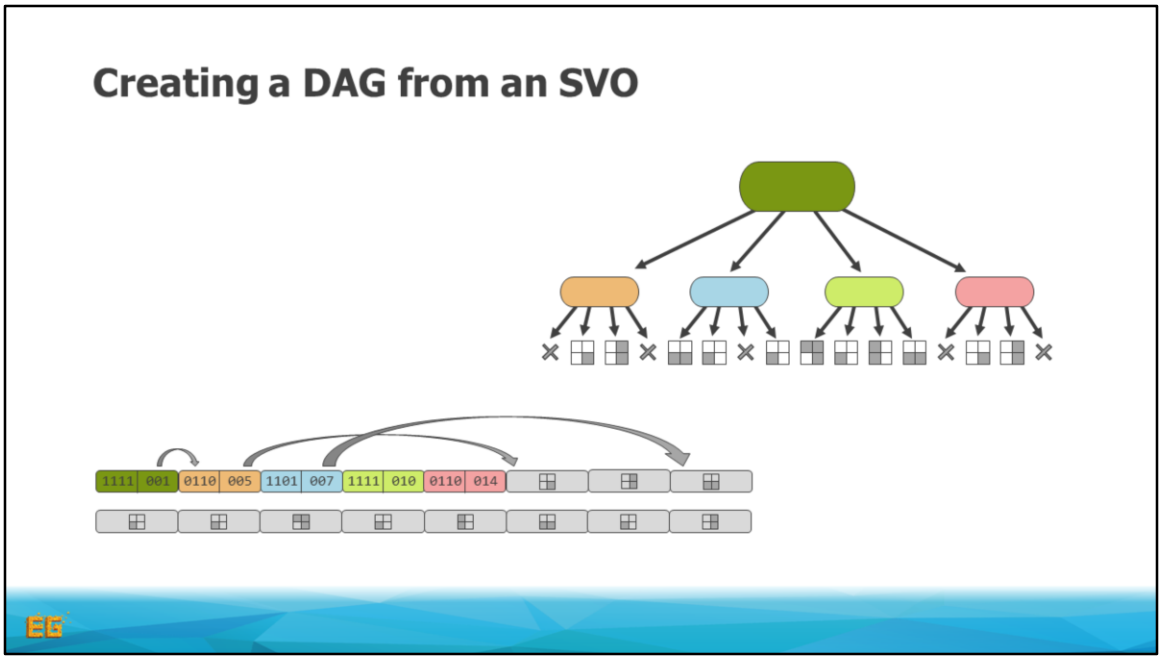
We can then do a *prefix sum* on this list, which will give us, for every voxel, the index of its parent.



Now we can build the leaf level of our SVO.
<click>
We first build a list of all the leaf level coordinates, that we will use to continue building the tree
We do this by running through all the voxels,
finding the position of the leaf node it ended up in,
and writing down the parent coord at that location
<click>
Then we build the actual leaf level of the SVO, that is, the childmasks of all the leafs.
We mask out the index of the voxel in the leaf node,
look up the position of the leaf node
and set the right bit in the corresponding child-mask

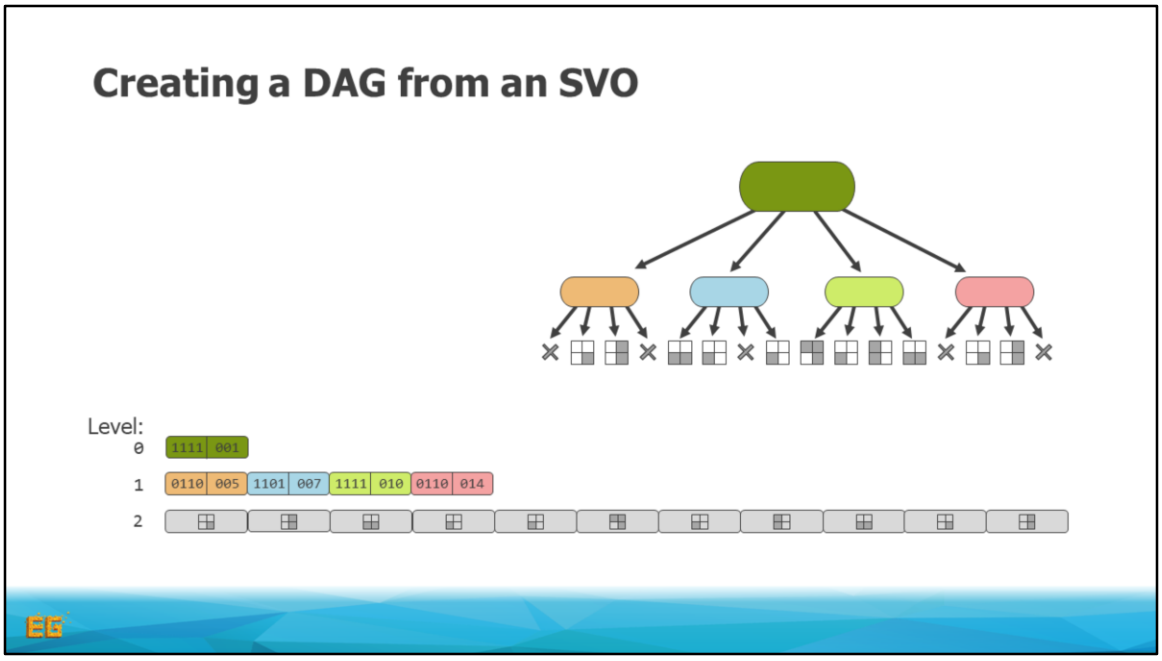


Now we have the leaf level of the SVO, and we need to build all of the internal nodes.
We do this in almost exactly the same way.
So, we *start from the leaf coordinates* we created in the last pass,
Mark the first child of all parents
Prefix sum that list to get the position of the parents.

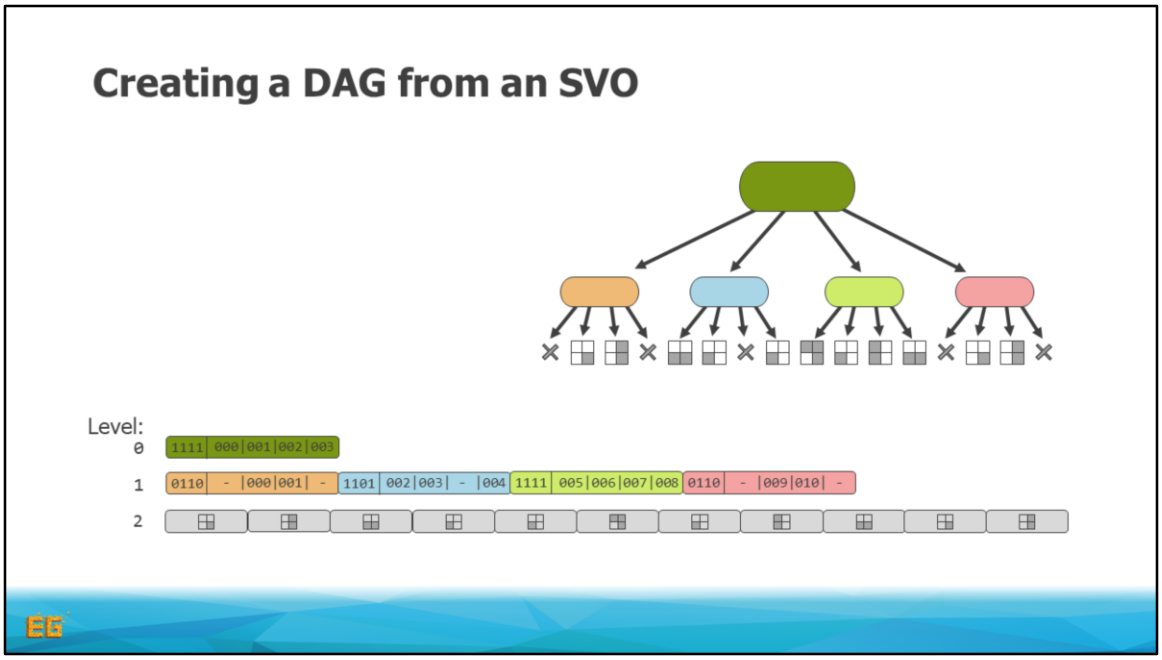


This dag is the ‘S’

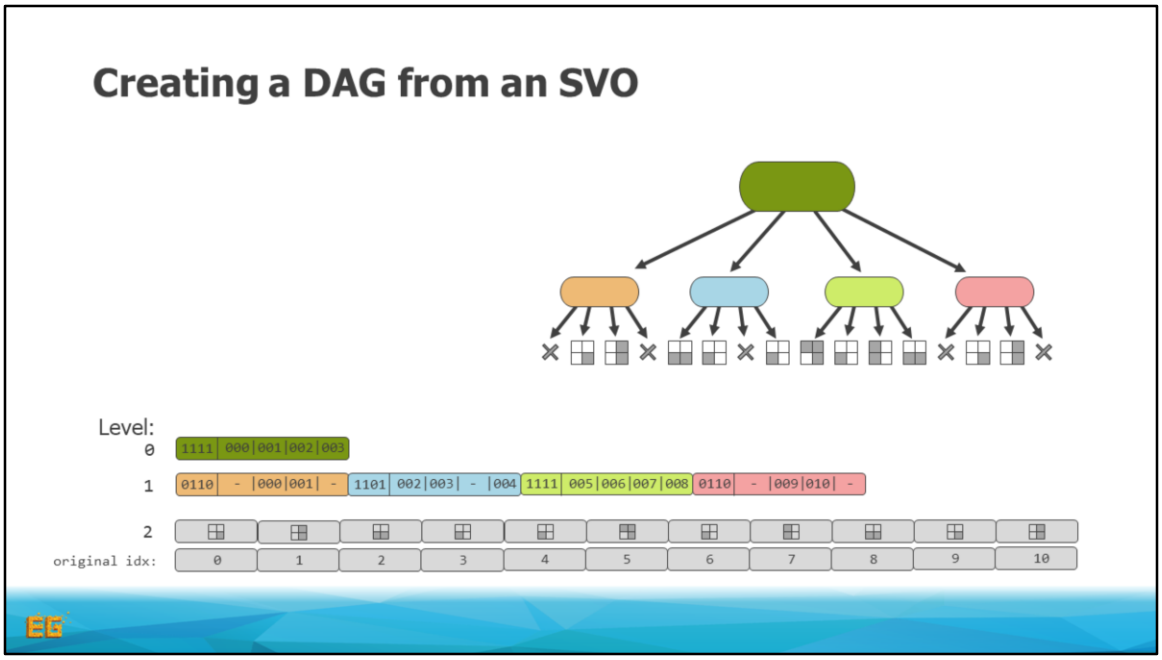
For brevity, let’s say each node takes one word



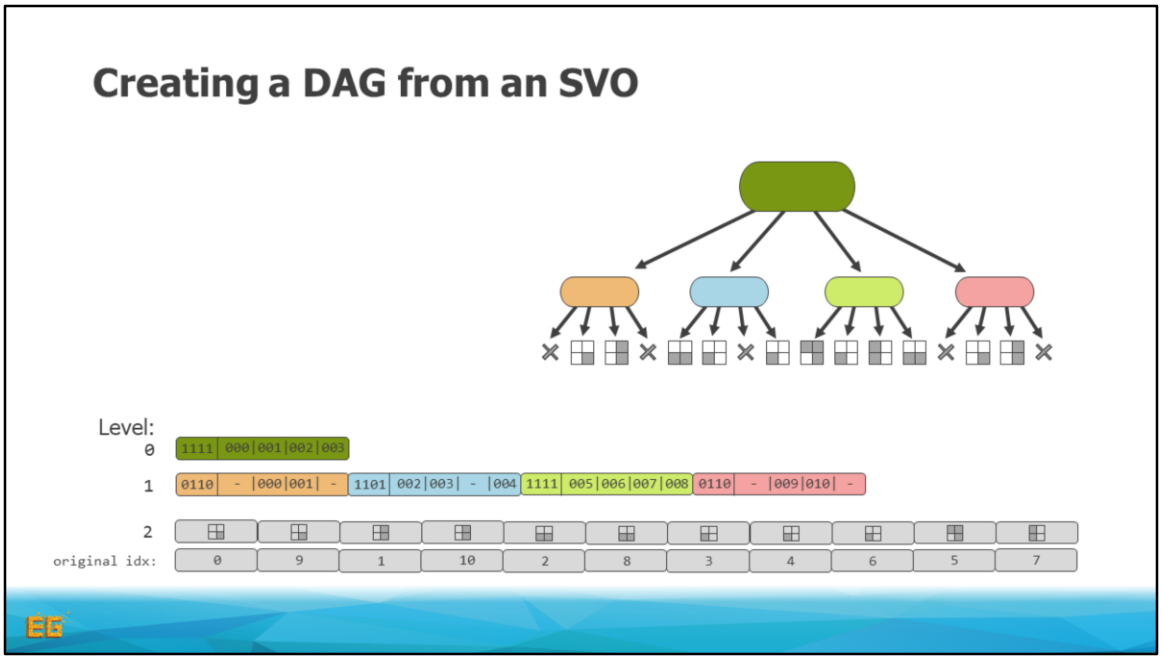
Ta make this easier, let's visualize the words divided into levels



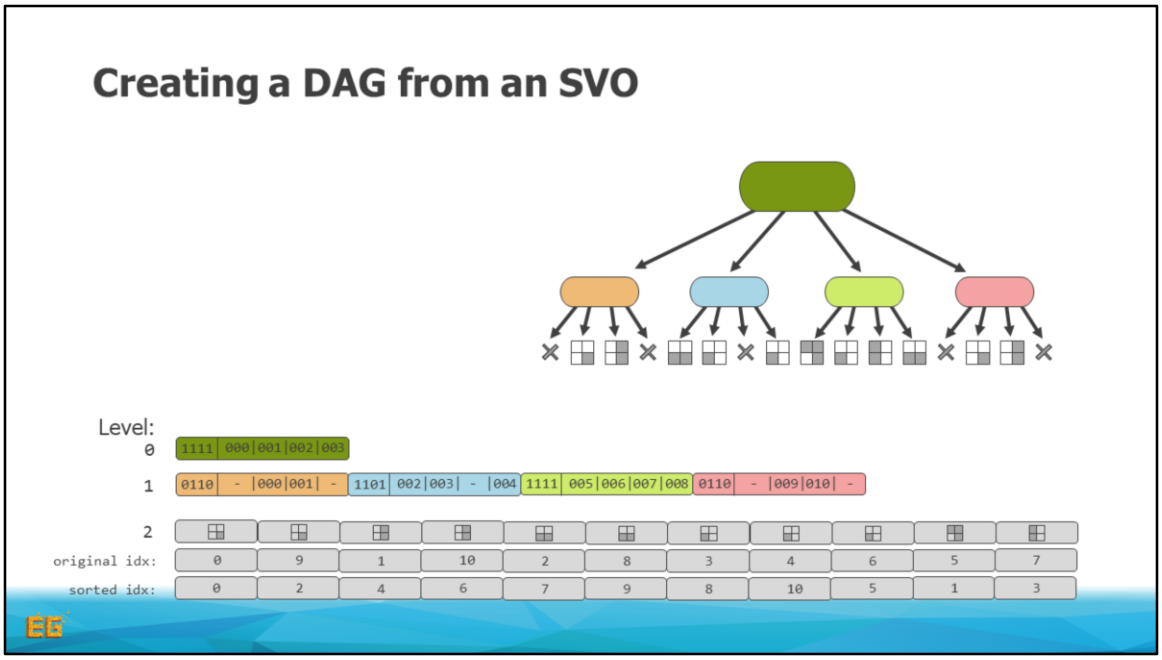
Then, the first step is to expand nodes so they have eight pointers (one per possible child)
While building, we store null pointers for non-existing children, so the nodes are easily indexable
Also, we write the indexes as offsets into each *level*



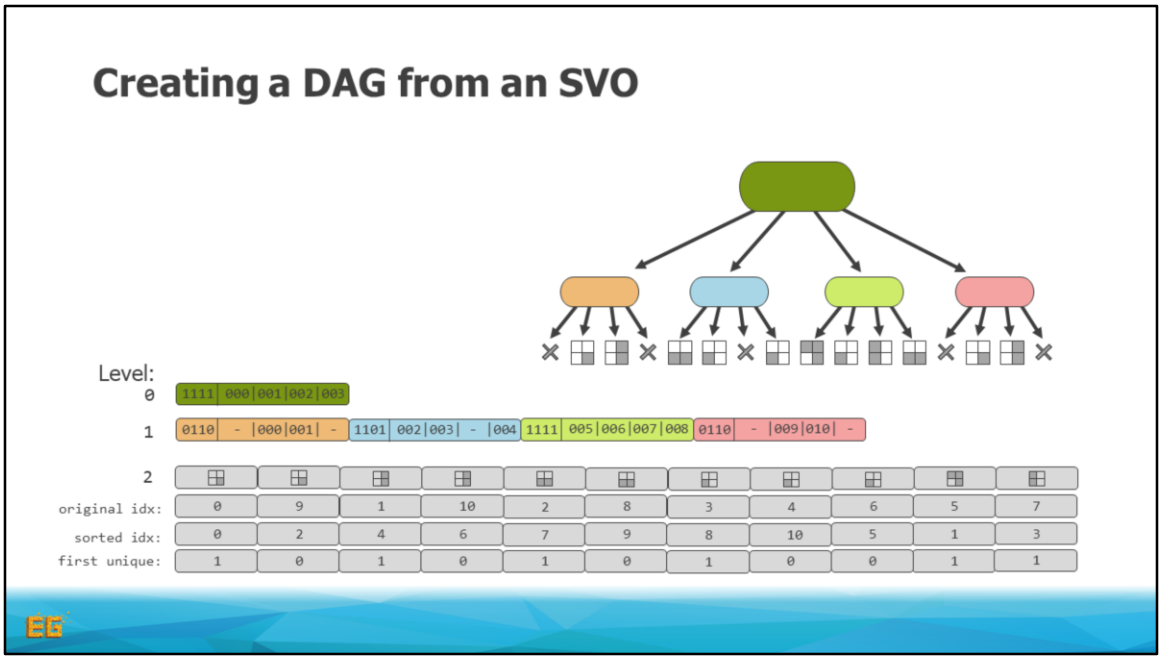
Now we create a list with the index of each node in the leaf level



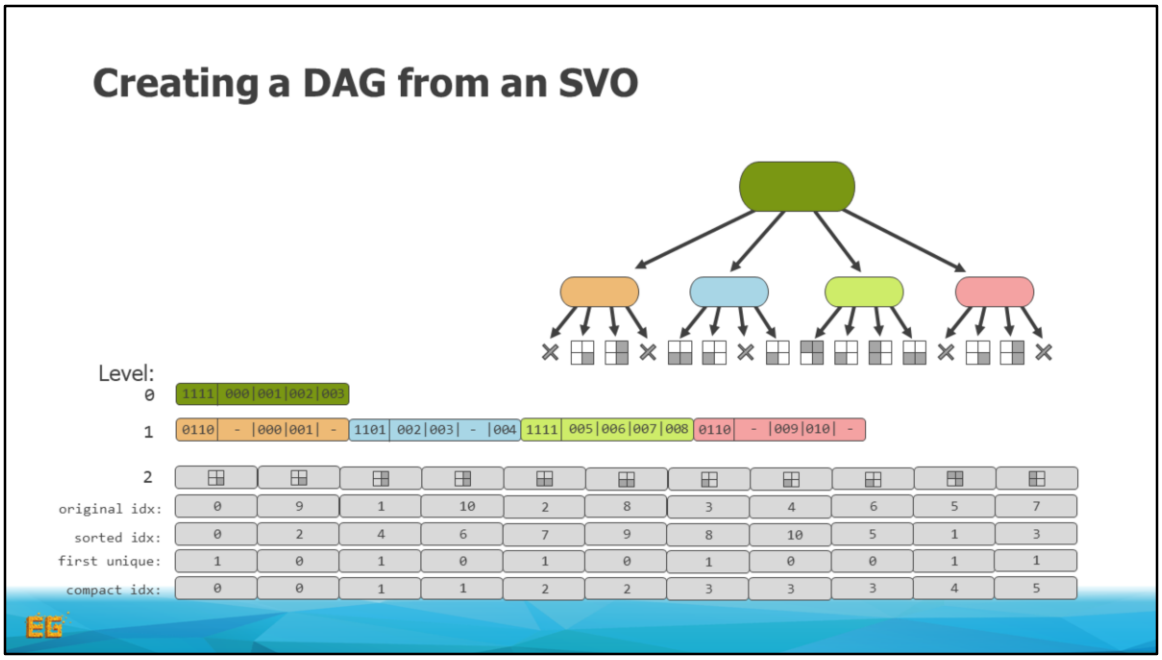
We then sort that list, and the leaf nodes, based on the child masks in the leaf nodes. The sorted list of numbers tells us where each node was *before* sorting.



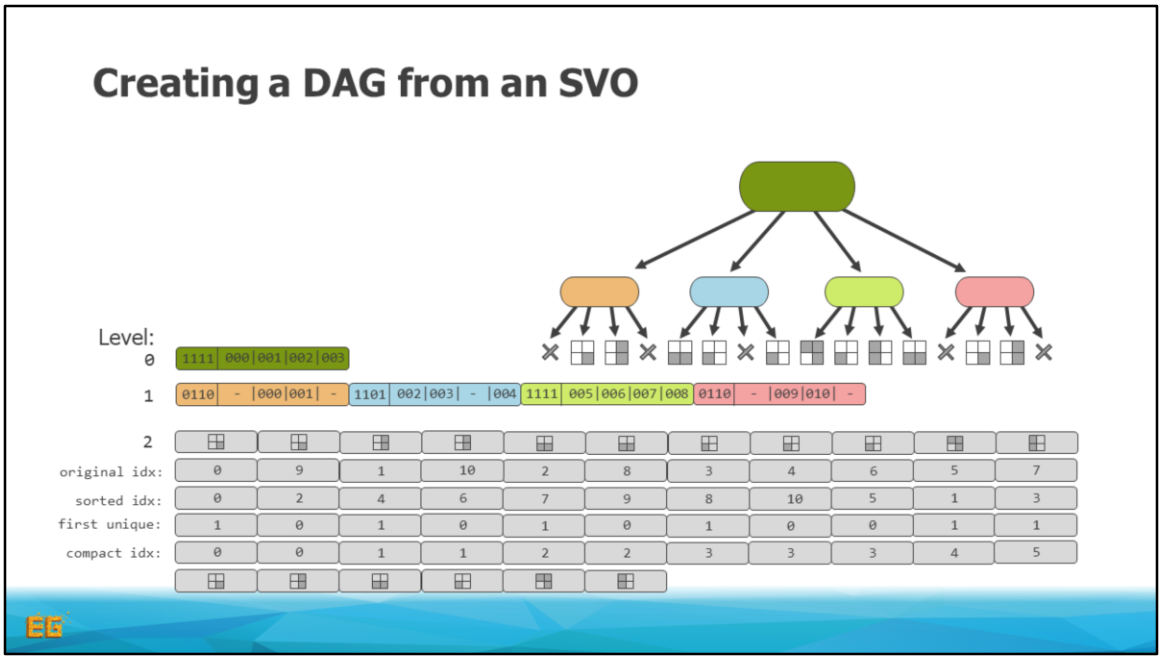
We now create the inverse of that list so we can look up where each node *ended up* after sorting



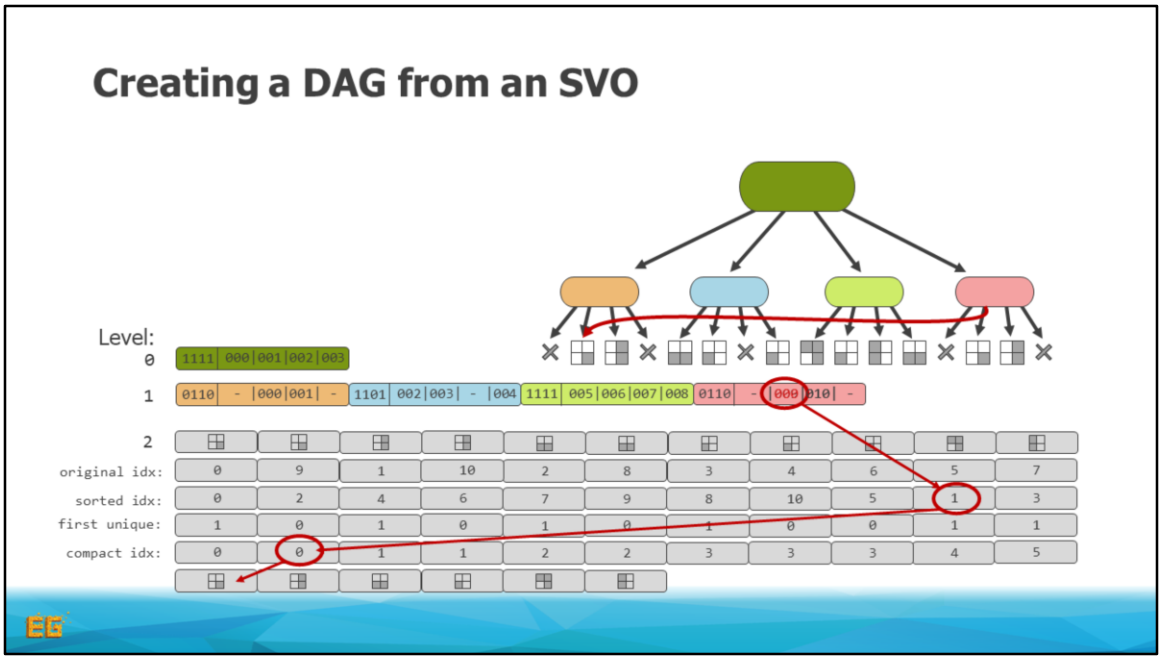
Next we run through all the leaf nodes again, and write a “1” if it is the first of its kind, and zero otherwise



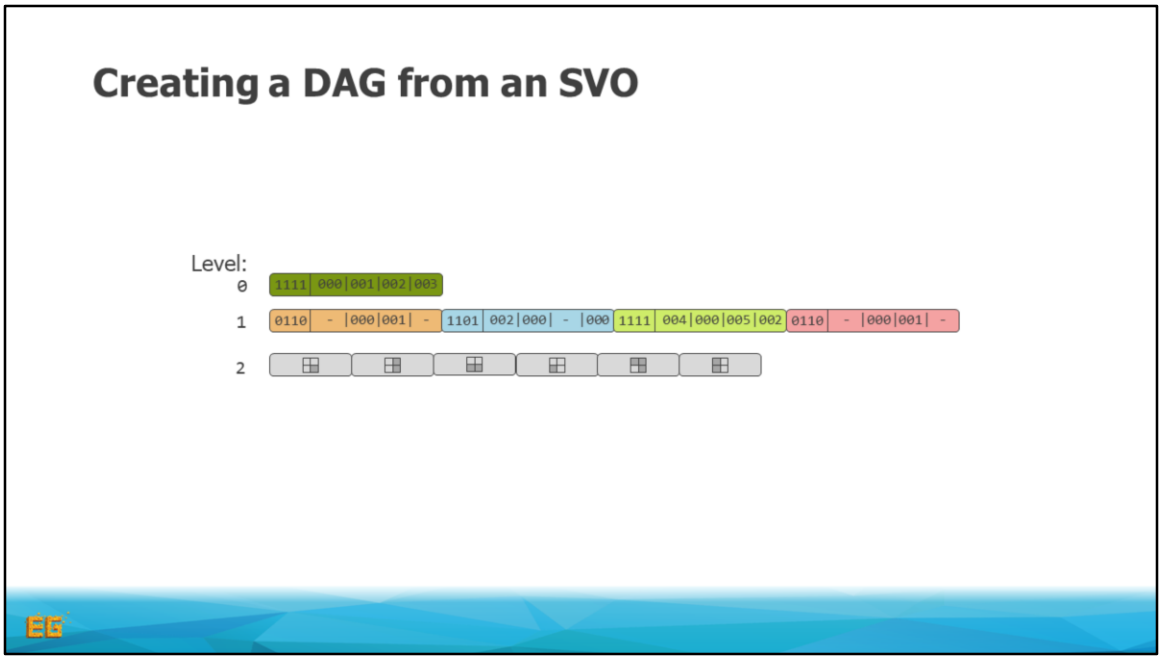
Then we calculate the prefix sum of this new array to get, for each leaf node, where the new index it should have within the leaf level



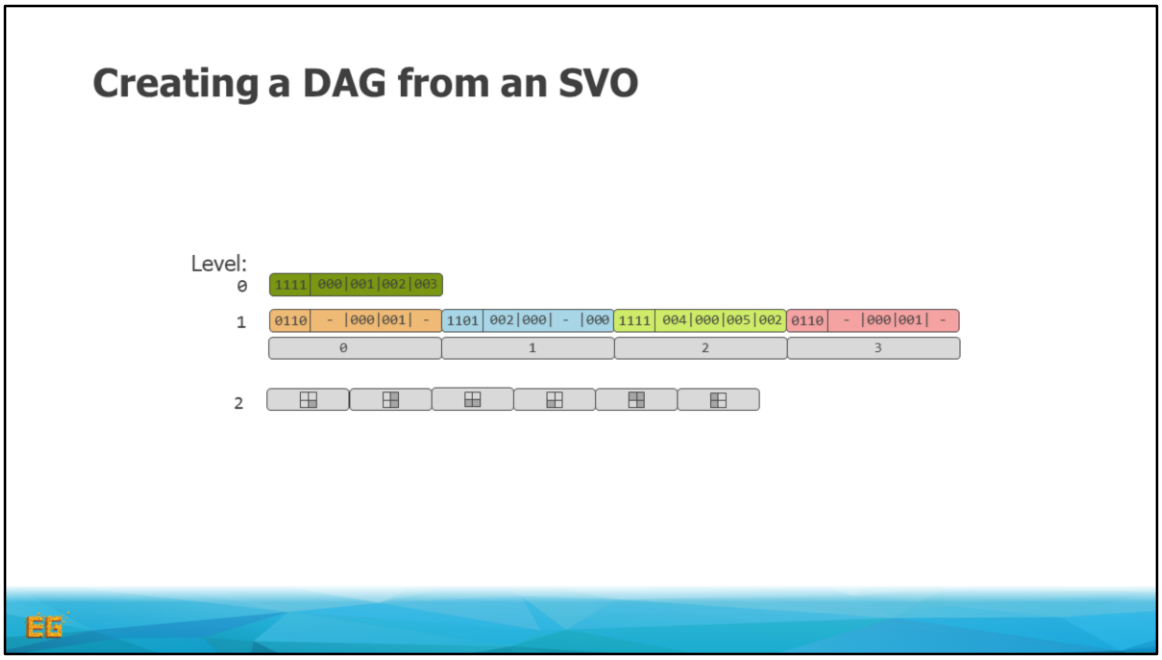
We run a stream compaction pass on the leaf nodes, to keep only the unique nodes



Finally, we run through all the parent pointers
<click>
We use the previous index to look up the index of the same node in the sorted list
<click>
We use *that* index to find the offset in the compact list
<click>
And so we update the pointer with that index

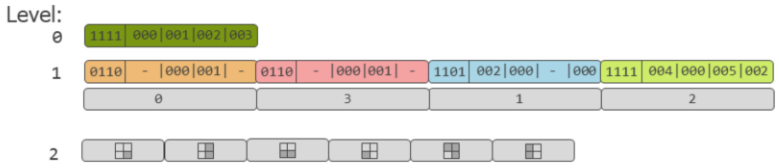


Now that we have reduced the leaf level, we can do excactly the same thing to level 1.

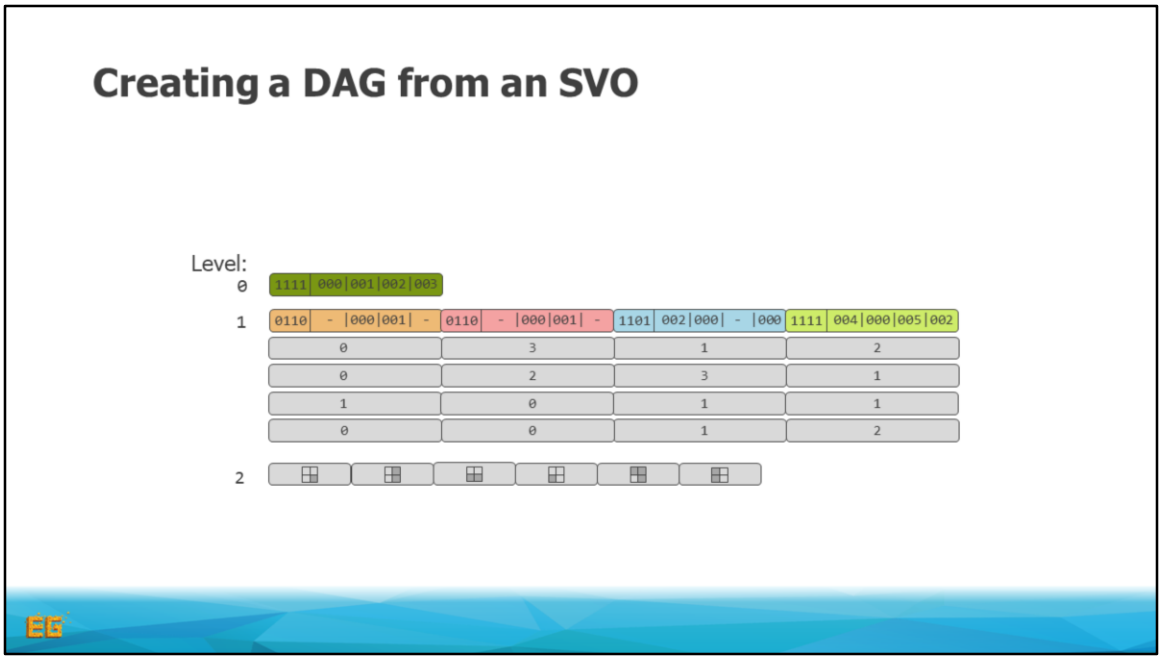


So we generate a list of indices

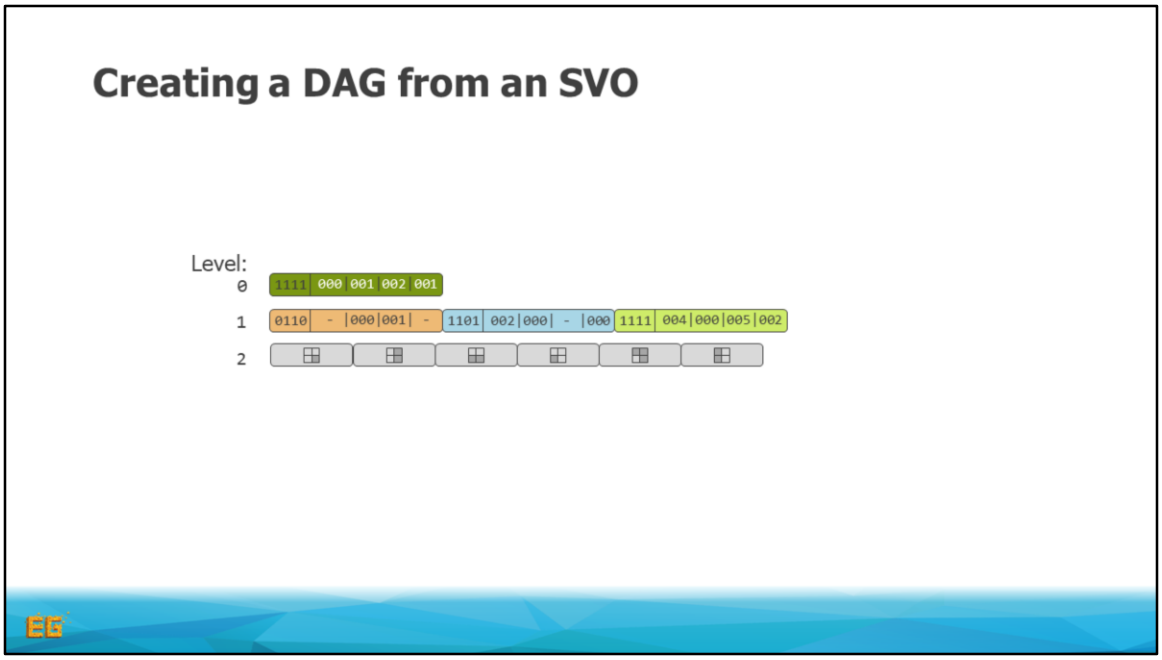
Creating a DAG from an SVO



We sort the nodes. Note that this time we use ALL THE POINTERS as our sorting key.



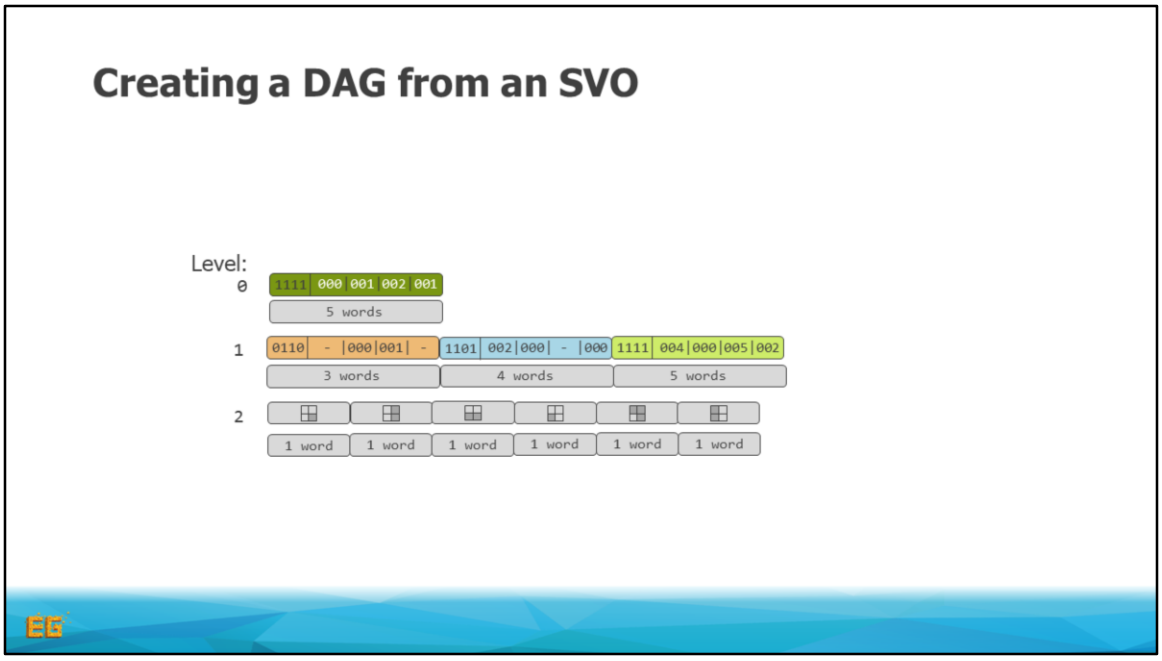
We generate the new index list, identify unique nodes and calculate the index in the compact list



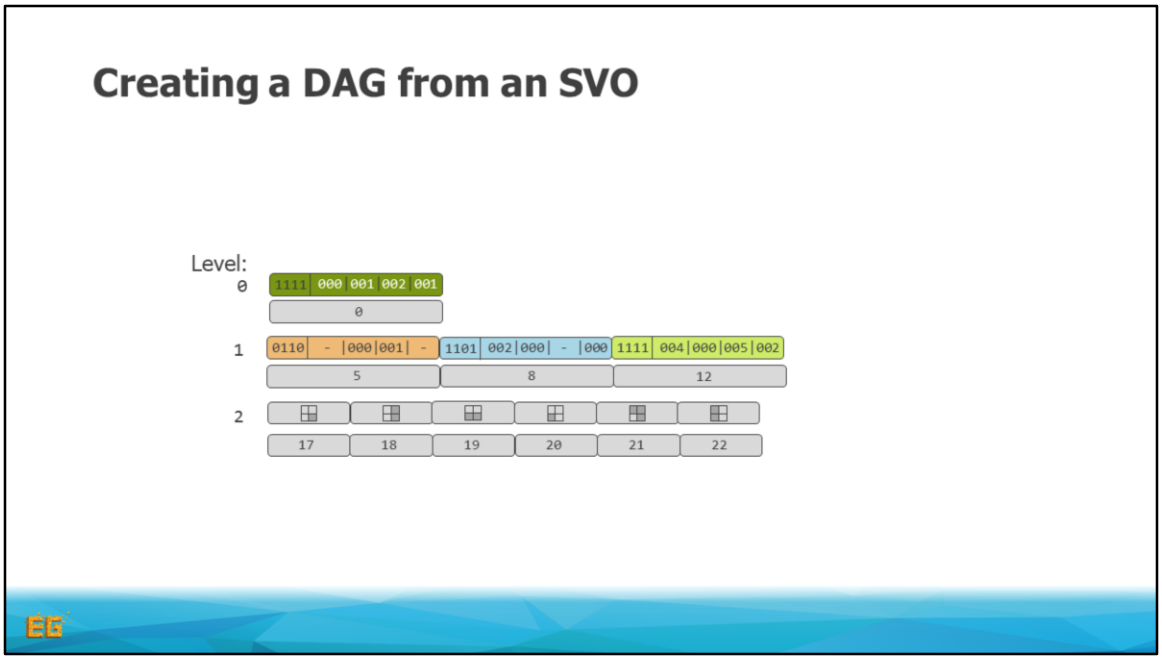
Then we stream compact the nodes in level 1 and update the pointers in level 0.

In a larger SVO we would keep doing this for all levels.

When we are done, we still have a DAG with lots of NULL pointers, and the levels do not lie in a compact array.

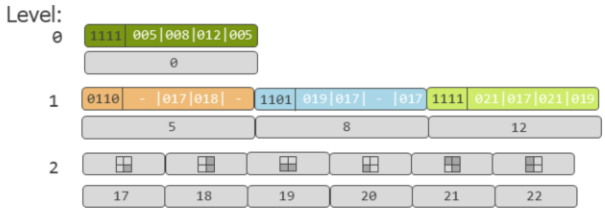


So the final step is to run through all the nodes and calculate their final size

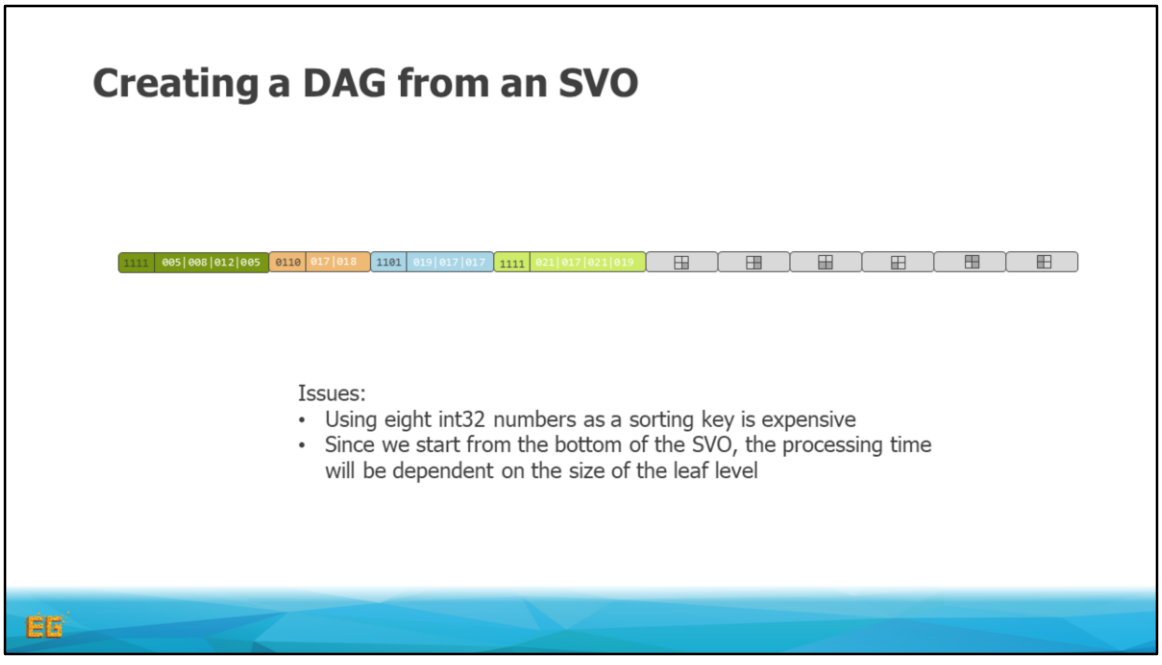


Run a prefix sum on this array to find the final index

Creating a DAG from an SVO



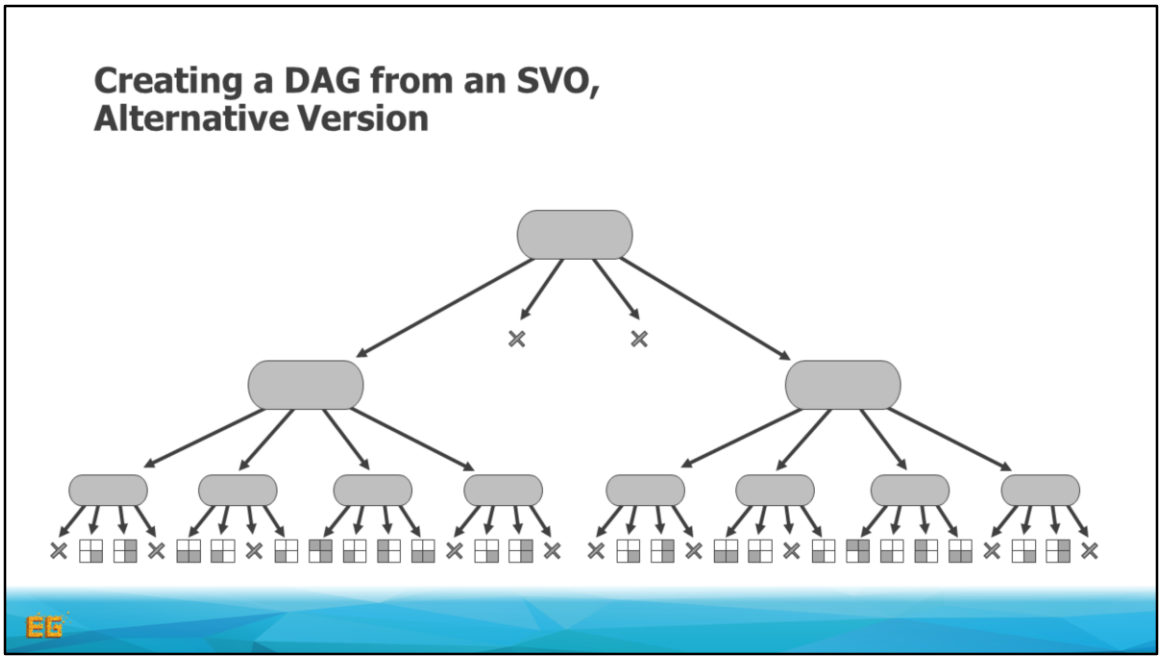
Update the pointers to the new indices



And copy the nodes to their new indices in a compact array.

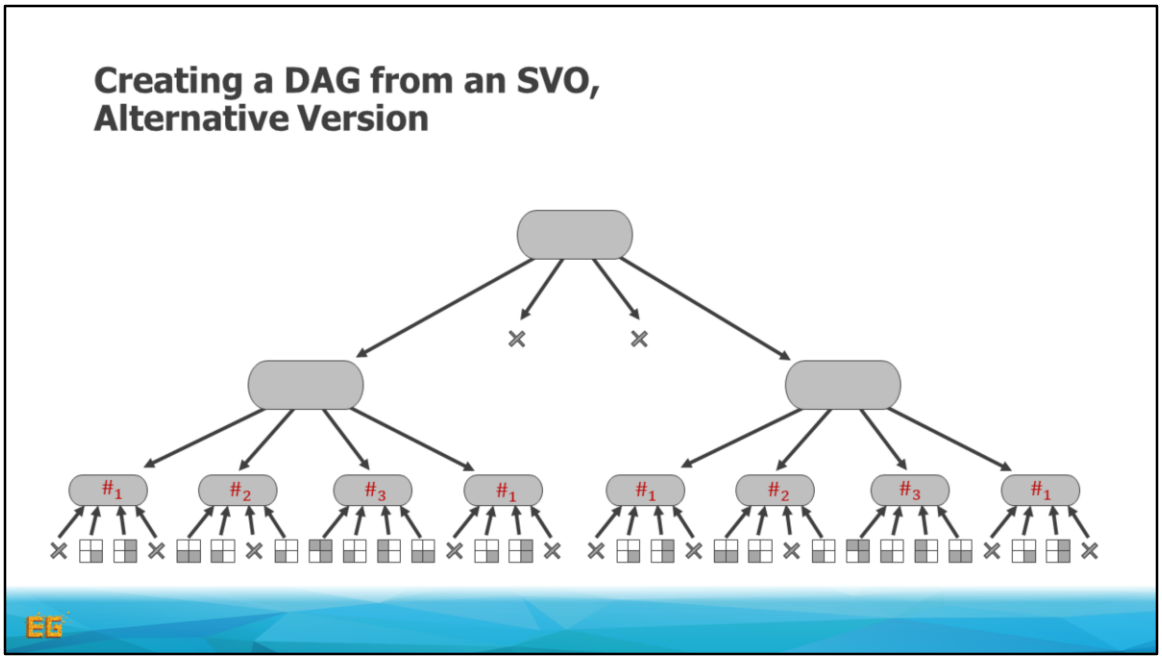
This algorithm works, and is really quite fast if implemented with cudpp/cub/thrust on the GPU.

But there are two annoying problems with it.

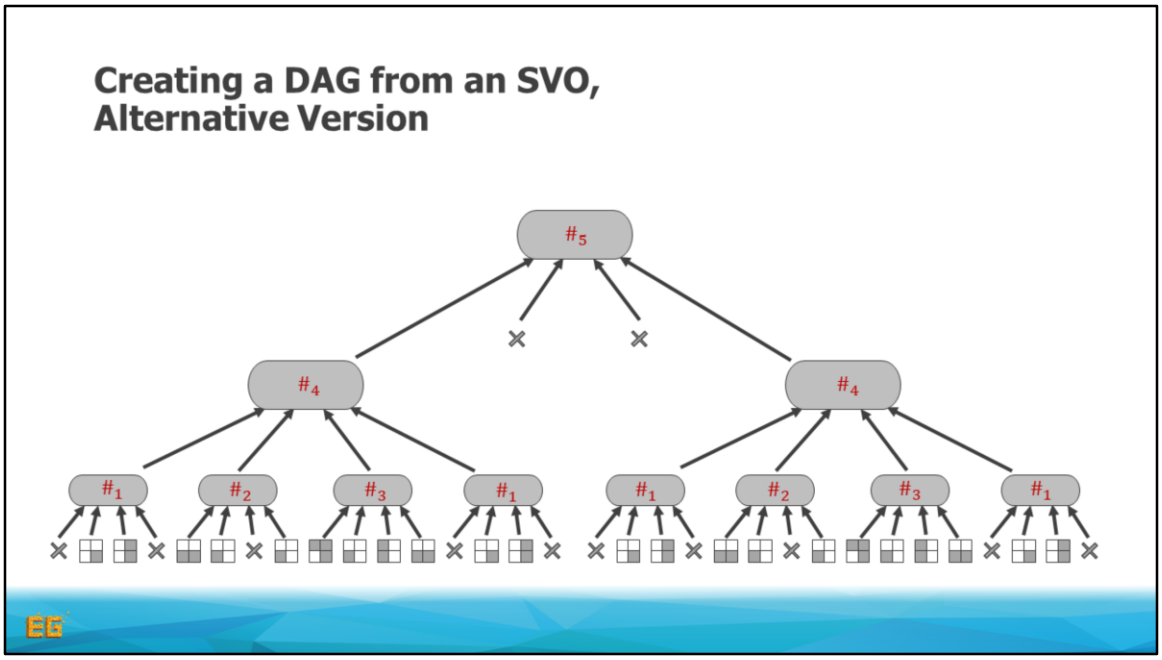


While writing a TVCG extension to our I3D paper on fast construction of precomputed voxelized shadows, we came up with a much simpler and faster method.

Let's say our original SVO looks like this.
We start at level 2, and calculate a *hash value* for each node, based on the child-masks of its contained leaf nodes. <click>



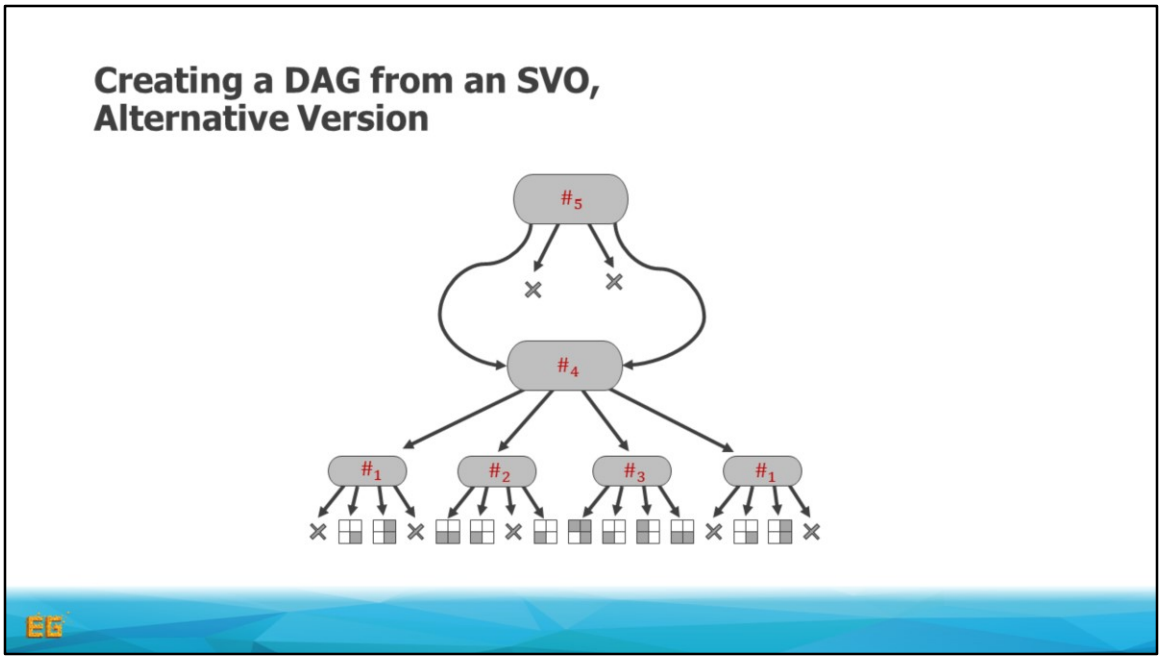
This is already interesting. Since the has value only depends on the childmasks, we can immediately see which subtrees are identical.
So, without ever having sorted the leaf level, we could remove the redundant nodes in level 1.



We contine to create hash values for the upper levels, always creating the hash based on the hashes of the child-nodes.

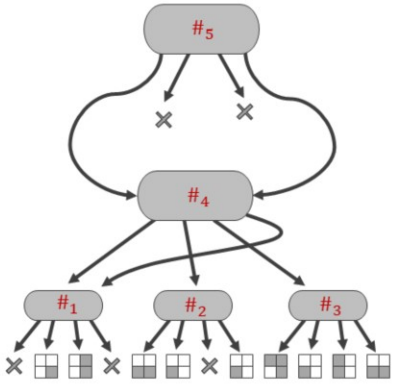
Now we can remove common sub trees with a top-down algorithm instead.

Starting at level 1, we sort the nodes, based on their hash, remove redundant nodes and update the parent pointers.

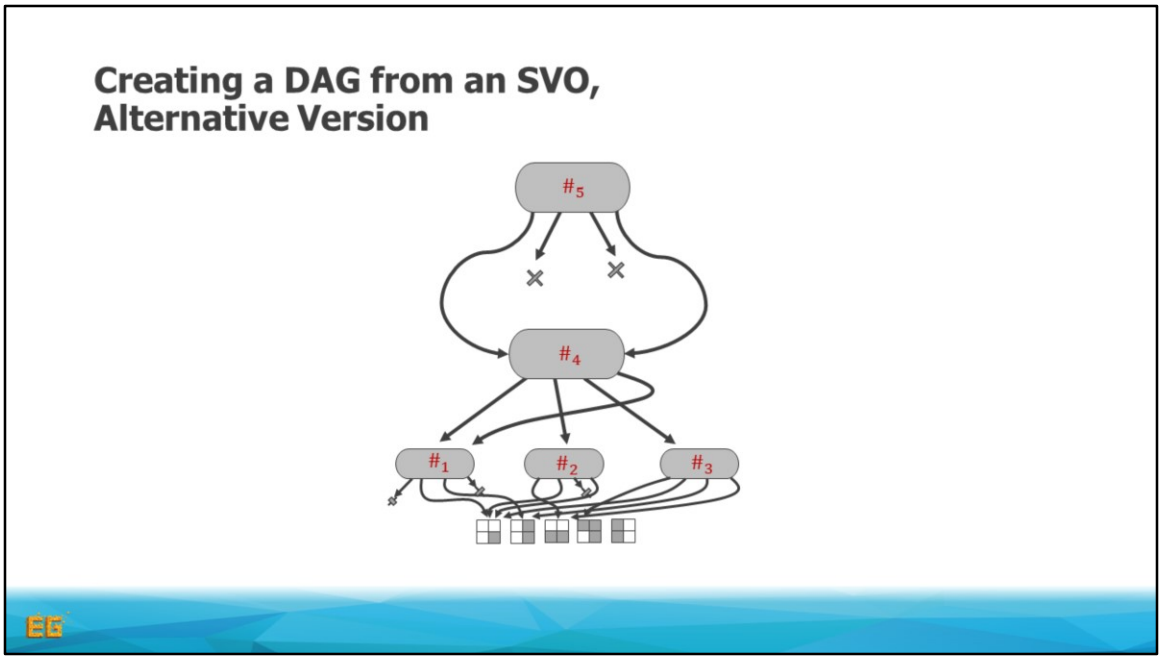


Now we move on to level 2, which is a lot smaller than it was before, sort that level, remove redundant nodes and update parent pointers…

Creating a DAG from an SVO, Alternative Version



That reduced the size of the leaf level, which we finally sort and reduce.



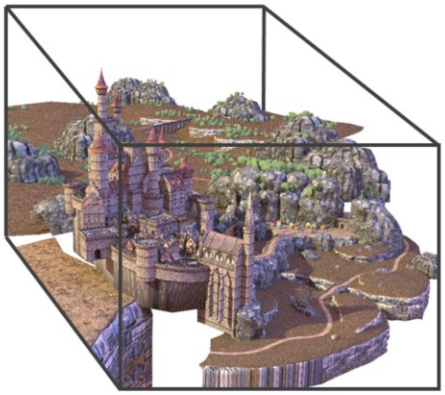
This algorithm can be much faster (was about 2x as fast in our shadow tests). Processing time will be proportional to the largest level *in the DAG* rather than in the SVO so it should scale much better.
Also, we always sort on the hash value, rather than the pointers, which is much better.

There *is* a theoretical possibility that hash values could collide, which would cause very weird result, but this risk goes towards zero as we increase the size of the hash value.

For us, 64 bits was sufficient to never experience any collisions.

Compressing Large DAGs

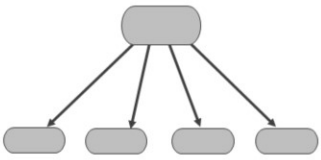
- Epic Citadel
- 128k x 128k x 128k voxels
- DAG size ~1GB
- Pointerless (!) SVO 5.1GB

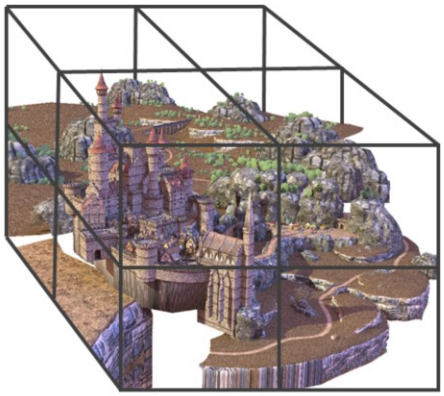


At high resolutions, fitting the original SVO on GPU memory can be impossible. Since the GPU based construction algorithm also requires a fair share of temporary memory, it is often necessary to split construction into subvolumes. I will describe that briefly now.

Compressing Large DAGs

- Create a dense top of the tree:

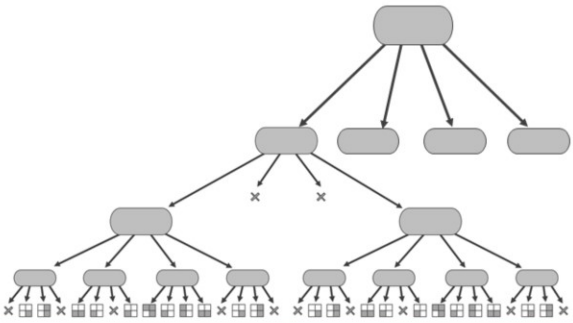


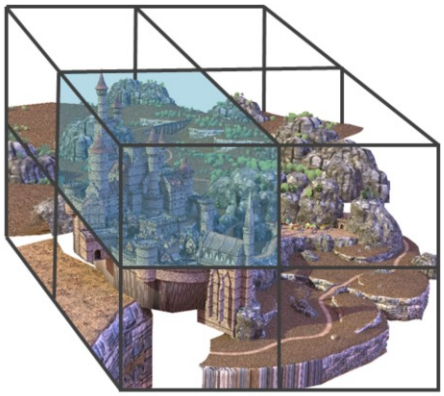


At high resolutions, fitting the original SVO on GPU memory can be impossible. Since the GPU based construction algorithm also requires a fair share of temporary memory, it is often necessary to split construction into subvolumes. I will describe that briefly now.

Compressing Large DAGs

- Voxelize the first child

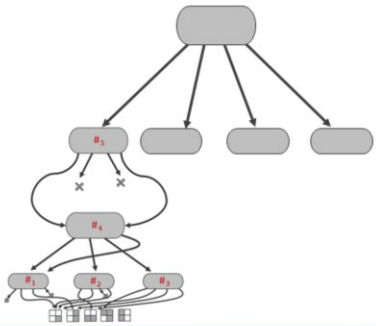


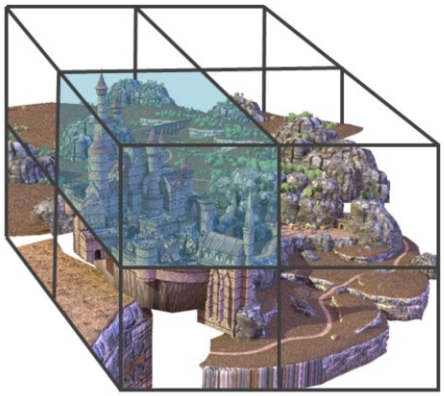


At high resolutions, fitting the original SVO on GPU memory can be impossible. Since the GPU based construction algorithm also requires a fair share of temporary memory, it is often necessary to split construction into subvolumes. I will describe that briefly now.

Compressing Large DAGs

- Create a DAG

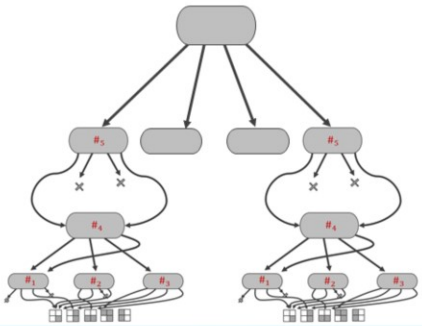


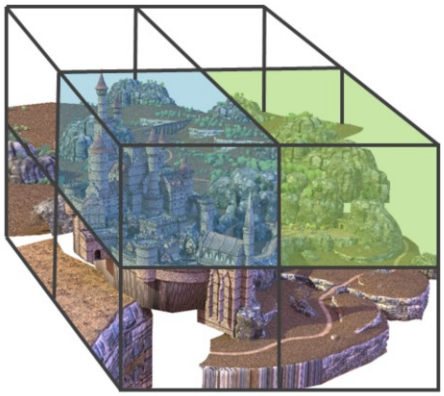


At high resolutions, fitting the original SVO on GPU memory can be impossible. Since the GPU based construction algorithm also requires a fair share of temporary memory, it is often necessary to split construction into subvolumes. I will describe that briefly now.

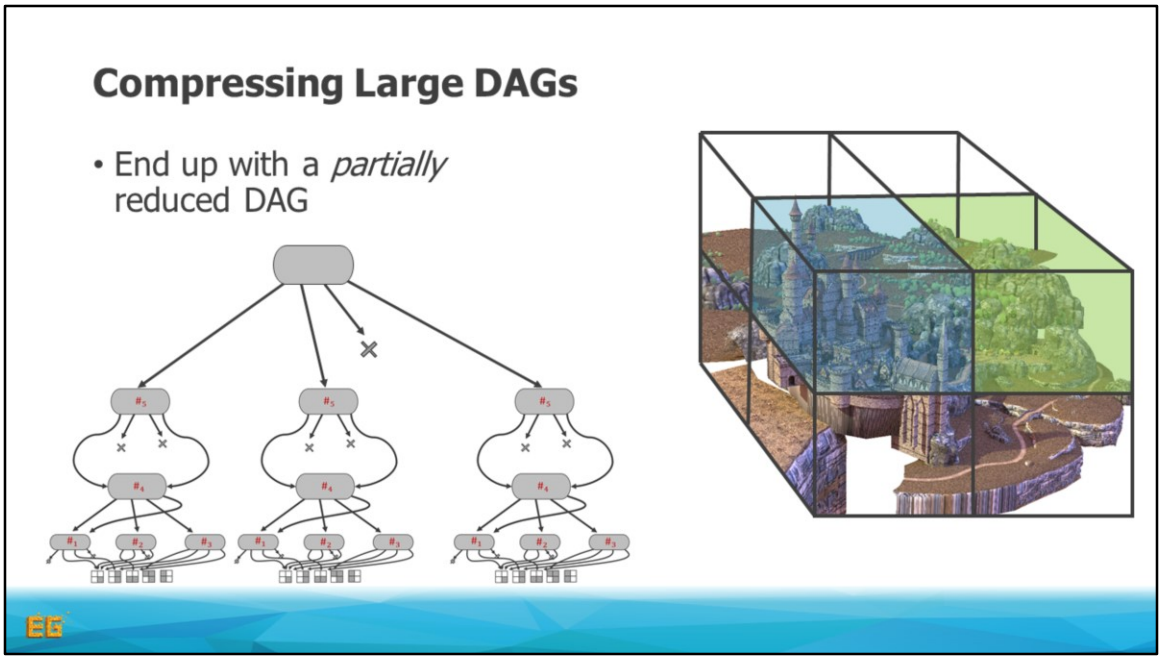
Compressing Large DAGs

- Voxelize and Compress a second child





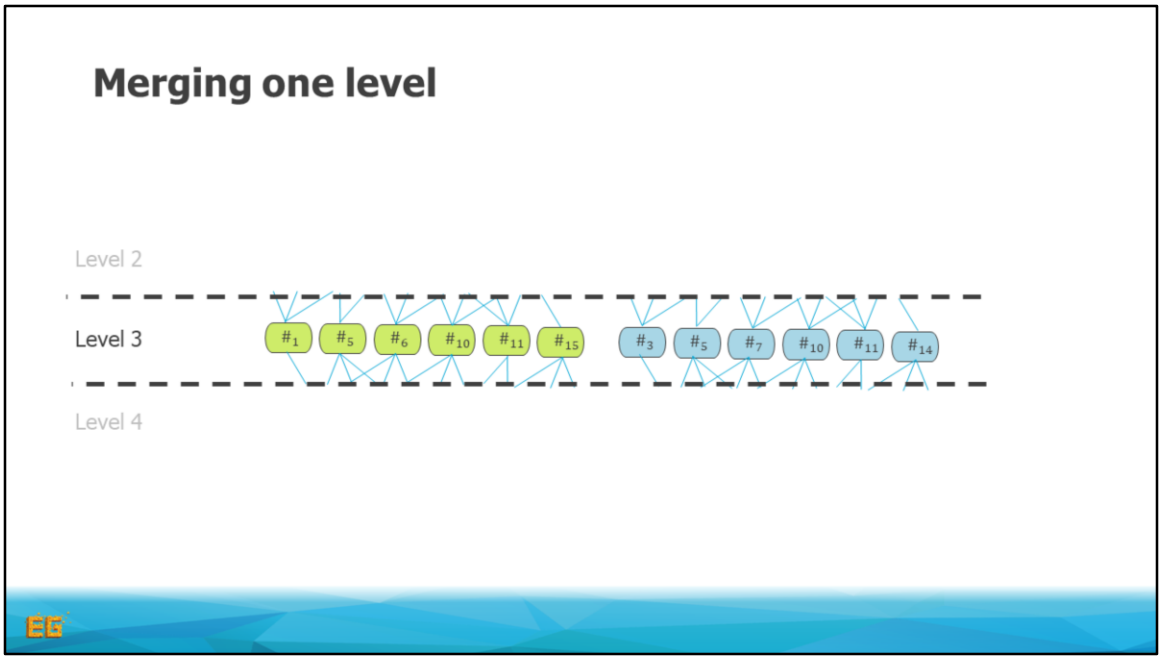
At high resolutions, fitting the original SVO on GPU memory can be impossible. Since the GPU based construction algorithm also requires a fair share of temporary memory, it is often necessary to split construction into subvolumes. I will describe that briefly now.



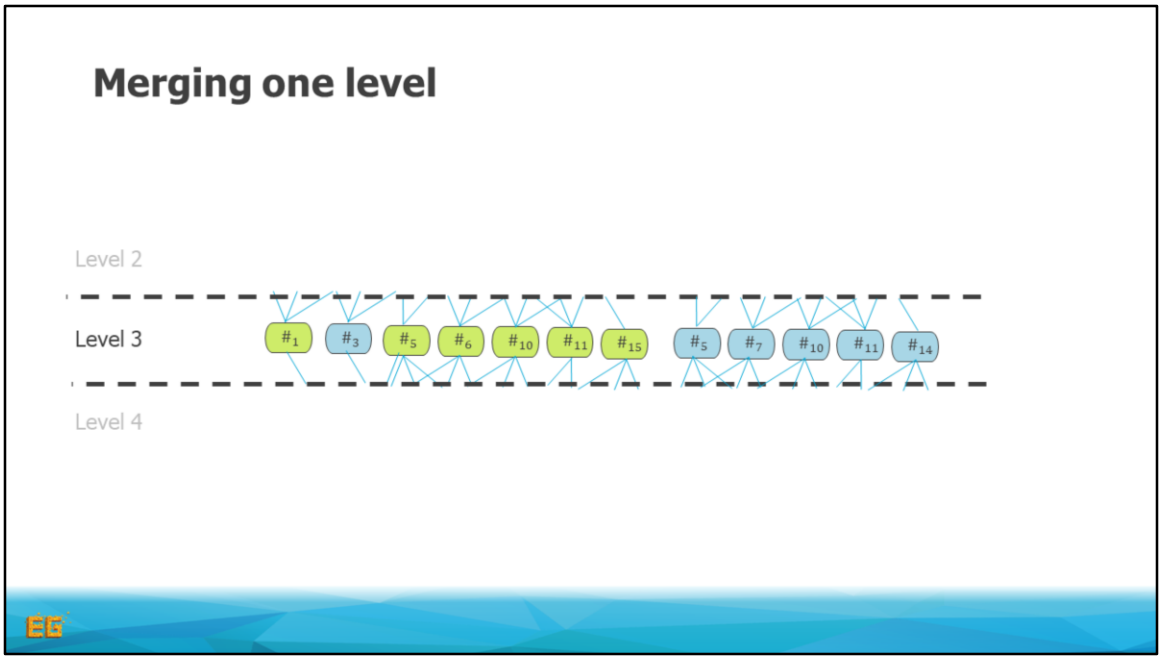
When we have done this for all subvolumes, we end up with a *partially* reduced DAG. That is, all of these subtrees are optimally compressed, but there may well be subtrees in this tree that are identical to subtrees in this tree.

We already have the hash values for all the nodes, so we CAN repeat the top-down algorithm exactly as before.

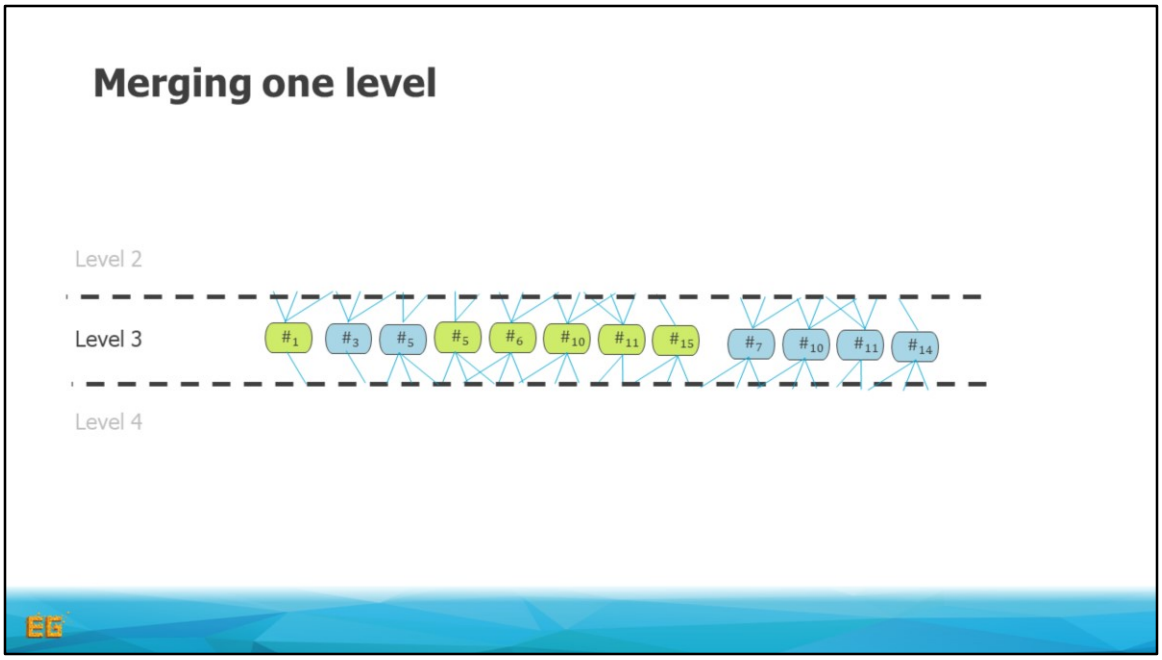
However, there is one trick that can speed things up significantly.



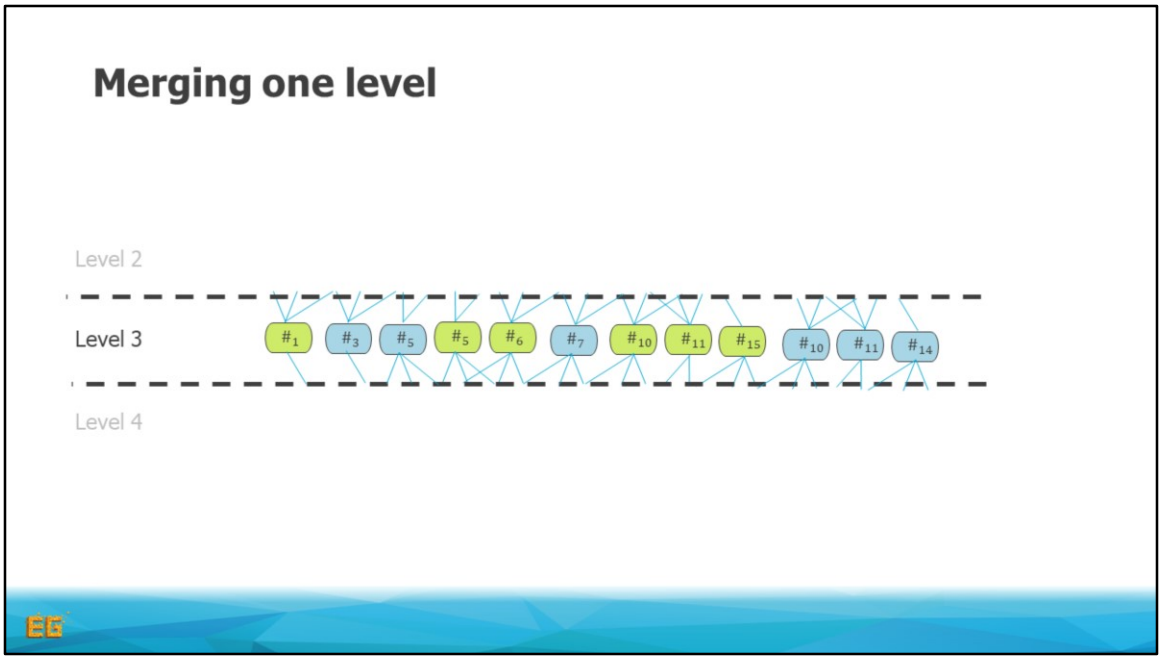
Since nodes in any level are already sorted, we do not have to perform a proper sort in our top down algorithm.
Simply merge the two levels in a linear sweep through the arrays.



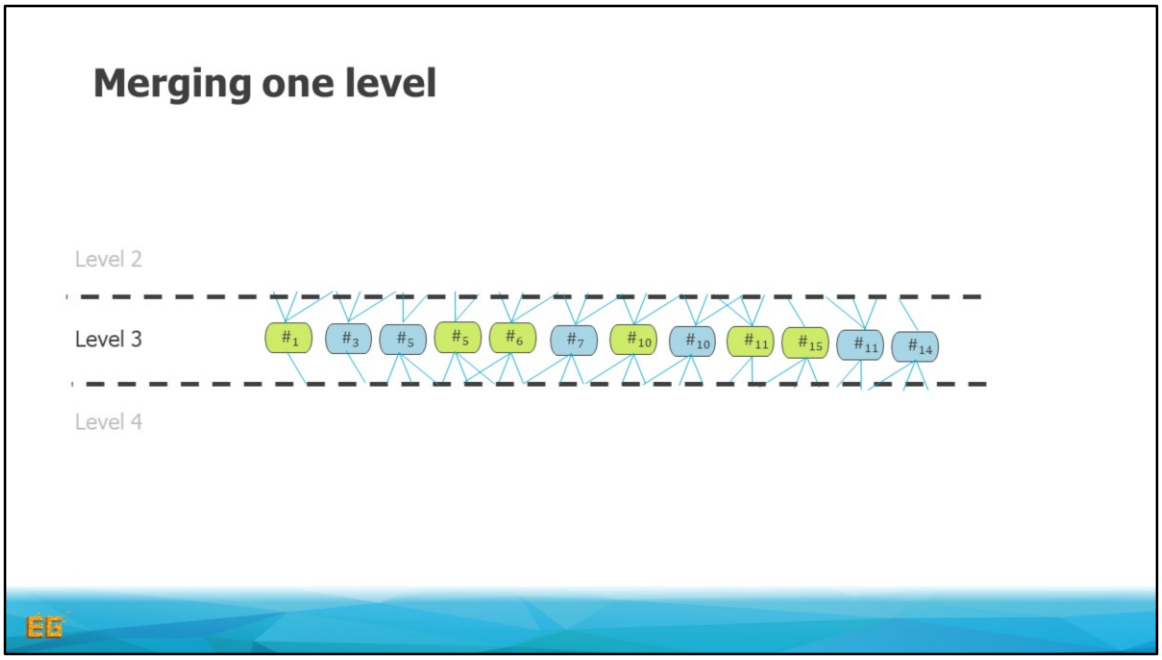
Since nodes in any level are already sorted, we do not have to perform a proper sort in our top down algorithm.
Simply merge the two levels in a linear sweep through the arrays.



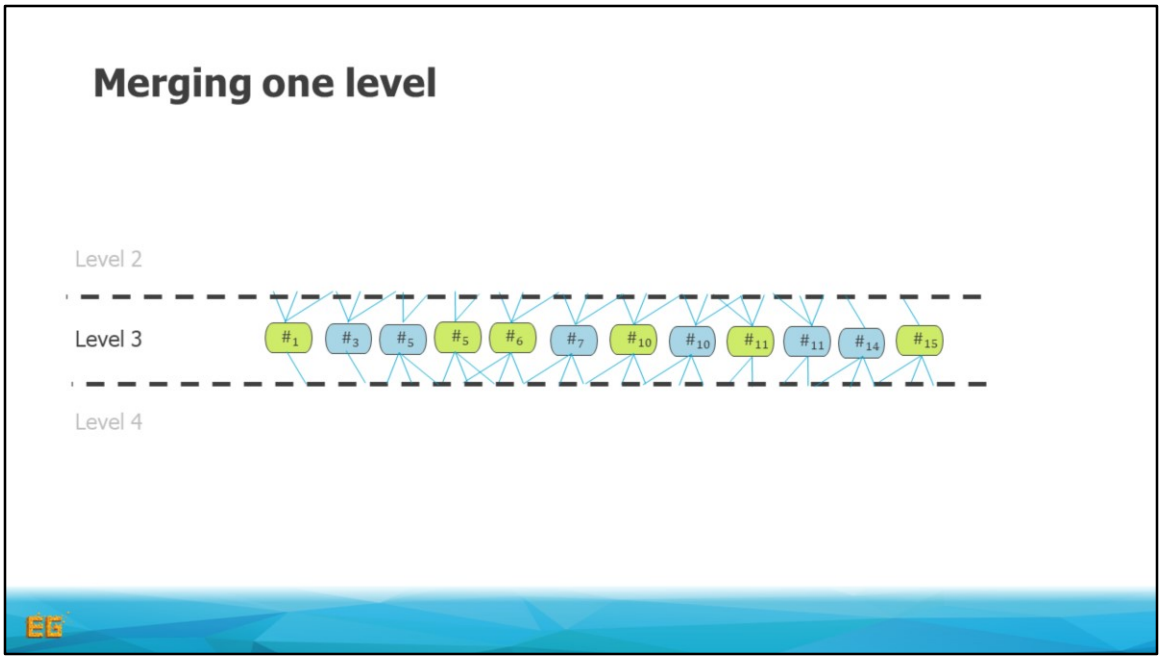
Since nodes in any level are already sorted, we do not have to perform a proper sort in our top down algorithm.
Simply merge the two levels in a linear sweep through the arrays.



Since nodes in any level are already sorted, we do not have to perform a proper sort in our top down algorithm.
Simply merge the two levels in a linear sweep through the arrays.



Since nodes in any level are already sorted, we do not have to perform a proper sort in our top down algorithm.
Simply merge the two levels in a linear sweep through the arrays.



Then, we simply remove redundant nodes and update the parent pointers as usual.