Voxel DAGs and Multiresolution Hierarchies
From Large-Scale Scenes to Pre-computed Shadows

Implementation: Ray-Tracing DAGs

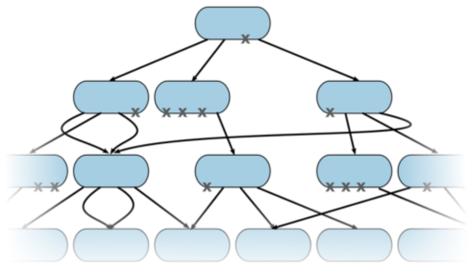
Markus Billeter[1]

[1] Delft University of Technology, The Netherlands

EUROGRAPHICS
2018

Previously.

- Have a DAG
- Now: ray-cast against it

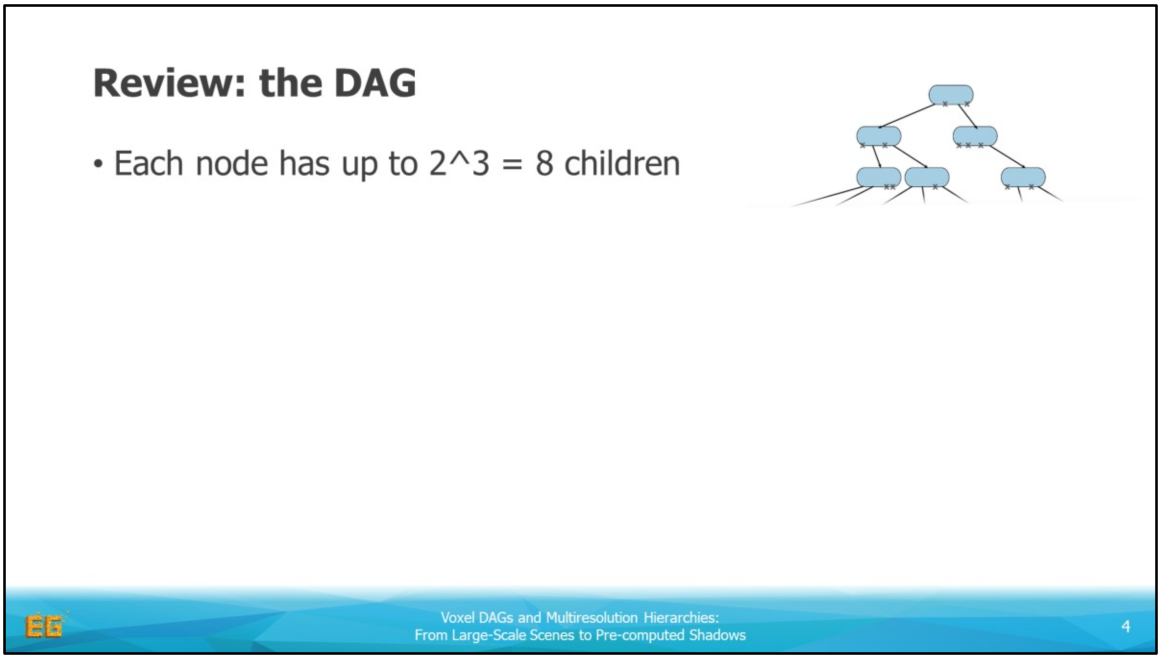


In the previous presentation, Erik Sintorn presented methods for practically constructing a DAG structure from a voxel data set. This presentation presents how such a DAG structure can be accessed immediately – without decompressing it first – for ray casting.

Goals

- Given a ray r(t) = x + t*d
- Find (first) intersection with a voxel
  - Position of intersection
  - Identity of intersected voxel

The setup is just like any other ray casting problem: Given a ray, the goal is to find its intersection(s). More specifically, we want to identify whether a ray intersects with the voxel geometry and if so, we want to retrieve the position of the intersection and the identity of the intersected voxel.

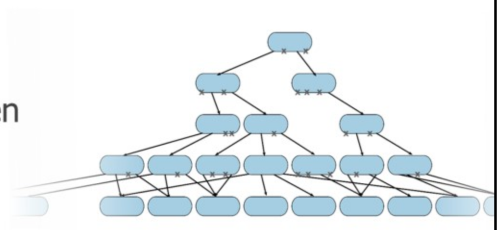


Let’s review the DAG structure. For the purpose of ray casting, we need to know the following:

- Each node has between one and eight children.

Review: the DAG

- Each node has up to 2^3 = 8 children



- Not a complete/balanced/… tree
- Use pointers internally

Let’s review the DAG structure. For the purpose of ray casting, we need to know the following:

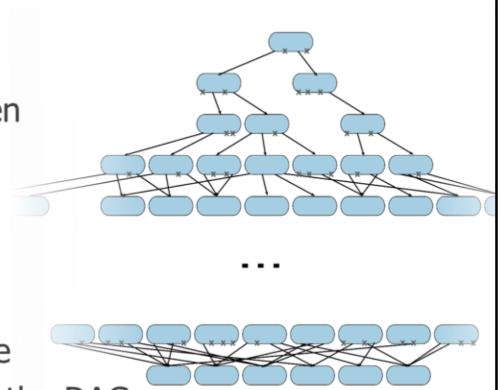
- *Each node has between one and eight children.*
- It is not a complete, balanced or otherwise “nice” tree …
- … and it uses pointers internally, which we will need to fetch and follow.

Review: the DAG

- Each node has up to 2^3 = 8 children

- Not a complete/balanced/… tree
- Use pointers internally

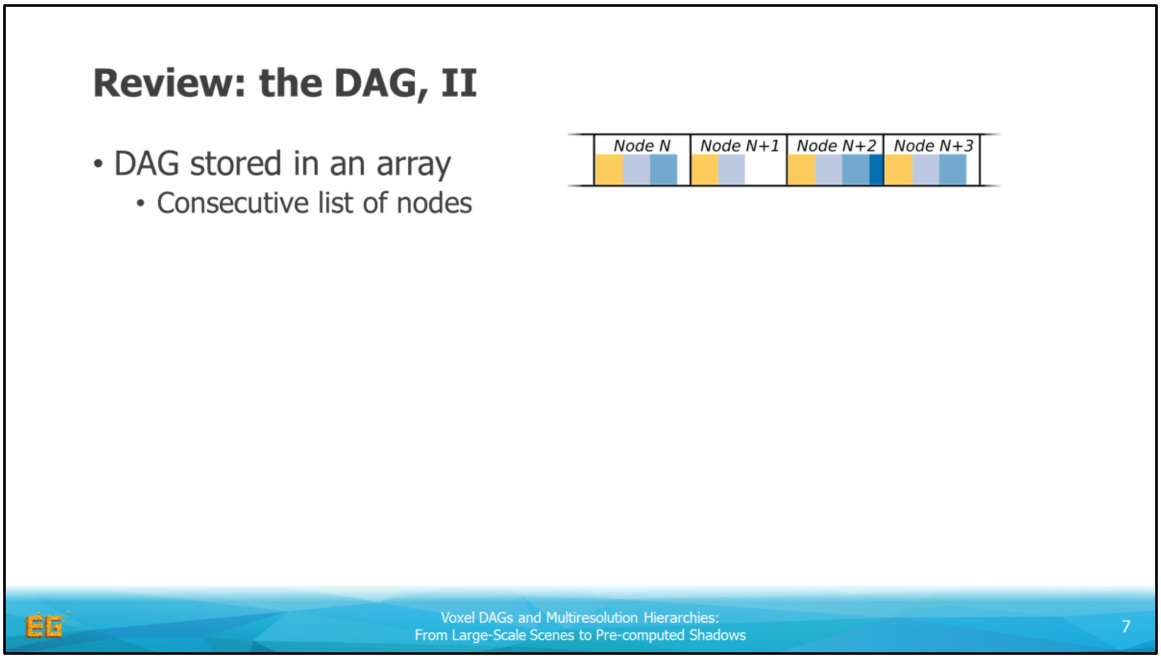
- Looks a lot like a Sparse Voxel Octree
- Except: can't store backward links in the DAG
  - Sub-trees can be reached from different parents



Let's review the DAG structure. For the purpose of ray casting, we need to know the following:

- *Each node has between one and eight children.*
- *It is not a complete, balanced or otherwise "nice" tree …*
- *… and it uses pointers internally, which we will need to fetch and follow.*

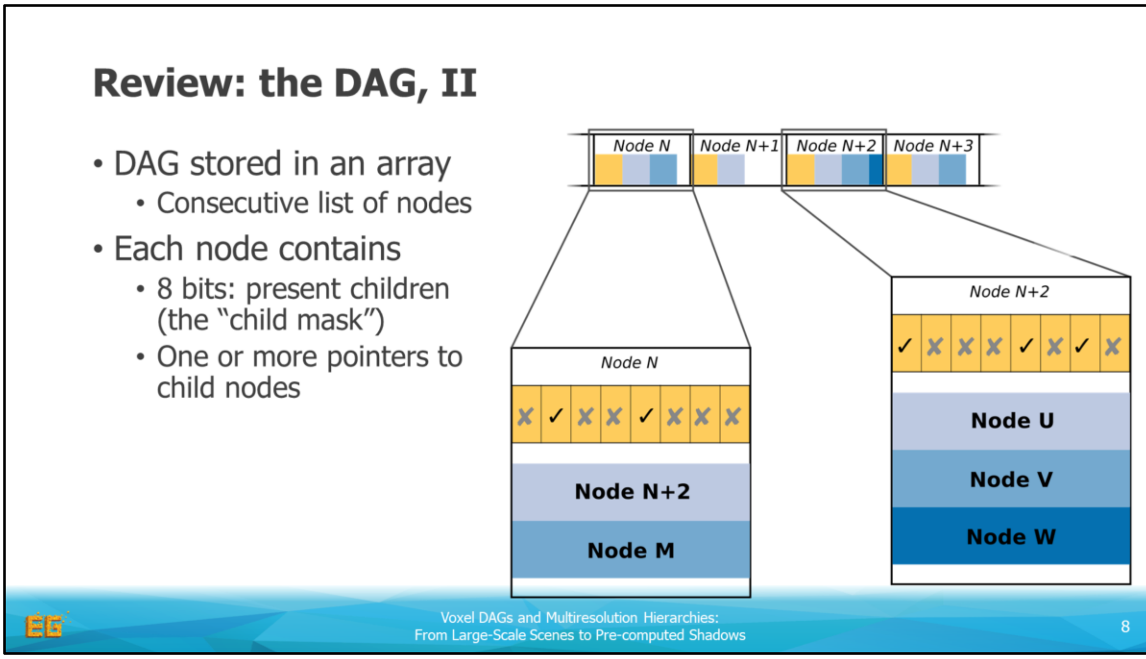
This is very similar to an ordinary sparse voxel octree with the exception that the DAG cannot store backwards links (i.e., links towards the root node). This is because, unlike the SVO, each sub-tree of the DAG can potentially be reached from multiple different sources. A second consequence of this is that up to eight distinct pointers are required per node, whereas a single pointer per node suffices for an SVO.



One simple way to store a DAG is to place all the nodes consecutively in an array.

Additional notes:
While simple, explicitly separating nodes from different levels into distinct arrays or ranges in a single array may be a good idea (this isn't done here, nodes can appear in any order). The current level is generally known during traversal, so there is little-to-no overhead for doing so. On the other hand, this enables a few different things: lower levels could be streamed in on demand. Since there are fewer nodes per level compared to the whole tree, it's also possible to have shorter child-pointers (see next slide) at the various levels.

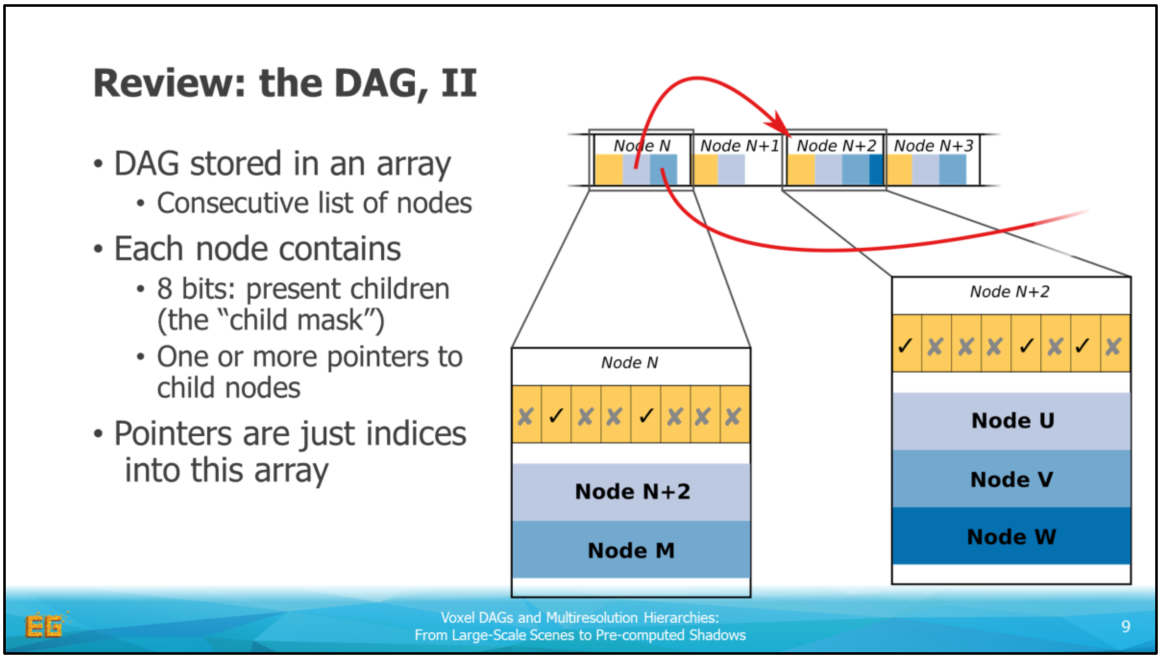
Nodes may be sorted arbitrarily either within the single array or within each level. For example if there is a well-known initial view of a scene, one could pack all nodes visible in this initial setup compactly at the beginning of the array/arrays, and initially only upload these to the GPU.



Each node contains two parts:

- The child mask, which describes which children are present for the current node. This is typically a bit mask with eight bits – one per potential child.
- One or more pointers to the child-nodes

The number of points is equal to the number of present children, i.e., it may vary between different nodes.



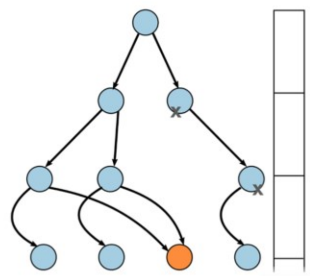
When storing the DAG in a single array, the pointers just become indices into this array. For example, we store the DAG in an array of 32 bit words. A pointer points to the word containing the child mask. The 8-bit child mask is padded to 32 bits, which seems wasteful on first glance, but the other 24 bits can be used to store additional information that applies to the whole subtree. Examples include opacity information (see later presentations on Shadows) or information required to access attributes.

Immediately following the child mask are the pointers. These are simply 32-bit (unsigned) integers containing the index of the word containing the corresponding child-node's child mask.

Using only 32-bit aligned 32-bit words makes reading data from the DAG memory much simpler/efficient.

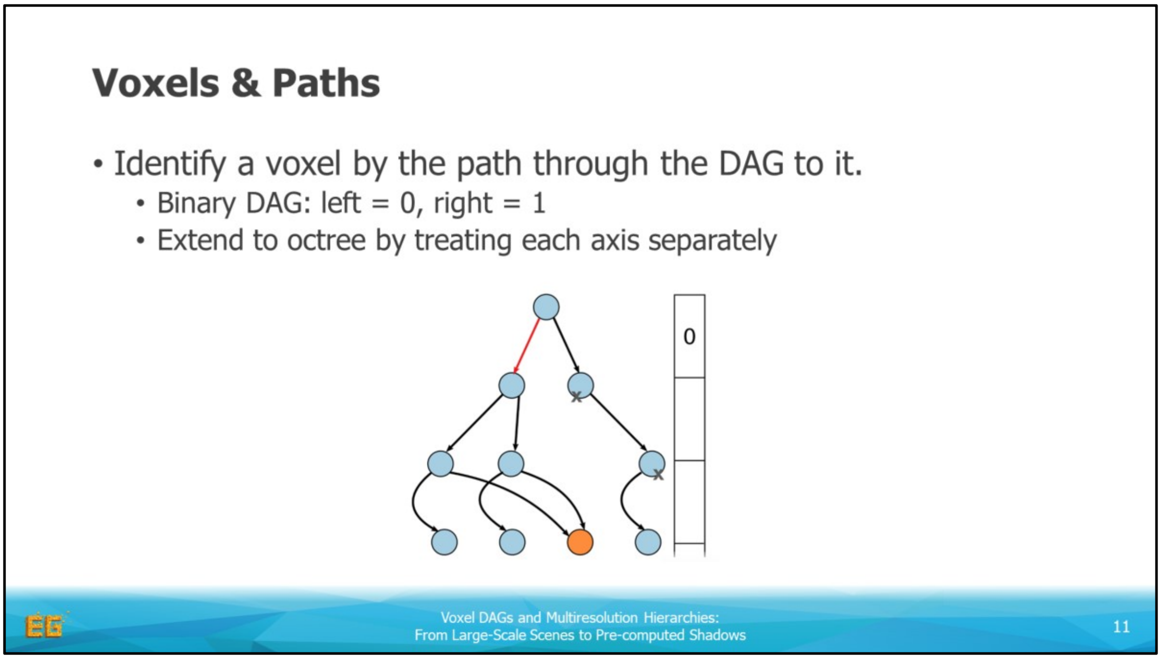
Voxels & Paths

- Identify a voxel by the path through the DAG to it.
  - Binary DAG: left = 0, right = 1
  - Extend to octree by treating each axis separately



As mentioned in the beginning, one an intersection has been found, we need to identify the corresponding voxel somehow. As we have seen in earlier presentations, the path through the DAG to the node uniquely identifies a leaf (or interior node).

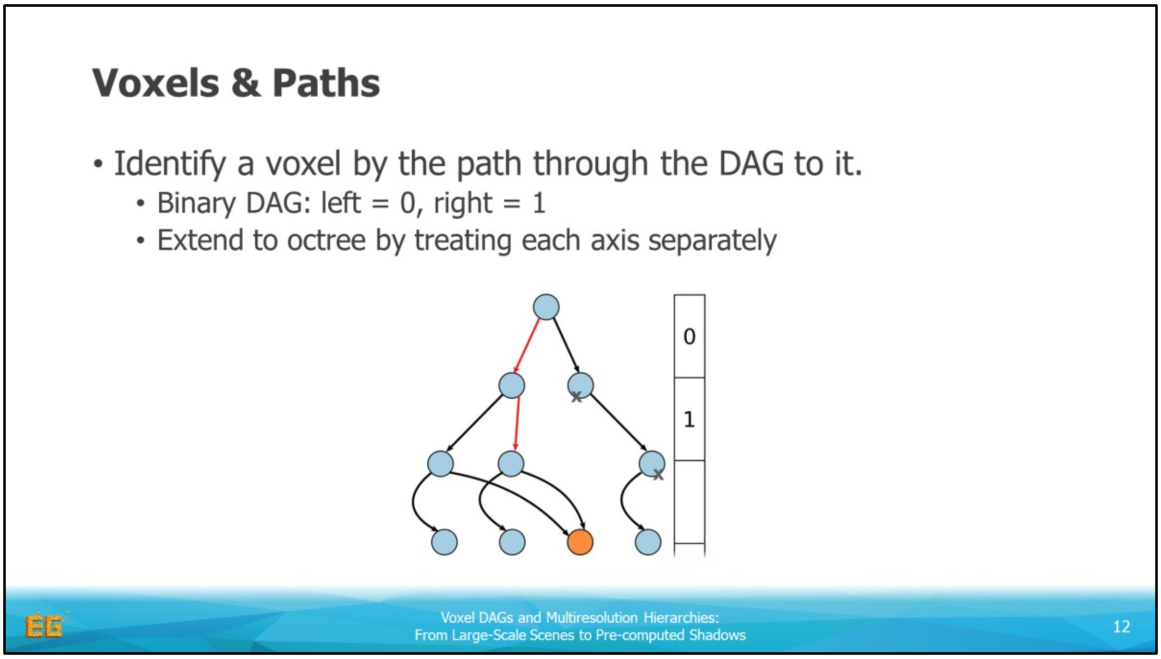
For example, in a binary DAG (as shown in the figure), at each level we may decide to either go left or right in the tree. We record either a zero or one at each step based on the decision.



As mentioned in the beginning, one an intersection has been found, we need to identify the corresponding voxel somehow. As we have seen in earlier presentations, the path through the DAG to the node uniquely identifies a leaf (or interior node).

For example, in a binary DAG (as shown in the figure), at each level we may decide to either go left or right in the tree. We record either a zero or one at each step based on the decision.

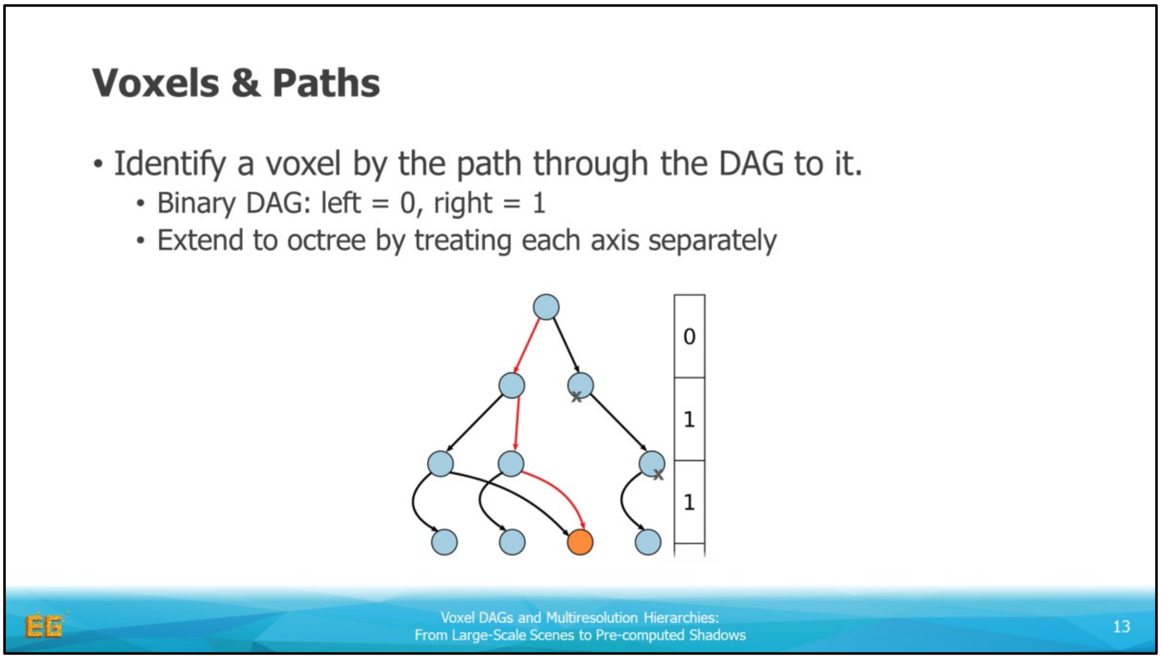
In this case, in order to reach the highlighted node, we first go left (=0) …



As mentioned in the beginning, one an intersection has been found, we need to identify the corresponding voxel somehow. As we have seen in earlier presentations, the path through the DAG to the node uniquely identifies a leaf (or interior node).

For example, in a binary DAG (as shown in the figure), at each level we may decide to either go left or right in the tree. We record either a zero or one at each step based on the decision.

In this case, in order to reach the highlighted node, we first go left (=0) … then right (=1)



As mentioned in the beginning, one an intersection has been found, we need to identify the corresponding voxel somehow. As we have seen in earlier presentations, the path through the DAG to the node uniquely identifies a leaf (or interior node).

For example, in a binary DAG (as shown in the figure), at each level we may decide to either go left or right in the tree. We record either a zero or one at each step based on the decision.

In this case, in order to reach the highlighted node, we first go left (=0) … then right (=1) and then right (=1) again, giving us the final path of 011.

To extend this to a octree (or quadtree), we simply treat each axis separately…

Voxels & Paths, II

- End up with three bit strings
  - One for each axis

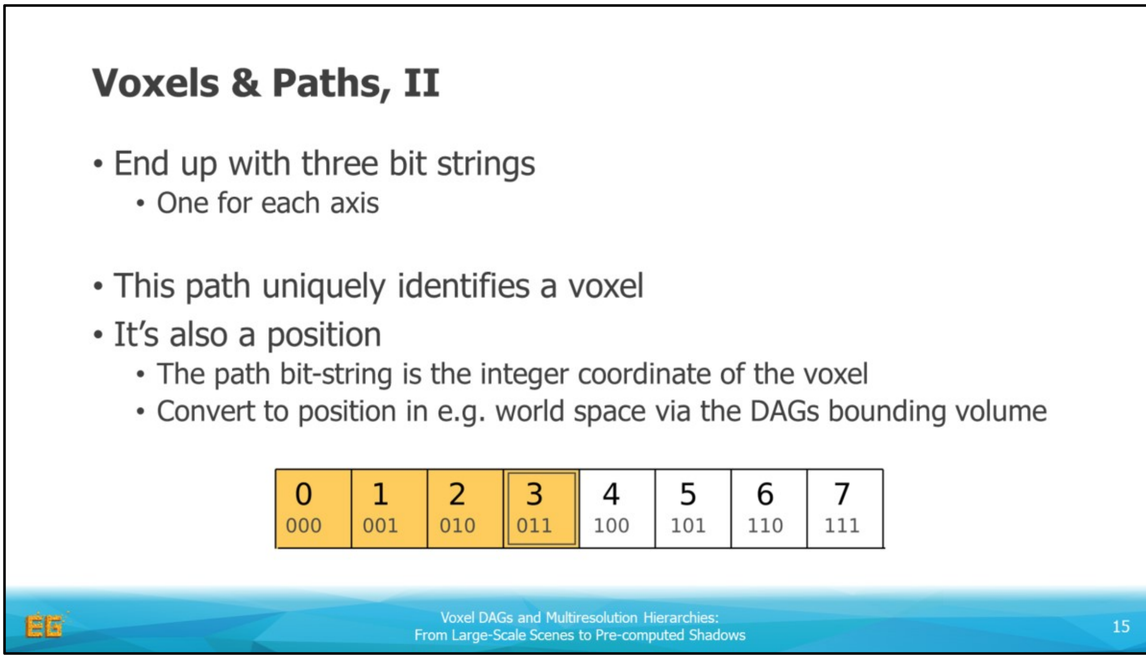
- This path uniquely identifies a voxel
- It's also a position
  - The path bit-string is the integer coordinate of the voxel
  - Convert to position in e.g. world space via the DAGs bounding volume

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

… so, for the voxel DAG we end up with tree bit strings, one for each axis.

These uniquely identify voxel. But they are also equivalent to a position.

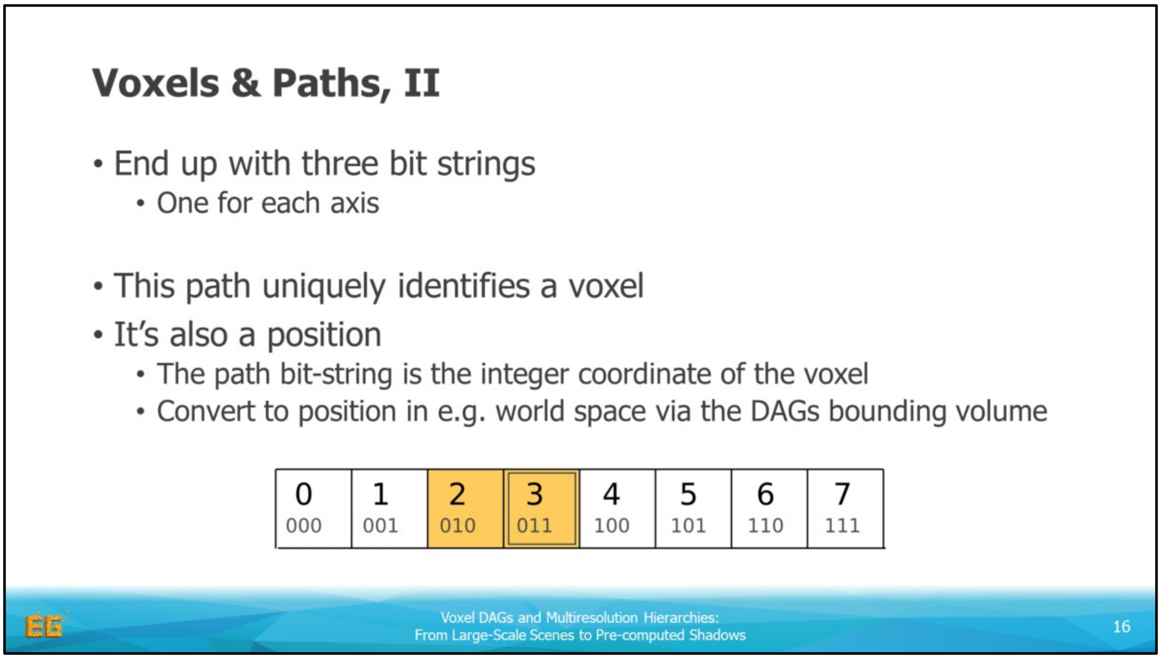
If we interpret the bit string as an integer, it corresponds to the integer position of the voxel on the corresponding axis…



… so, for the voxel DAG we end up with tree bit strings, one for each axis.

These uniquely identify voxel. But they are also equivalent to a position.

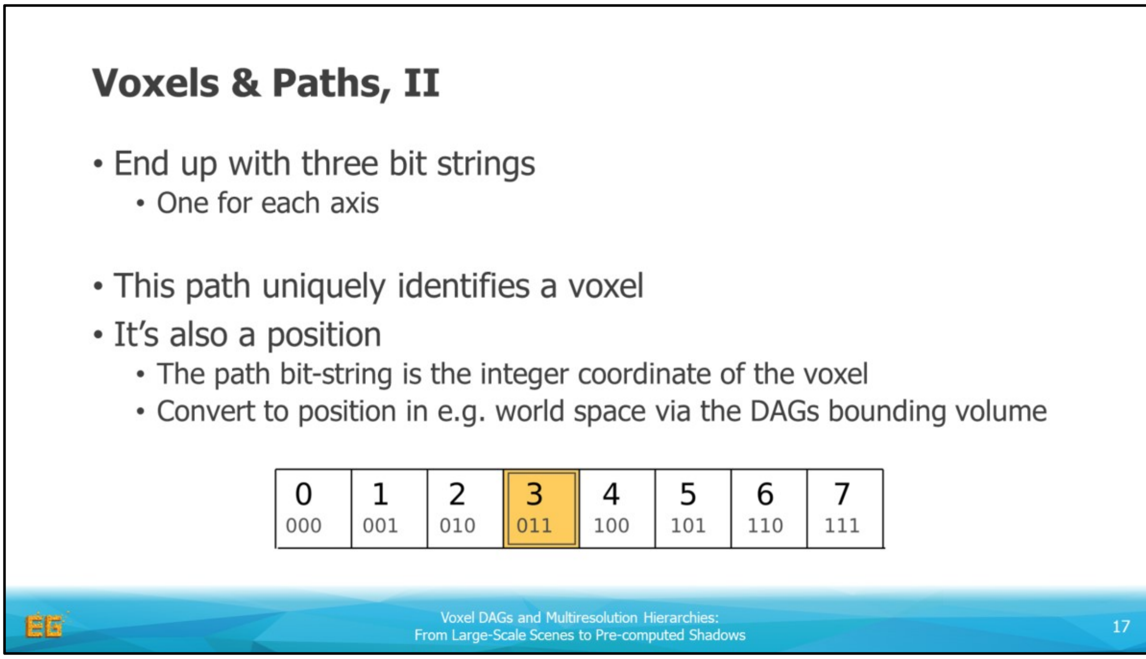
If we interpret the bit string as an integer, it corresponds to the integer position of the voxel on the corresponding axis…



… so, for the voxel DAG we end up with tree bit strings, one for each axis.

These uniquely identify voxel. But they are also equivalent to a position.

If we interpret the bit string as an integer, it corresponds to the integer position of the voxel on the corresponding axis…



… so, for the voxel DAG we end up with tree bit strings, one for each axis.

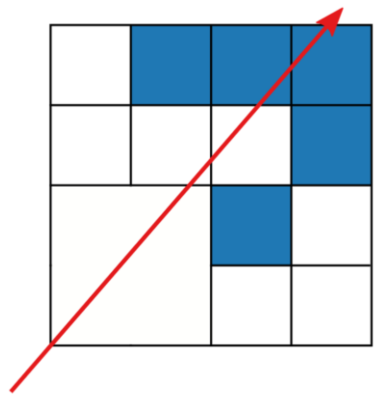
These uniquely identify voxel. But they are also equivalent to a position.

If we interpret the bit string as an integer, it corresponds to the integer position of the voxel on the corresponding axis…

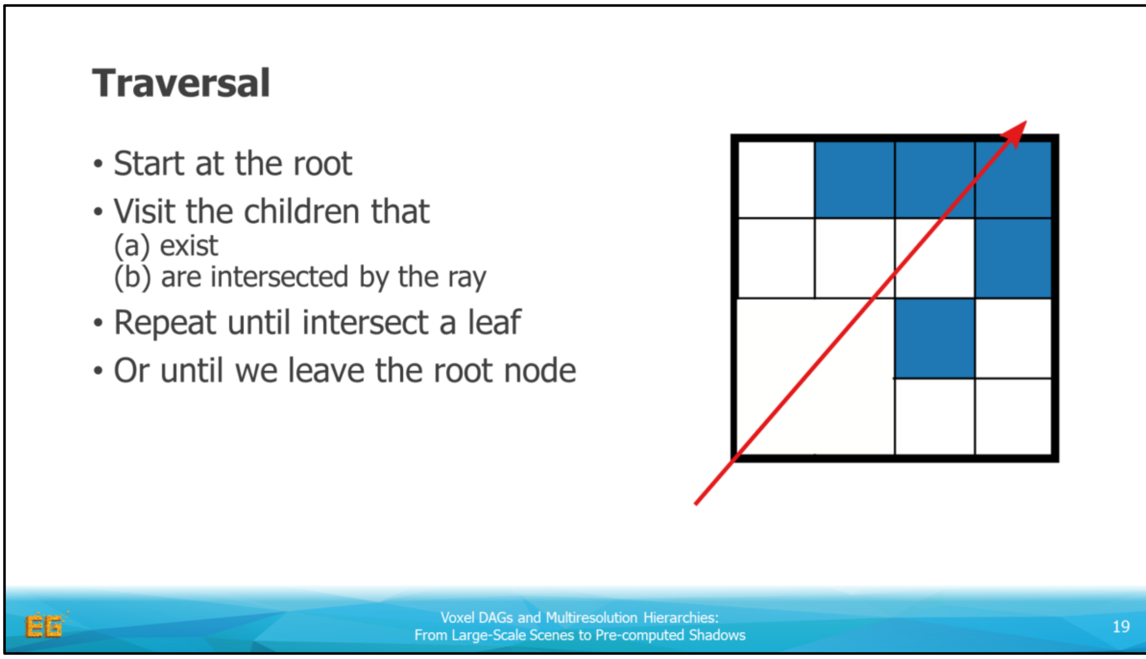
… We can obviously convert the position to any other space if we know the DAGs root bounding box/volume.

Traversal

- Start at the root
- Visit the children that
  (a) exist
  (b) are intersected by the ray
- Repeat until intersect a leaf
- Or until we leave the root node



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

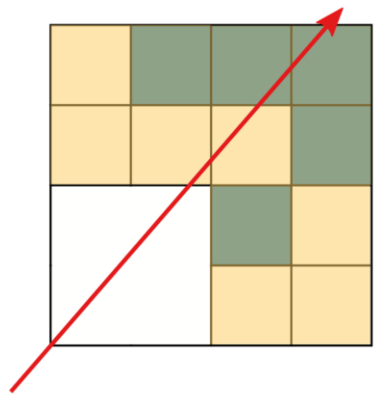


Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- We start at the root node

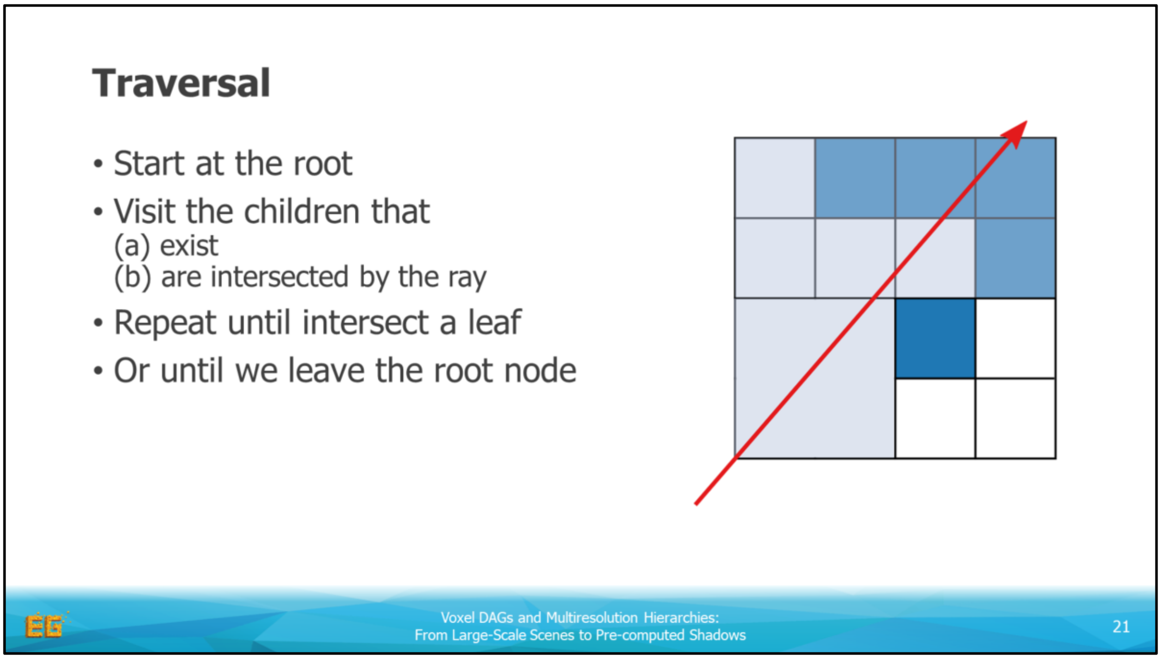
Traversal

- Start at the root
- Visit the children that
  (a) exist
  (b) are intersected by the ray
- Repeat until intersect a leaf
- Or until we leave the root node



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- … and we want to visit the children that (a) exist …

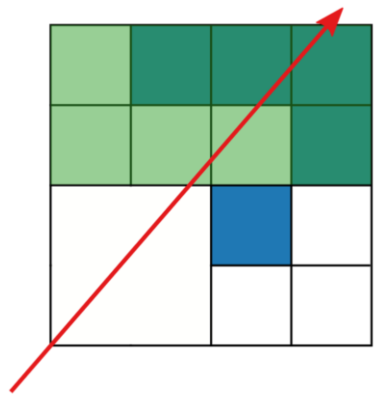


Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- … and (b) intersect the ray

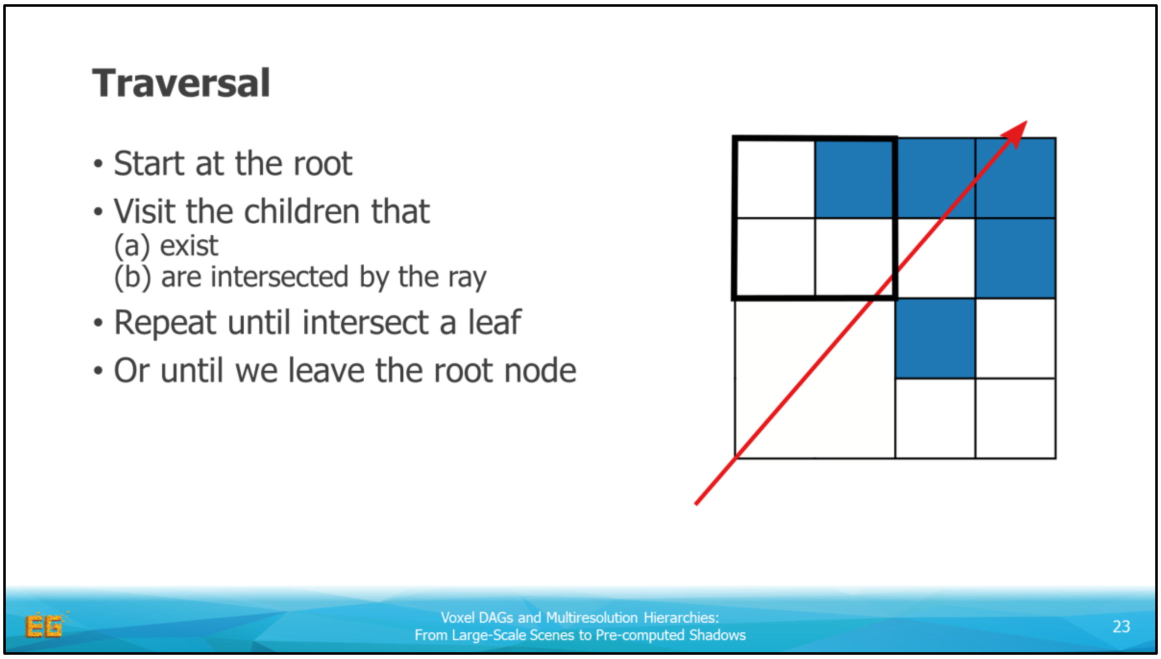
Traversal

- Start at the root
- Visit the children that
  (a) exist
  (b) are intersected by the ray
- Repeat until intersect a leaf
- Or until we leave the root node



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

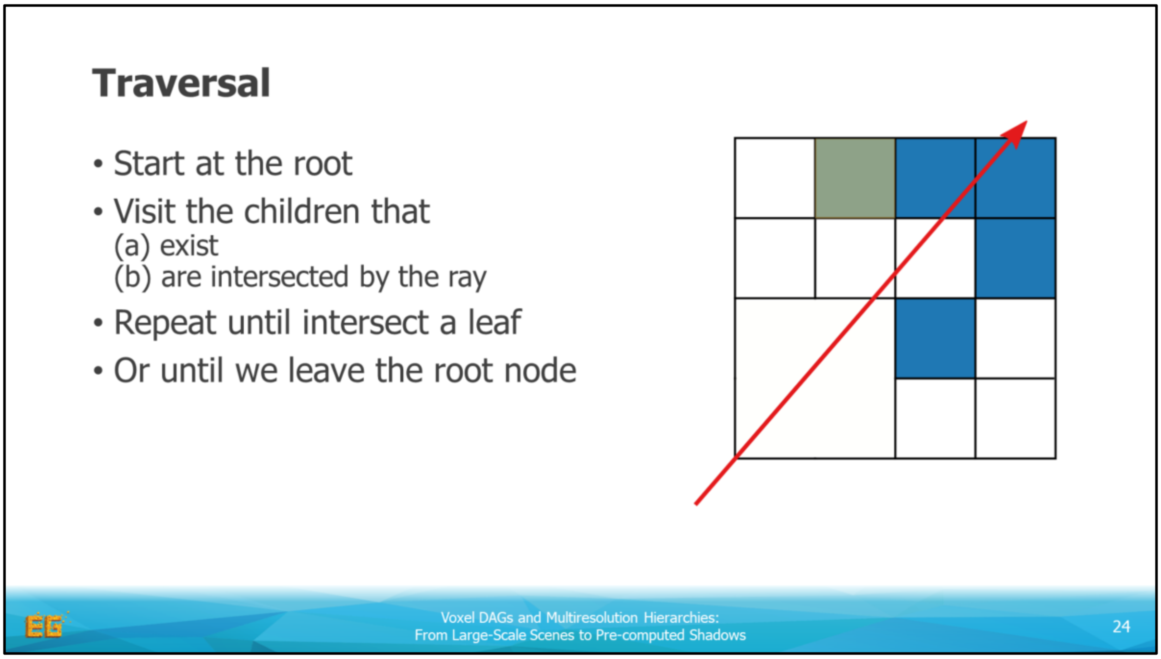
- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

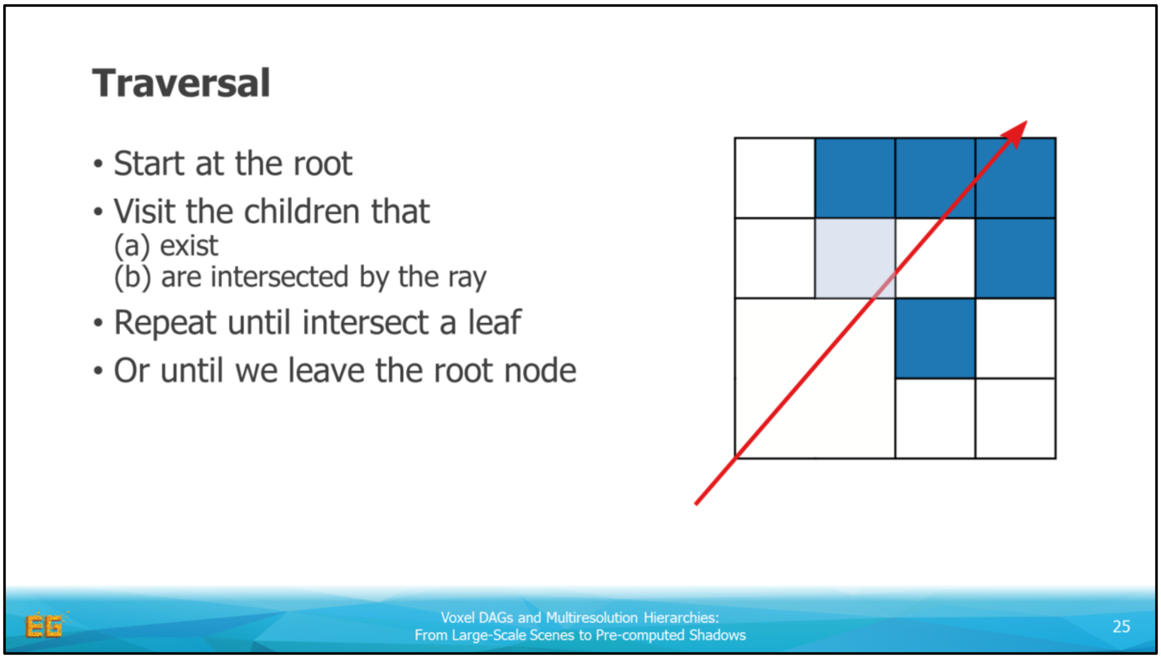
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

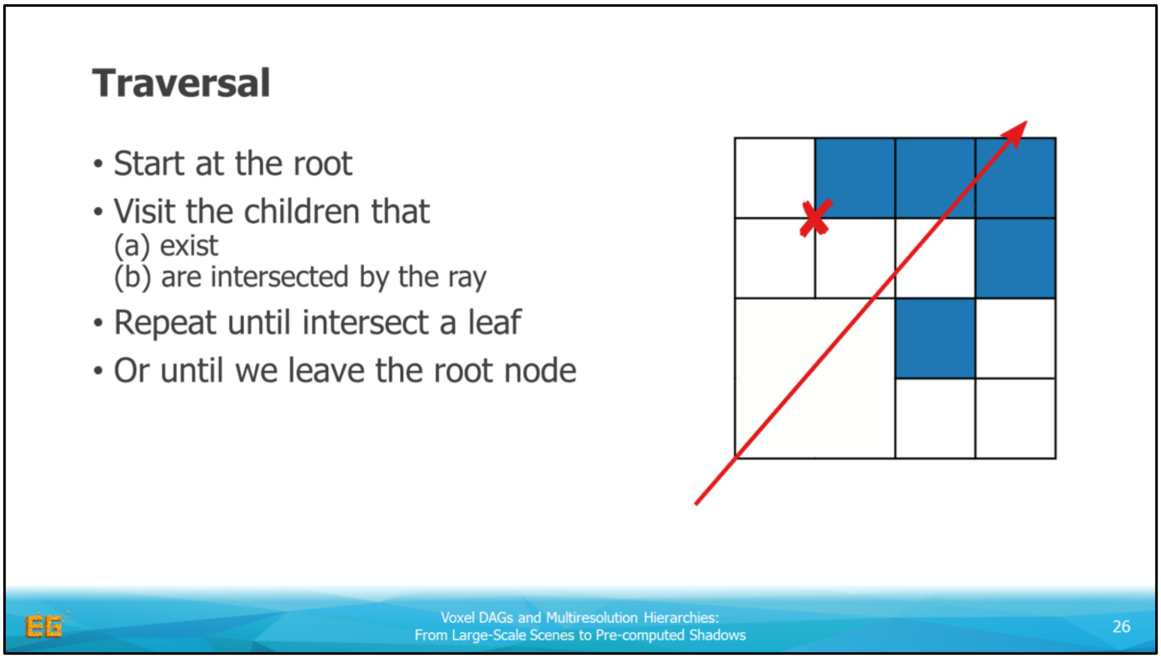
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

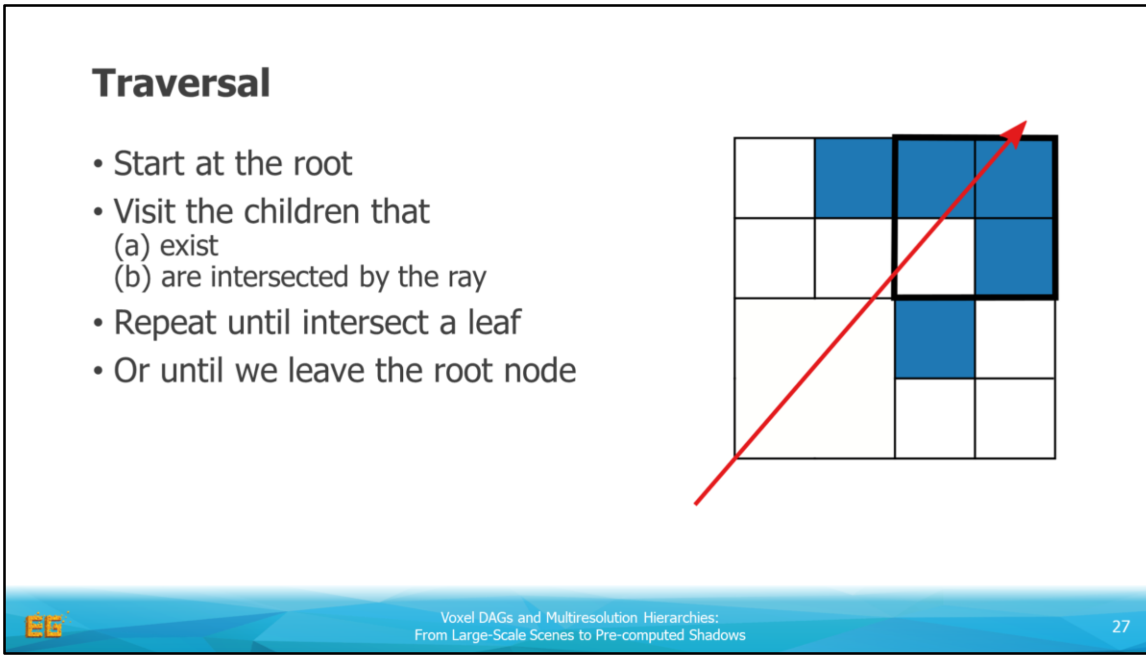
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

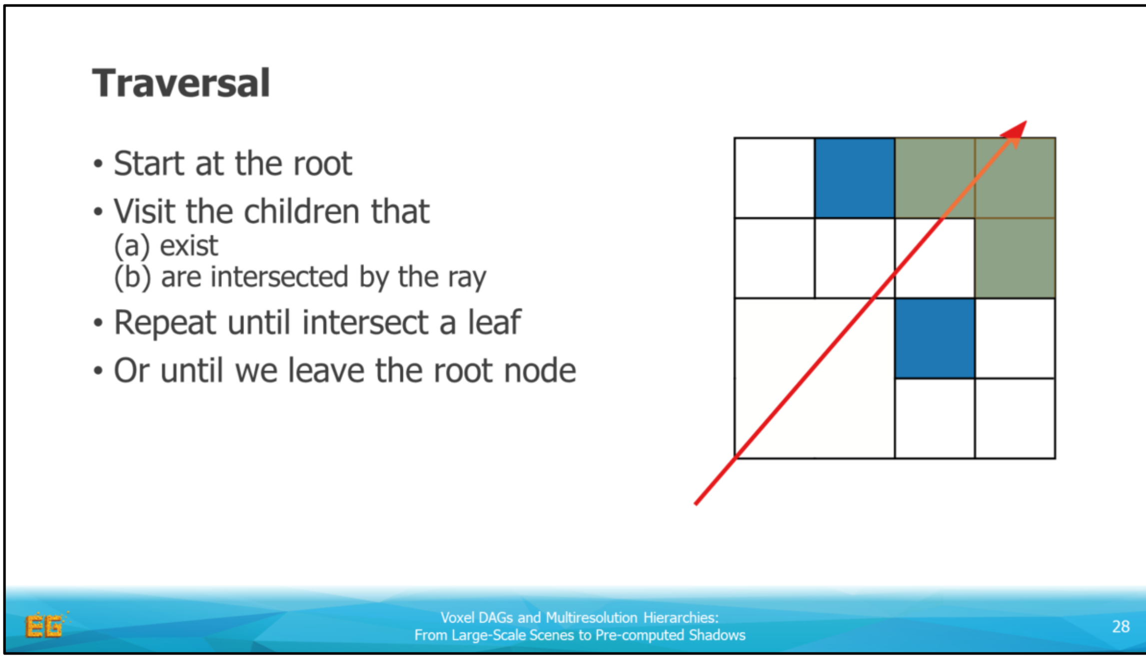
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

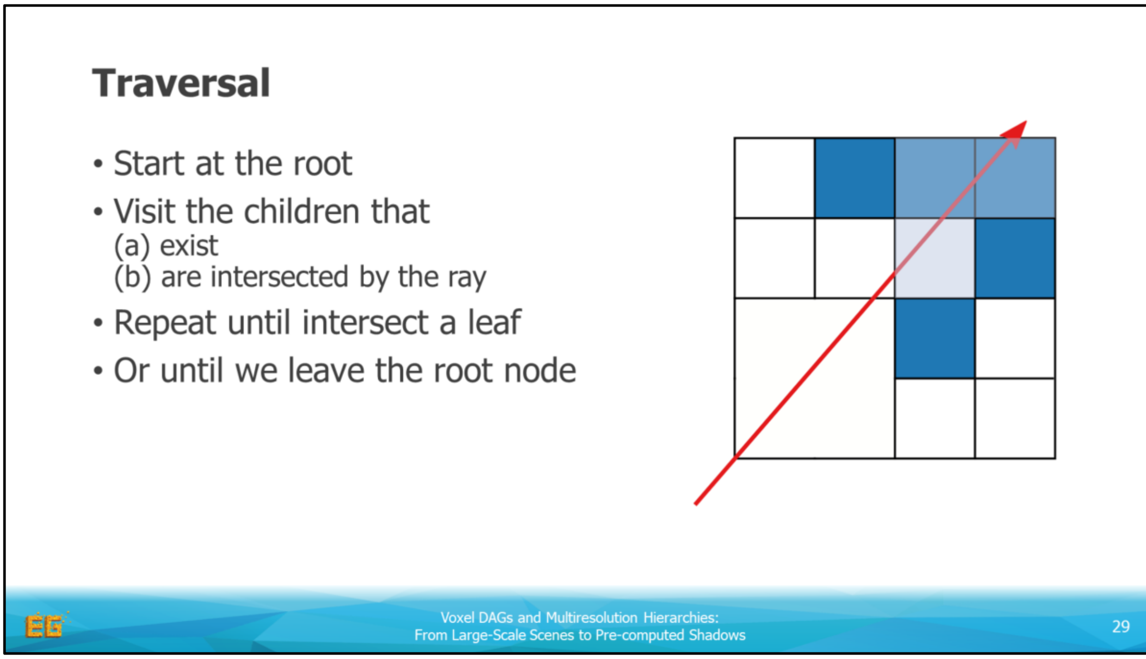
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

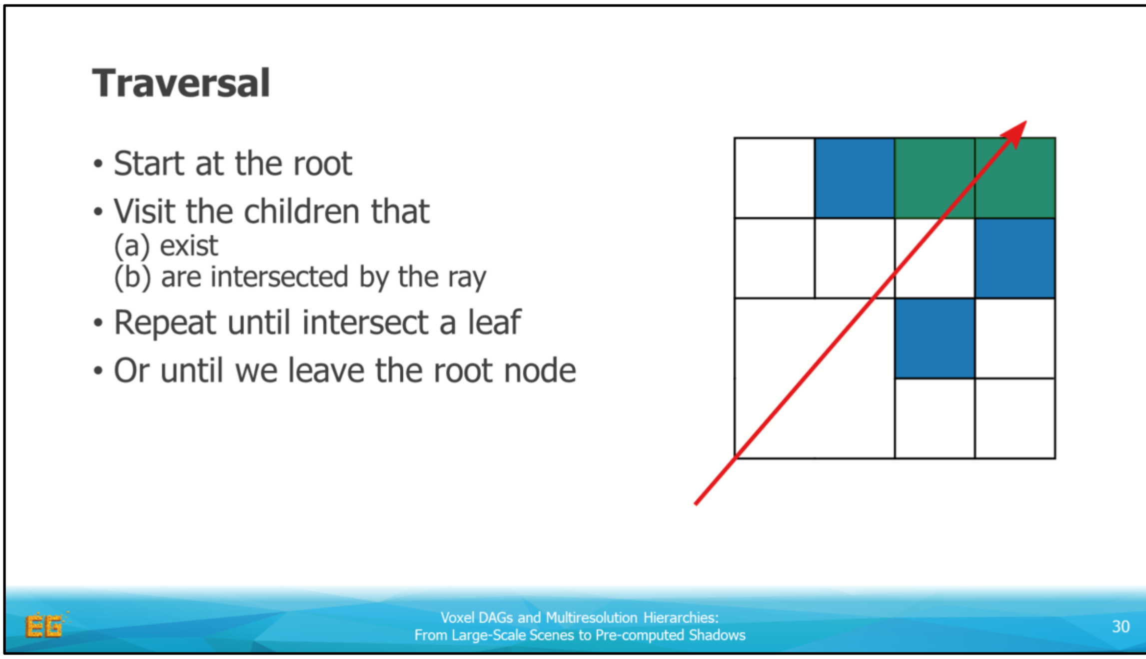
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

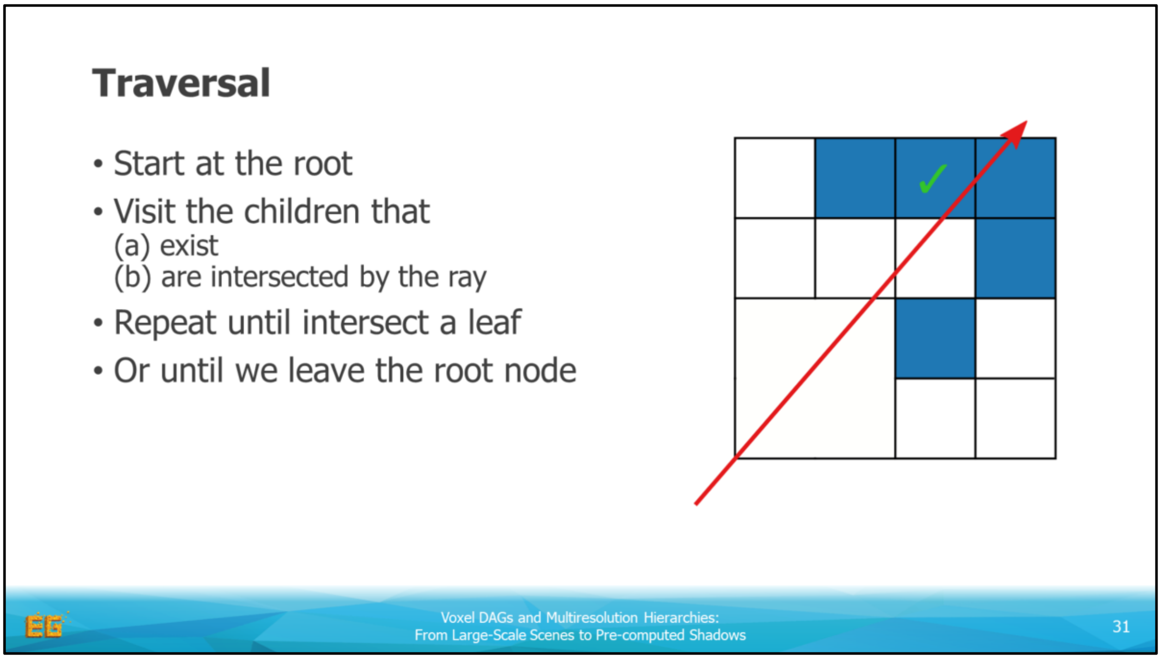
This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

This is repeated until we either ..



Traversing a voxel DAG is, in general terms, the same as any other tree traversal…

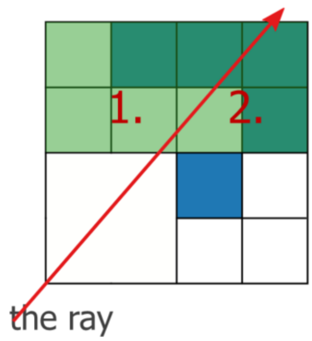
- *We start at the root node*
- *… and we want to visit the children that (a) exist …*
- *… and (b) intersect the ray*

This is repeated until we either ..
… hit a leaf node (or a node at the desired level), as illustrated here, or leave the root node, at which point we know that the ray did not intersect with the voxel geometry.

Traversal

- Start at the root
- Visit the children that
  (a) exist
  (b) are intersected by the ray
- Repeat until intersect a leaf
- Or until we leave the root node

- Depth first
- First intersection: visit children in order along the ray



As illustrated, the traversal is depth first (and thus fairly cheap in terms of memory requirements).

If we want to find the first intersection (as opposed to just any), we need to visit the children in the right order at each level; that is, we want to descend into the child closest to the rays origin first. In the illustration here, this is the left child. Only after determining that the is no intersection in this left child, will we descend into the right child.

Traversal, II

- At each node
  - Check each child, possibly **descend**
  - Or if no children left, **ascend**

During traversal, there are essentially two phases:
- Descending further down into the DAG
- Ascending to higher-level nodes when no intersections are found

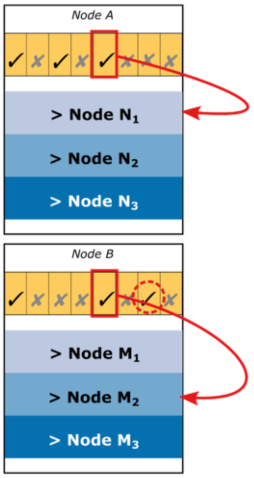
Traversal, III

- When **descending**
  - Figure out which child (number)
  - Fetch and follow corresponding pointer
  - Record this decision to build up the path

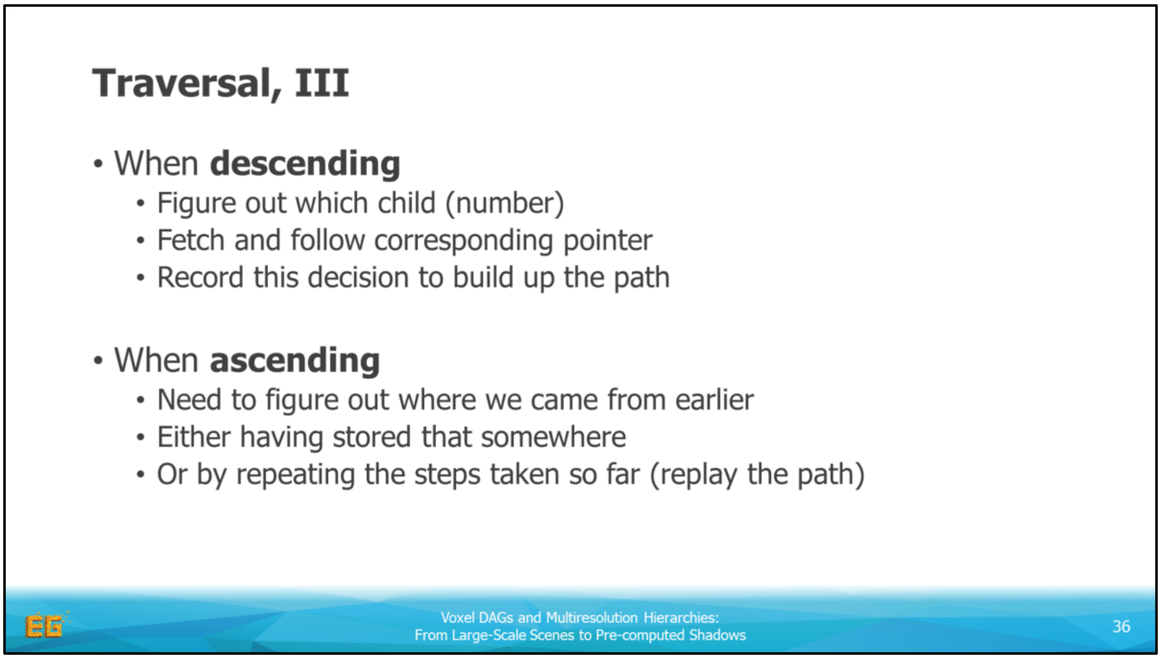
When descending, we need to figure out which child we should visit according to our ordering requirements. We then need to figure out the pointer corresponding to this child, fetch it and follow it. Finally, when doing this, we need to record the decision to build up the path.

Traversal, III

- When **descending**
  - Figure out which child (number)
  - Fetch and follow corresponding pointer
  - Record this decision to build up the path



The index of the pointer varies a bit, as it depends on the number of present children "preceding" the selected one. An efficient implementation to solve this will be shown later.

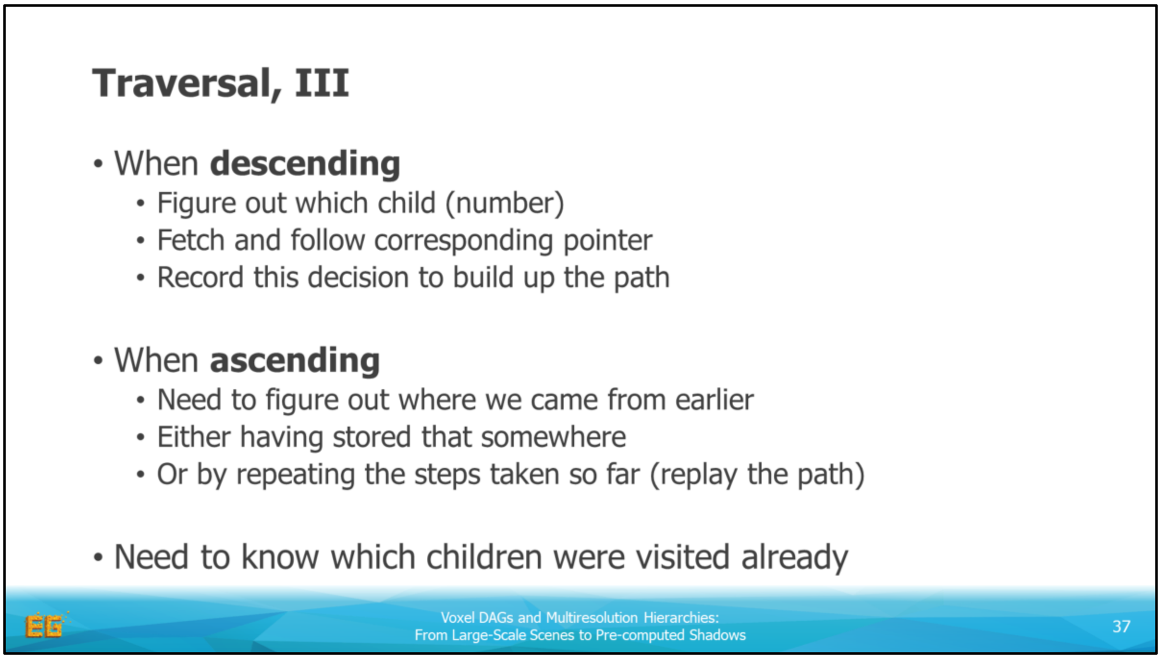


When we arrive at a node with no intersecting children, or where we've visited all intersecting children, we need to ascend to the parent node, that is, the node from which we earlier descended into the current node. To do this, well, we need to figure out where we came from.

There are essentially two options:

- Explicitly store where we have come from (using e.g., a stack)
- Or by repeating the steps that we've taken to get us here.

The latter information is available via the path that we've been recording, so we just need to replay it up to the second-to-last decision.



When we arrive at a node with no intersecting children, or where we've visited all intersecting children, we need to ascend to the parent node, that is, the node from which we earlier descended into the current node. To do this, well, we need to figure out where we came from.

There are essentially two options:
- *Explicitly store where we have come from (using e.g., a stack)*
- *Or by repeating the steps that we've taken to get us here.*

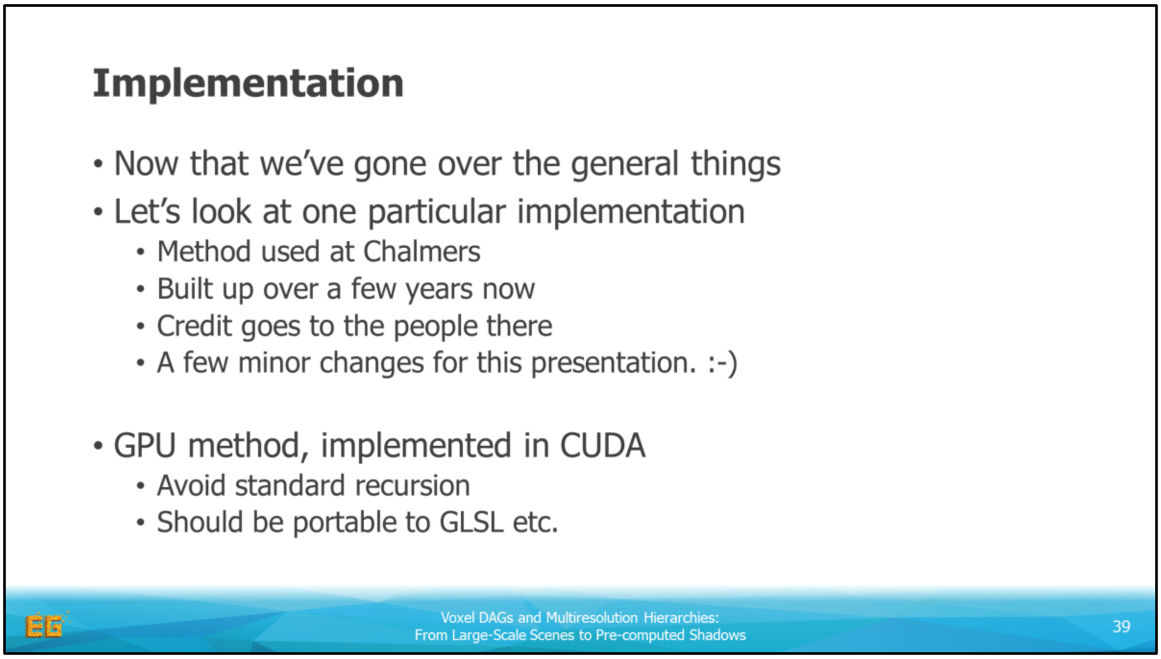
The latter information is available via the path that we've been recording, so we just need to replay it up to the second-to-last decision.

Additionally, we need to keep track of which children we've already visited at each node, as to avoid revisiting certain subtrees. There are a few options available. One option is to store this explicitly somewhere. A different one is to store a coordinate along the ray and update it during traversal as well, essentially indicating which nodes we've already "left behind".

Implementation

- Now that we've gone over the general things
- Let's look at one particular implementation
  - Method used at Chalmers
  - Built up over a few years now
  - Credit goes to the people there
  - A few minor changes for this presentation. :-)

I will now move on to describe one particular implementation in somewhat more detail. This particular method has been used Chalmers for a couple of years now.



It's a GPU-based method, implemented in CUDA. However, there are no CUDA specific operations, so the general idea should be implementable in most modern shader environments.

Additional information:
As mentioned by Ulf Assarsson after the talk, we implemented a DAG ray caster in GLSL targeting WebGL 2.0. The DAG structure can be encoded into textures (rather than simple buffers/arrays) should this be necessary. The tricky part is storing the stack, should local memory not be an option. In such a case, replaying the path + advancing along the ray might be a more appropriate option.

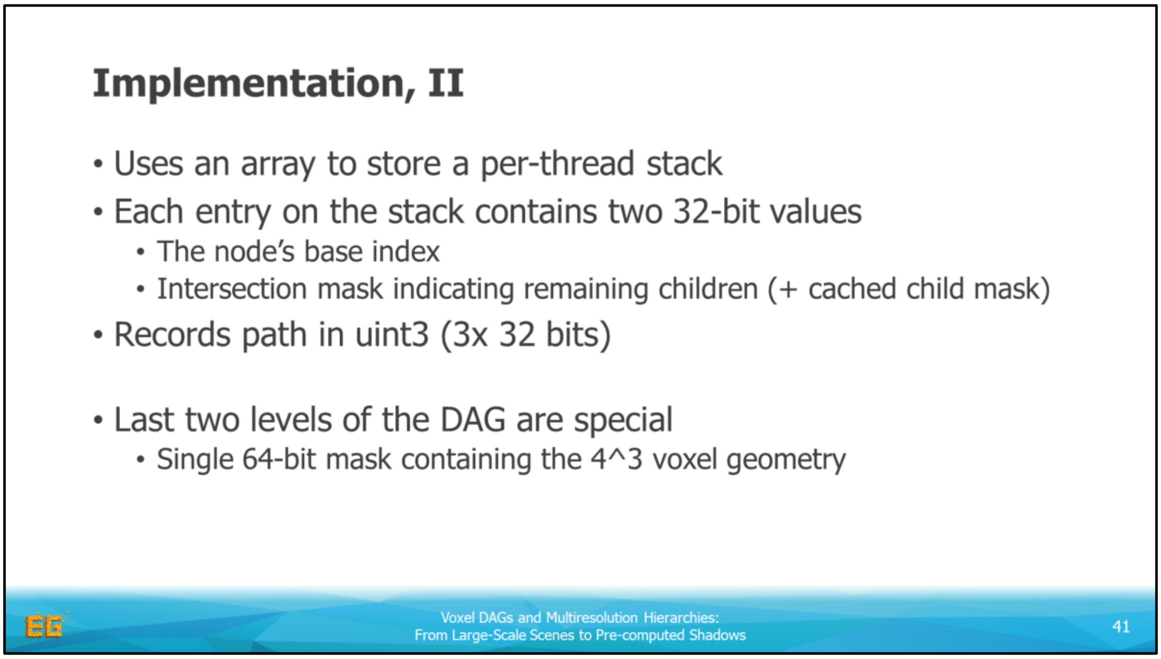
Implementation, II

- Uses an array to store a per-thread stack
- Each entry on the stack contains two 32-bit values
  - The node's base index
  - Intersection mask indicating remaining children (+ cached child mask)
- Records path in uint3 (3x 32 bits)

This implementation uses a small per-thread array to implement a per-thread stack.

Each entry on the stack contains two 32-bit values: the node's base index (i.e., the index of the word containing the child mask), and an intersection mask, which indicates the children that are present, **and** intersected by the ray **and** not yet visited. This intersection mask is the explicitly stored state mentioned earlier. We also store a copy of the original child mask; this avoids having to re-fetch it from the DAGs memory.

Paths are recoded pretty much as described earlier, specifically in three 32-bit unsigned integers.

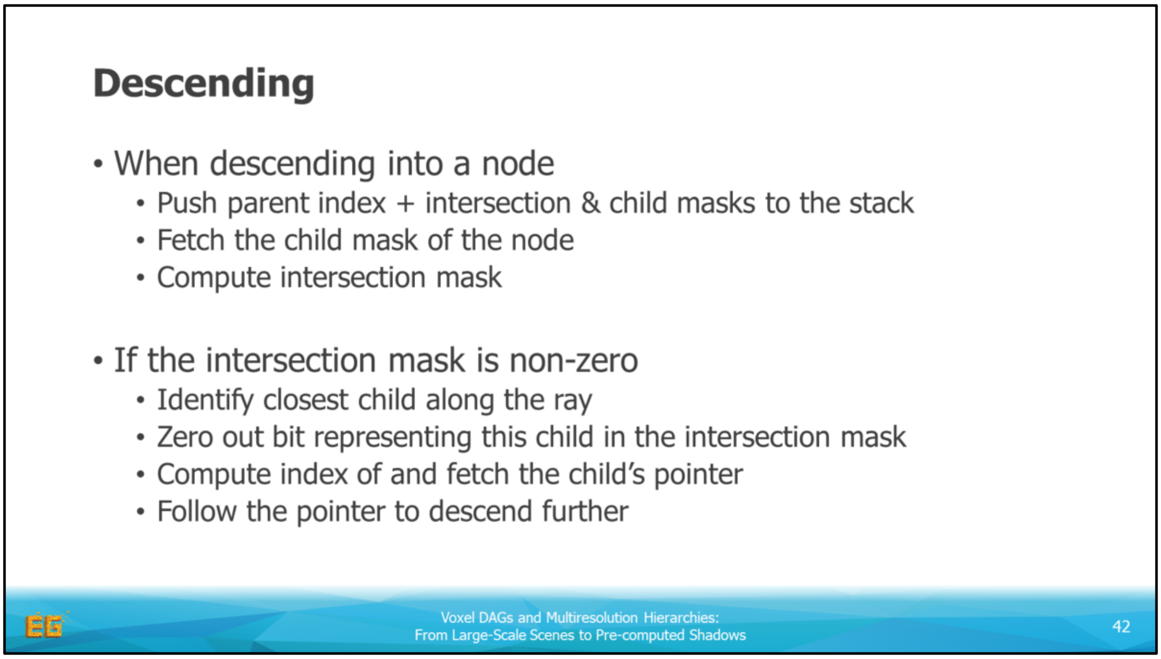


This implementation uses a small per-thread array to implement a per-thread stack.

Each entry on the stack contains two 32-bit values: the node's base index (i.e., the index of the word containing the child mask), and an intersection mask, which indicates the children that are present, ***and*** *intersected by the ray* ***and*** *not yet visited. This intersection mask is the explicitly stored state mentioned earlier. We also store a copy of the original child mask; this avoids having to re-fetch it from the DAGs memory.*

Paths are recoded pretty much as described earlier, specifically in three 32-bit unsigned integers.

As indicated in earlier talks, the last two levels of the DAG are stored differently. Instead of storing the DAG structure, the full 4x4x4 voxel sub-volume is packed into a single 64-bit mask (using two 32-bit words).

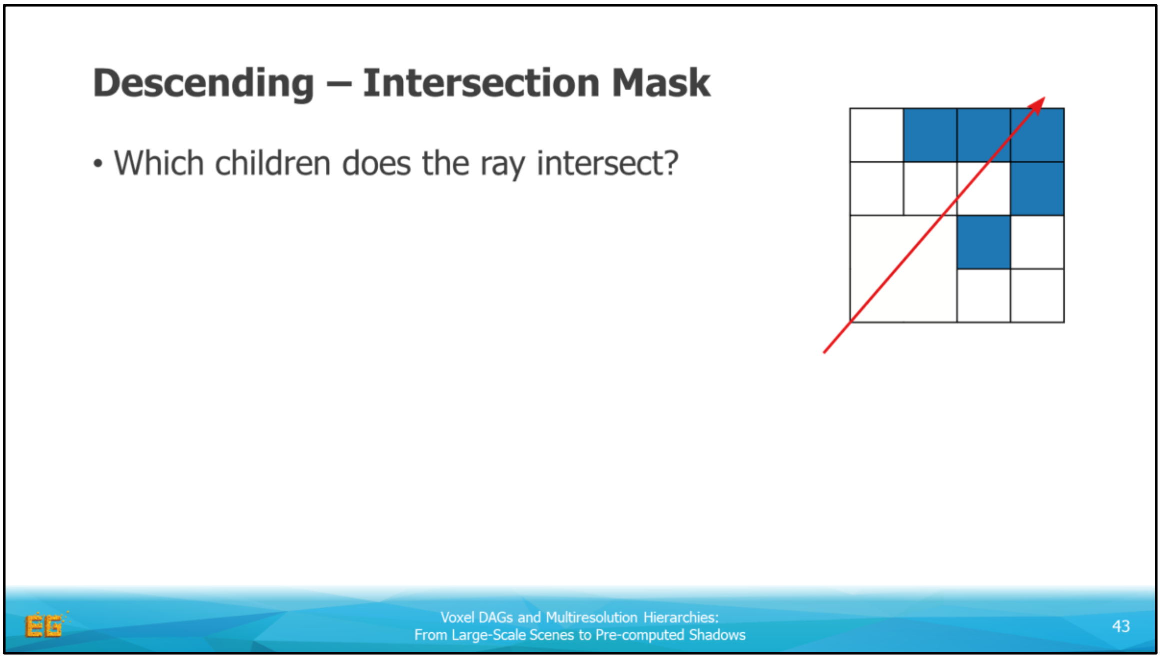


First, let's look at the descending-phase of the traversal in a bit more detail.

When we descend into a node, we need to
- Push the parent index and intersection mask (+ child mask) onto the stack, in case we should later return to it
- Then we need to fetch the child mask of the new node
- Based on this, we want to compute the initial intersection mask, which tells us which children we need to visit

If the intersection mask is zero, we should ascend to the parent node (described later). If it is non-zero, we need to identify the child closest along the ray. We then want to update the intersection mask to indicate that we've already visited that child. Then, we need to find correct pointer and so that we can fetch and follow it to descend further into the DAG.

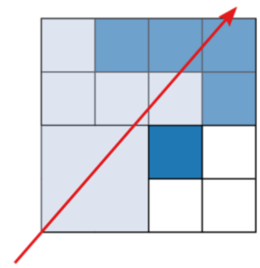
I will briefly detail three operations in the following slides:
1. Computing the intersection mask
2. Identifying the closest child
3. Computing the index of the pointer



The intersection mask tells us which children we need to visit. Initially it is constructed by combining

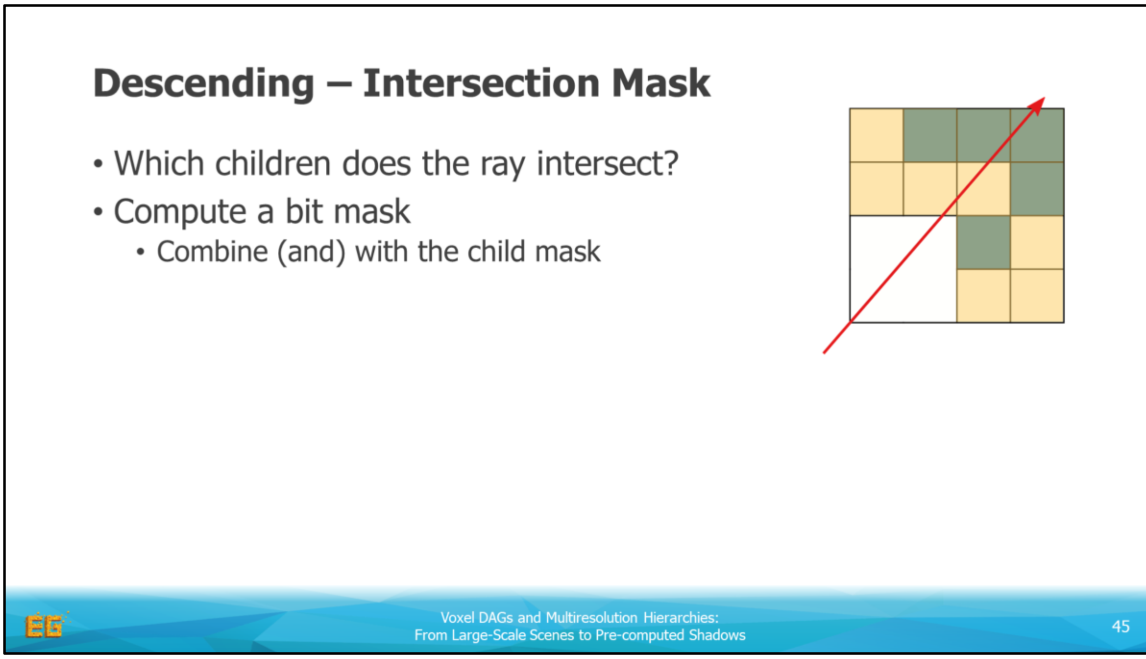
Descending – Intersection Mask

- Which children does the ray intersect?
- Compute a bit mask
  - Combine (and) with the child mask



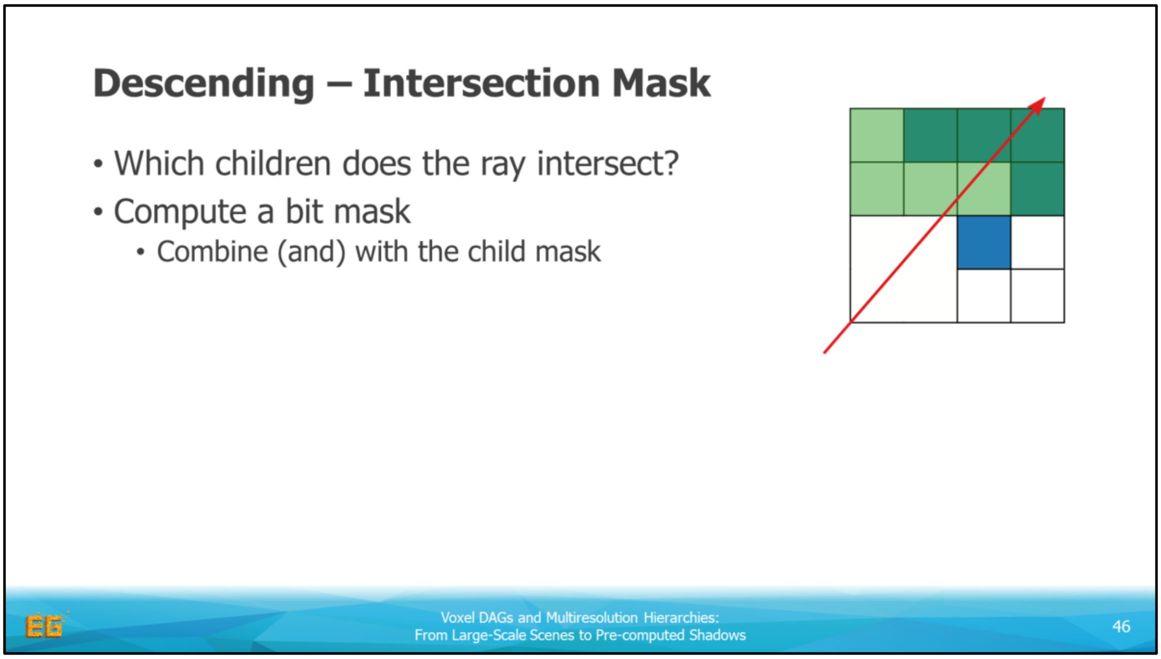
The intersection mask tells us which children we need to visit. Initially it is constructed by combining

- The bit mask indicating which children then the ray intersects



The intersection mask tells us which children we need to visit. Initially it is constructed by combining

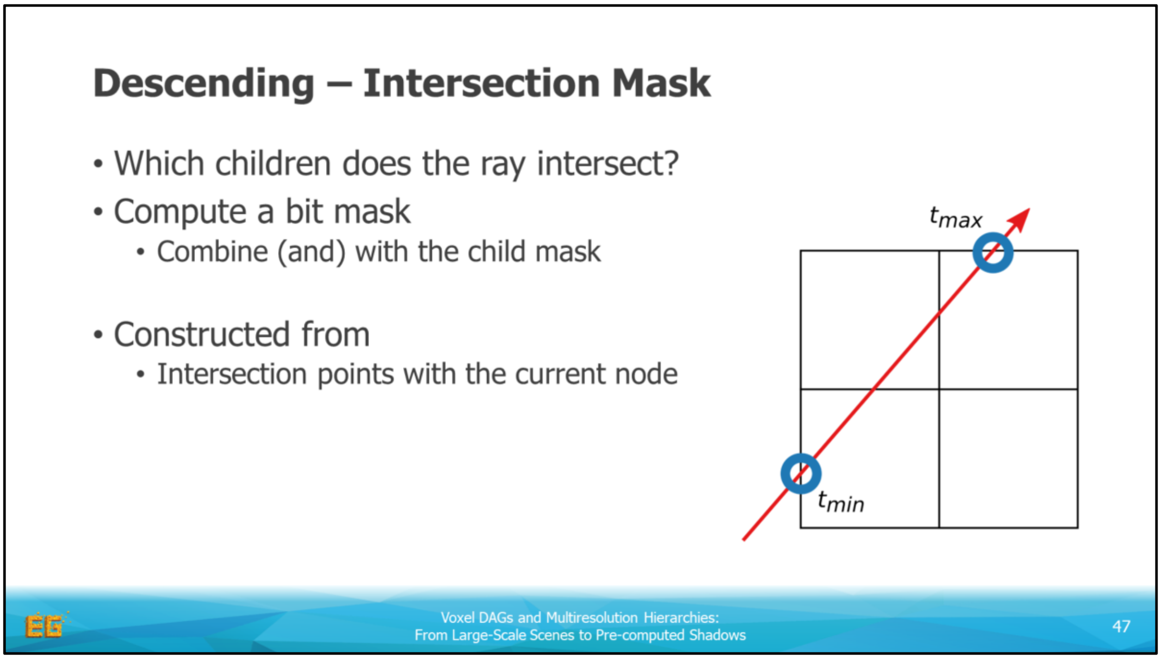
- *The bit mask indicating which children then the ray intersects*
- The bit mask indicating the present children (i.e., the child mask)



The intersection mask tells us which children we need to visit. Initially it is constructed by combining

- *The bit mask indicating which children then the ray intersects*
- *The bit mask indicating the present children (i.e., the child mask)*

using bit-wise and.

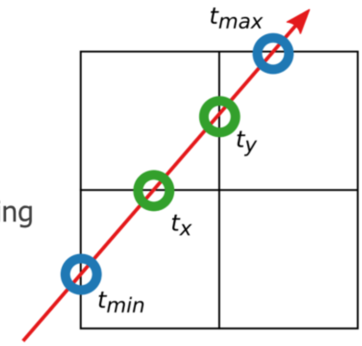


The first bit-mask may be computed from a number of points, as follows.

First, we need the intersection points between the ray and the bounding box of the current node, as shown here.

Descending – Intersection Mask

- Which children does the ray intersect?
- Compute a bit mask
  - Combine (and) with the child mask
- Constructed from
  - Intersection points with the current node
  - Intersections with axis aligned planes bisecting the current node. (t_z not shown)

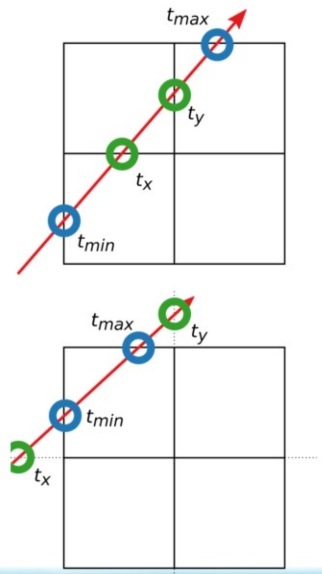


The first bit-mask may be computed from a number of points, as follows.

First, we need the intersection points between the ray and the bounding box of the current node, as shown here.
Second, we need the intersection points between the ray and the axis aligned planes bisecting the current node; that is, intersections with the planes that go through the middle of the node. This figure shows two of them, the final t_z is not shown here.

Descending – Intersection Mask, II

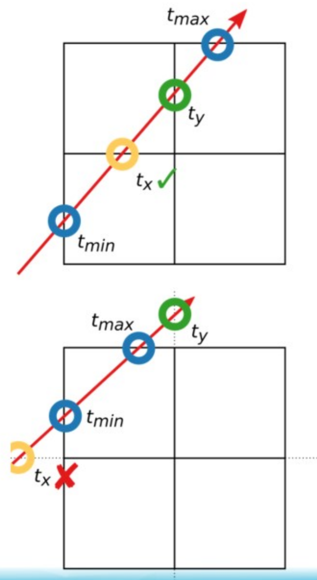
- If t_x is in the range $[t_{min}, t_{max}]$



We then perform checks for each axis. In this example, I will begin with the X-axis.

Descending – Intersection Mask, II

- If t_x is in the range $[t_{min}, t_{max}]$

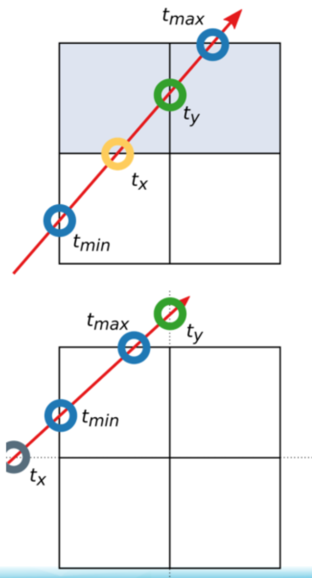


We then perform checks for each axis. In this example, I will begin with the X-axis.

If the intersection point (t_x) is in the range of t_min and t_max,

Descending – Intersection Mask, II

- If t_x is in the range $[t_{min}, t_{max}]$
  - Add children to the intersection mask which are located in the same half as t_y and in the same half as t_z.



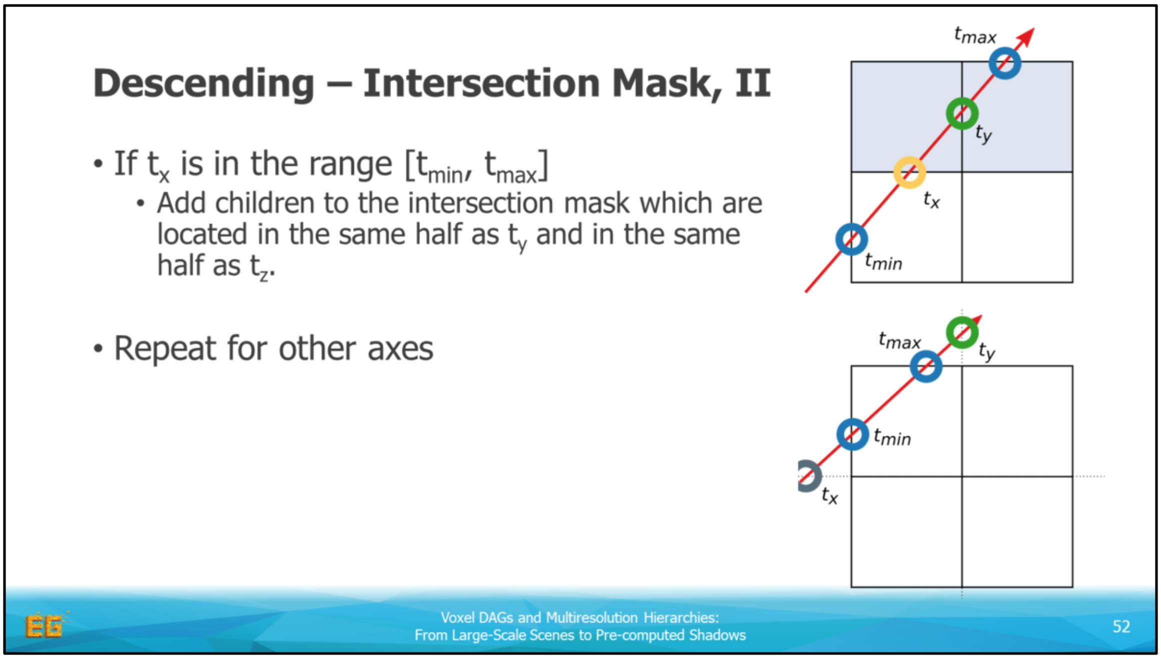
Voxel DAGs and Multiresolution Hierarchies:
From Large-Scale Scenes to Pre-computed Shadows

51

We then perform checks for each axis. In this example, I will begin with the X-axis.

If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection **and** in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.

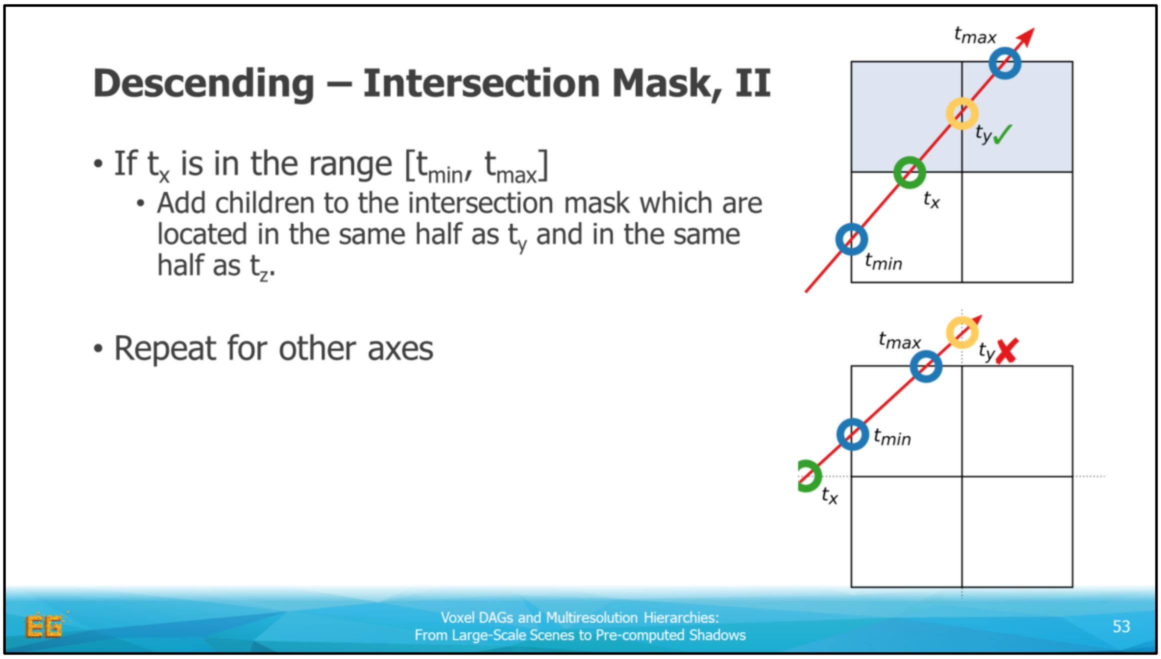
Additional notes:
We include a small epsilon to the check for which half of the node the intersection lies in – this avoids some numerical problems. Essentially, the nodes in the top figure are added if t_y is larger than (MidPlane_y – epsilon); the nodes in the bottom would have been added if t_y was smaller than (MidPlane_y + epsilon). This can be implemented efficiently using bitwise operations (ORing the cases for each axis and ANDing the results). More than a quarter of the node can be added for each axis, though.



We then perform checks for each axis. In this example, I will begin with the X-axis.

If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection ***and*** *in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

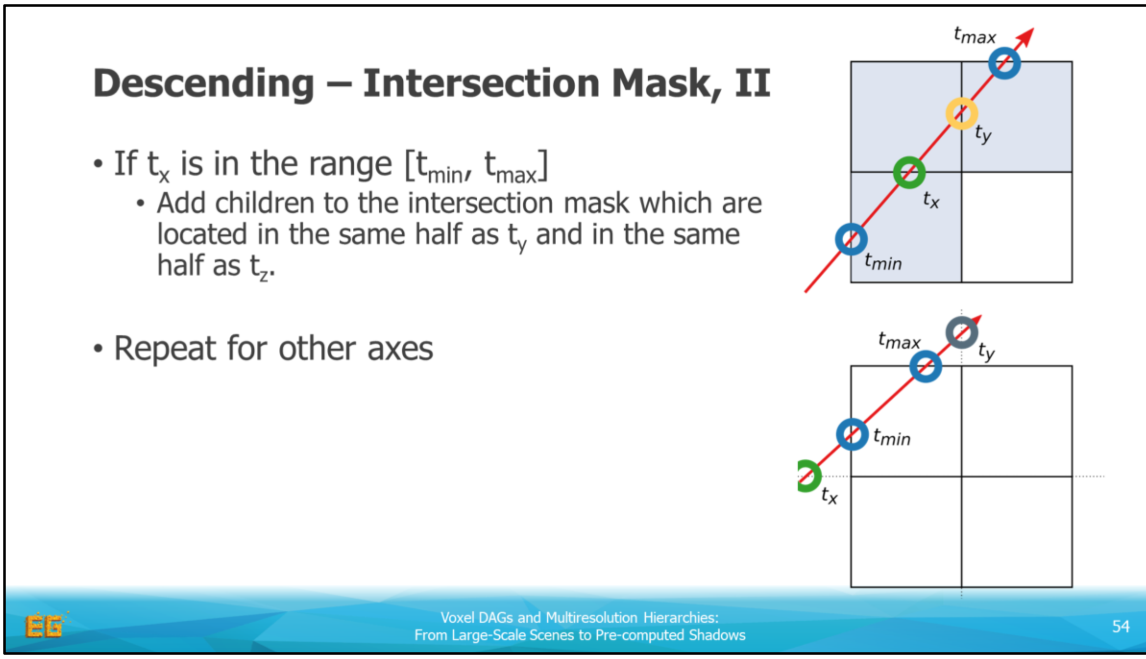
The same operation is repeated for the other axes.



We then perform checks for each axis. In this example, I will begin with the X-axis.

If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection ***and*** *in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

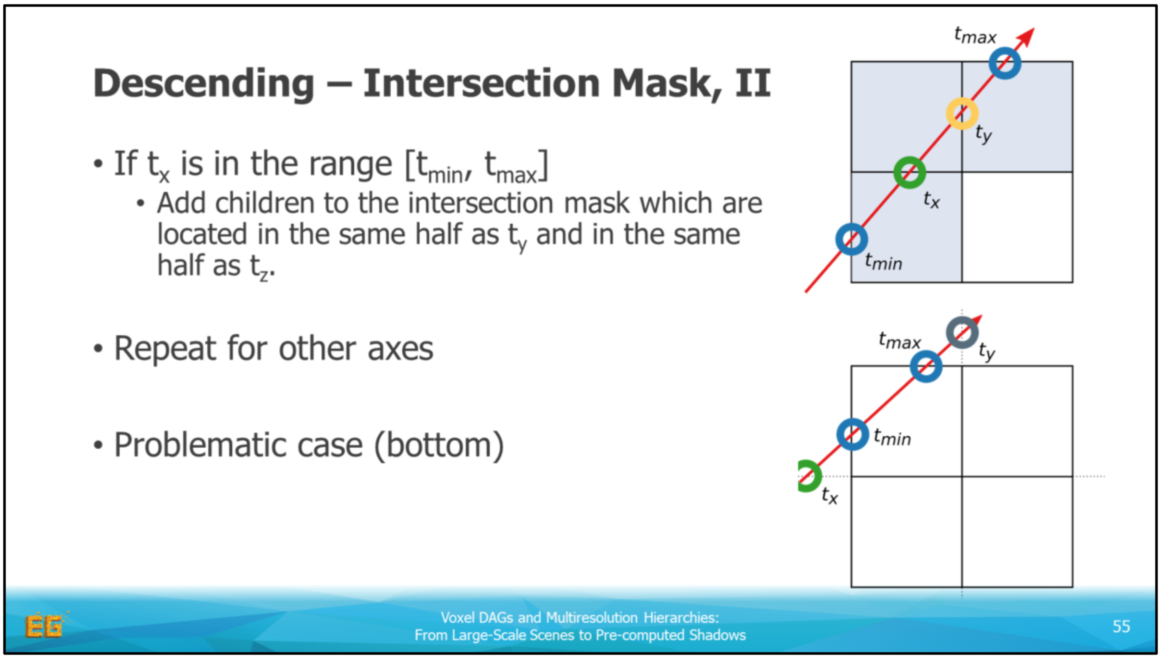
The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well.



We then perform checks for each axis. In this example, I will begin with the X-axis.

If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection ***and*** *in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well, which adds the node as indicated. I will omit the Z-axis here, but one would have to repeat the same operations for it too.

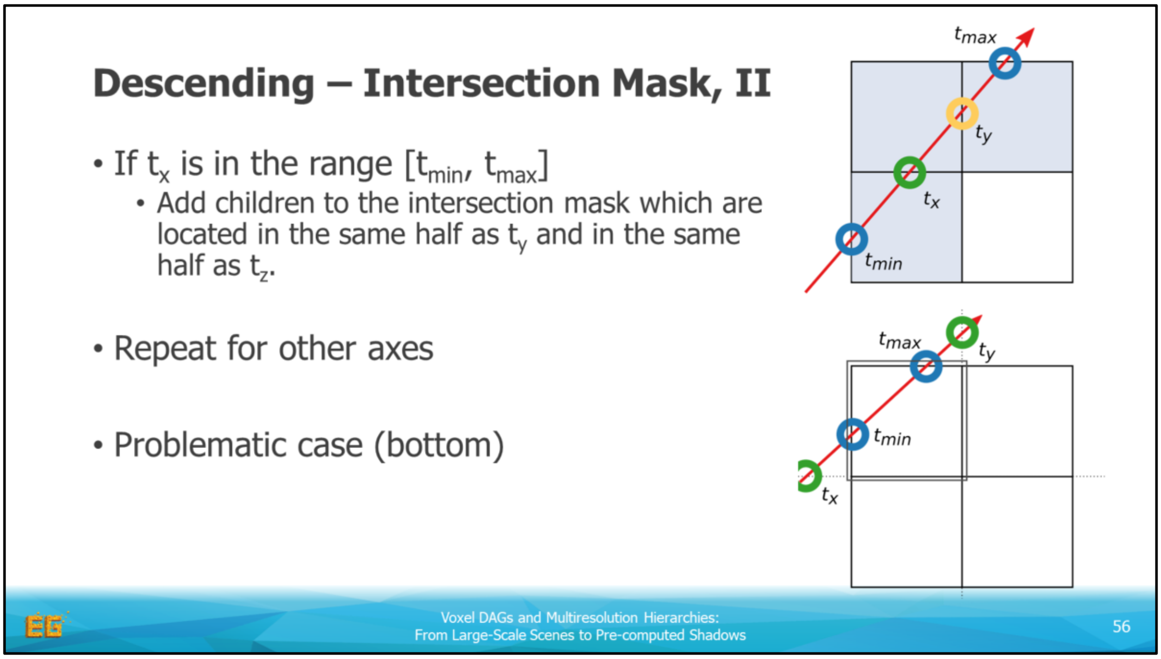


We then perform checks for each axis. In this example, I will begin with the X-axis.

*If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection **and** in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well, which adds the node as indicated. I will omit the Z-axis here, but one would have to repeat the same operations for it too.

As we see, there is a problematic case that has been illustrated in the bottom figure: the previous steps added no children to the mask, despite the ray intersecting with one of them.

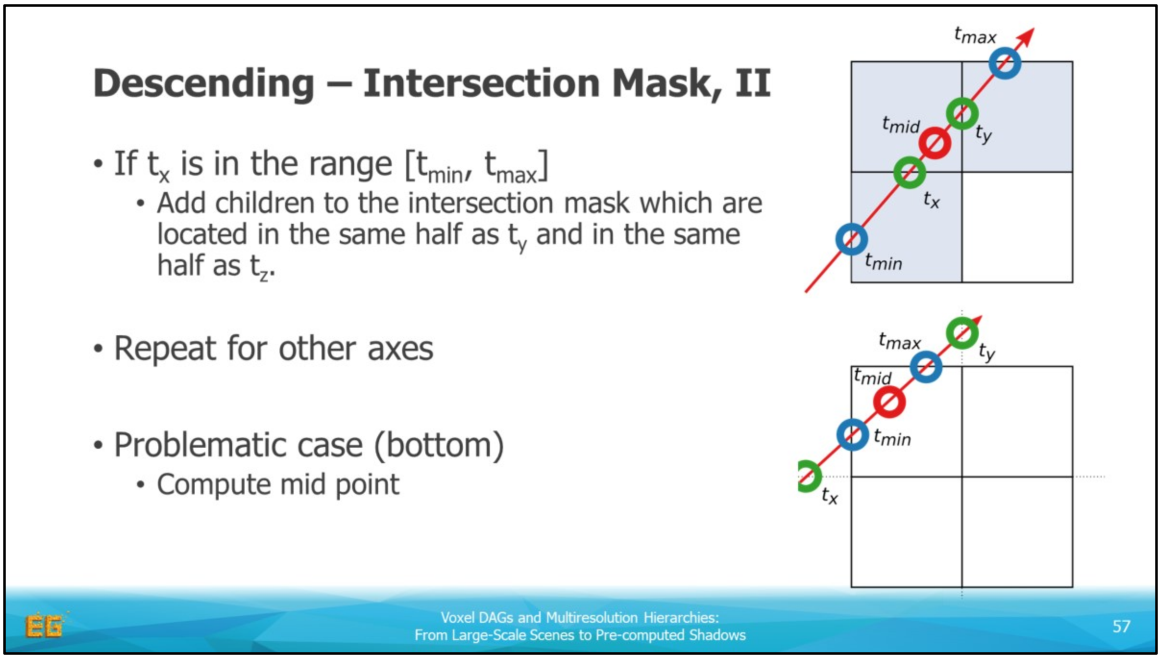


We then perform checks for each axis. In this example, I will begin with the X-axis.

*If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection **and** in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well, which adds the node as indicated. I will omit the Z-axis here, but one would have to repeat the same operations for it too.

As we see, there is a problematic case that has been illustrated in the bottom figure: the previous steps added no children to the mask, despite the ray intersecting with one of them.



We then perform checks for each axis. In this example, I will begin with the X-axis.

*If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection **and** in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

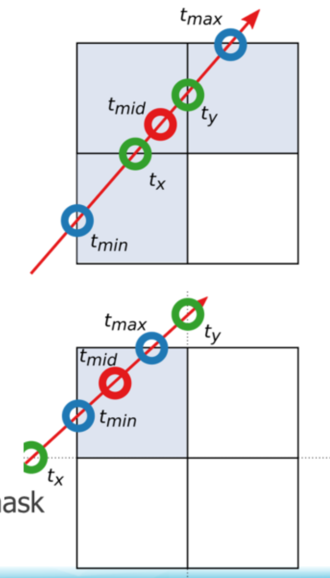
The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well, which adds the node as indicated. I will omit the Z-axis here, but one would have to repeat the same operations for it too.

As we see, there is a problematic case that has been illustrated in the bottom figure: the previous steps added no children to the mask, despite the ray intersecting with one of them.

This is fixed by computing the mid point between t_min and t_max.

Descending – Intersection Mask, II

- If t_x is in the range $[t_{min}, t_{max}]$
  - Add children to the intersection mask which are located in the same half as t_y and in the same half as t_z.
- Repeat for other axes
- Problematic case (bottom)
  - Compute mid point
  - Add voxel in octant of midpoint to intersection mask



We then perform checks for each axis. In this example, I will begin with the X-axis.

*If the intersection point (t_x) is in the range of t_min and t_max, we will add the children that are located in the same half of the node as the t_y intersection **and** in the same half as the t_z intersection, giving us the children in one quarter of the node. These will be added to the intersection mask.*

The same operation is repeated for the other axes. I will briefly illustrate the Y-axis as well, which adds the node as indicated. I will omit the Z-axis here, but one would have to repeat the same operations for it too.

As we see, there is a problematic case that has been illustrated in the bottom figure: the previous steps added no children to the mask, despite the ray intersecting with one of them.

This is fixed by computing the mid point between t_min and t_max, and simply adding the child in the octant where this midpoint resides. In the top figure, no new children were added – this doesn't generate any false positives.

The computed intersection mask is combined with the child mask to give final intersection mask indicating which children should be visited.

Descending – Child Order

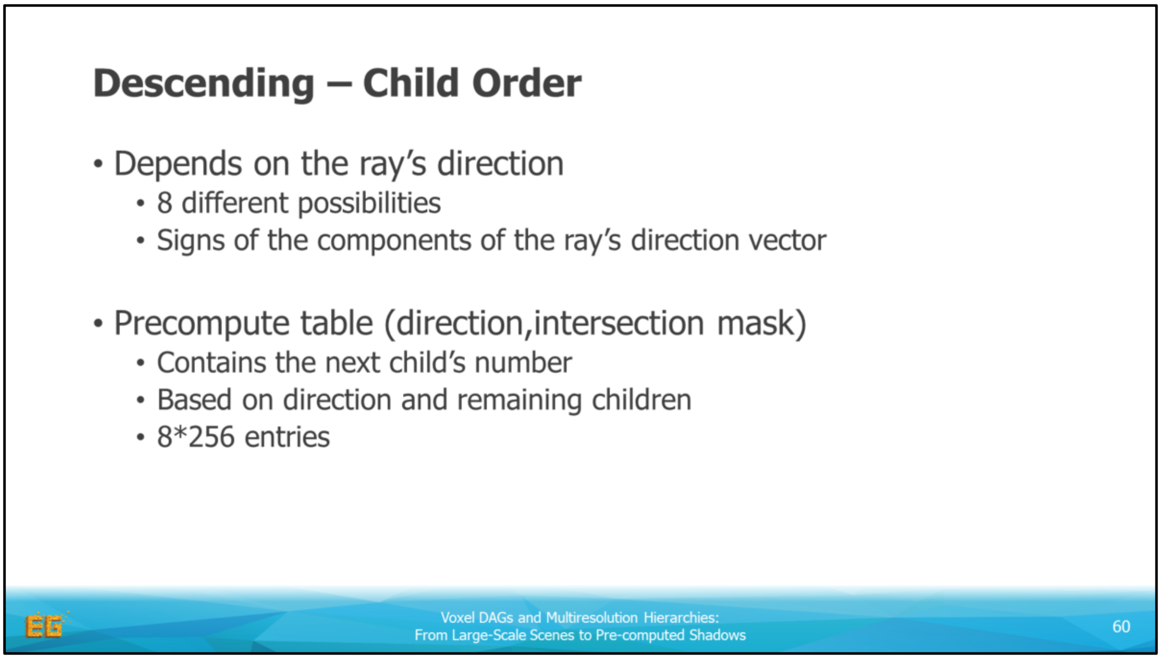
- Depends on the ray's direction
  - 8 different possibilities
  - Signs of the components of the ray's direction vector

Voxel DAGs and Multiresolution Hierarchies:
From Large-Scale Scenes to Pre-computed Shadows

59

The second item I wanted to highlight is the order in which children are visited.

There are eight different possibilities, and these are determined by the signs of the components of the ray's direction vector.



The second item I wanted to highlight is the order in which children are visited.

There are eight different possibilities, and these are determined by the signs of the components of the ray's direction vector.

So far, the most efficient way seems to be the use of a pre-computed table. The table gives the index of the next child that we should descend into, given the discretized ray direction (8 values) and the current intersection mask (256 values). The final table contains 8*256 (=2 k) entries; each entry can be packed into a single byte.

Additional Information:
The direction can be stored as a bit mask with 3 bits. Bit 3 is set to one if the x-component of the ray's direction is negative. Bit 2 is set to one if the y-component is negative, and bit 1 is set to one if the z-component is negative. Call this the directionMask.

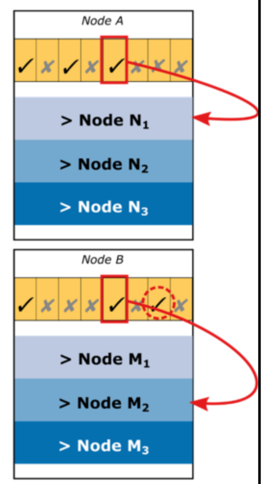
The lookup table may then be computed as follows:
- For each direction d in [0, 8)
- For each intersection mask m in [0,256)

- for(i = 0; i < 8; ++i) { j = directionMask ^ i; if bit j in is set in m, set lut[d*256+mask] = j; }

(technically, m = 0 has no bits set, but we should never ask for the next child of an intersection mask with no bits set.)

Descending – Fetch pointer

- Figure out which pointer belongs to a certain child
- Right side: fourth child (from right)
  - Depends on the number of present children preceding it

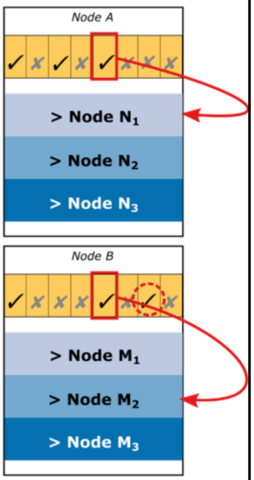


Once we know the (bit index) of the child we should visit next, we need to figure out which pointer corresponds to that child. This depends on the number of children present before the selected child (in the figure, this is the number of children present to the right of the selected one – zero in the upper figure and one in the lower one).

Descending – Fetch pointer

- Figure out which pointer belongs to a certain child
- Right side: fourth child (from right)
  - Depends on the number of present children preceding it

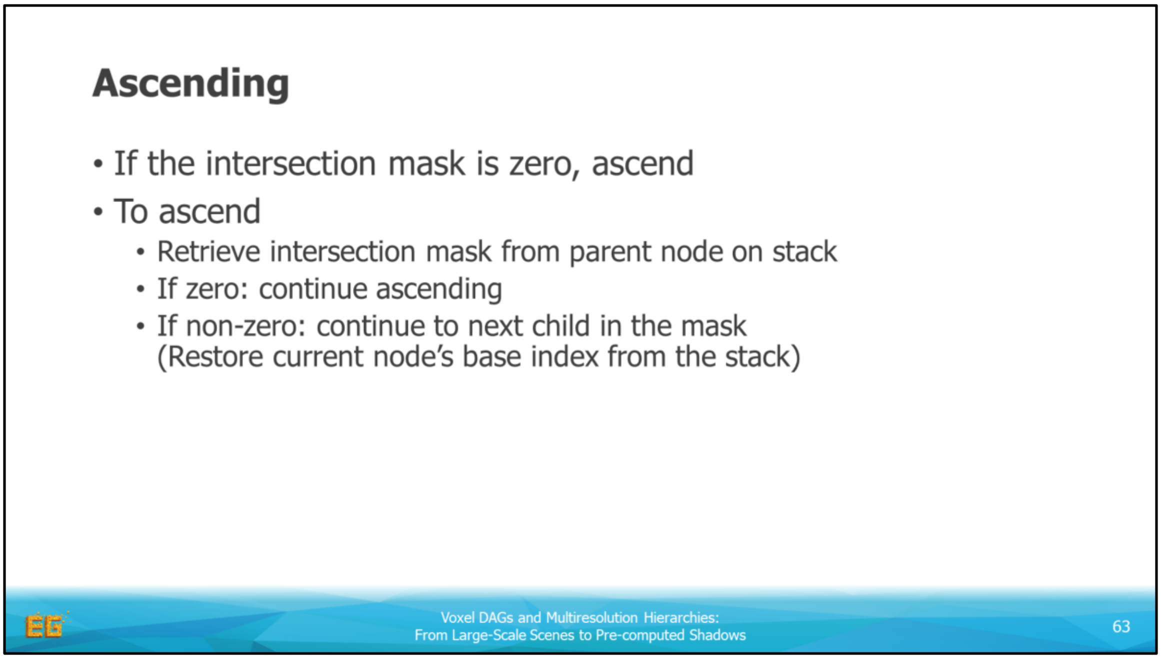
- Mask with lower bits: `lower = (1 << childNumber) - 1`
- And with child mask: `preceding = (childMask & lower)`
- Count set bits: `pointerNumber = __popc(preceding)`



Once we know the (bit index) of the child we should visit next, we need to figure out which pointer corresponds to that child. This depends on the number of children present before the selected child (in the figure, this is the number of children present to the right of the selected one – zero in the upper figure and one in the lower one).

The pointer's index can be computed in three steps:

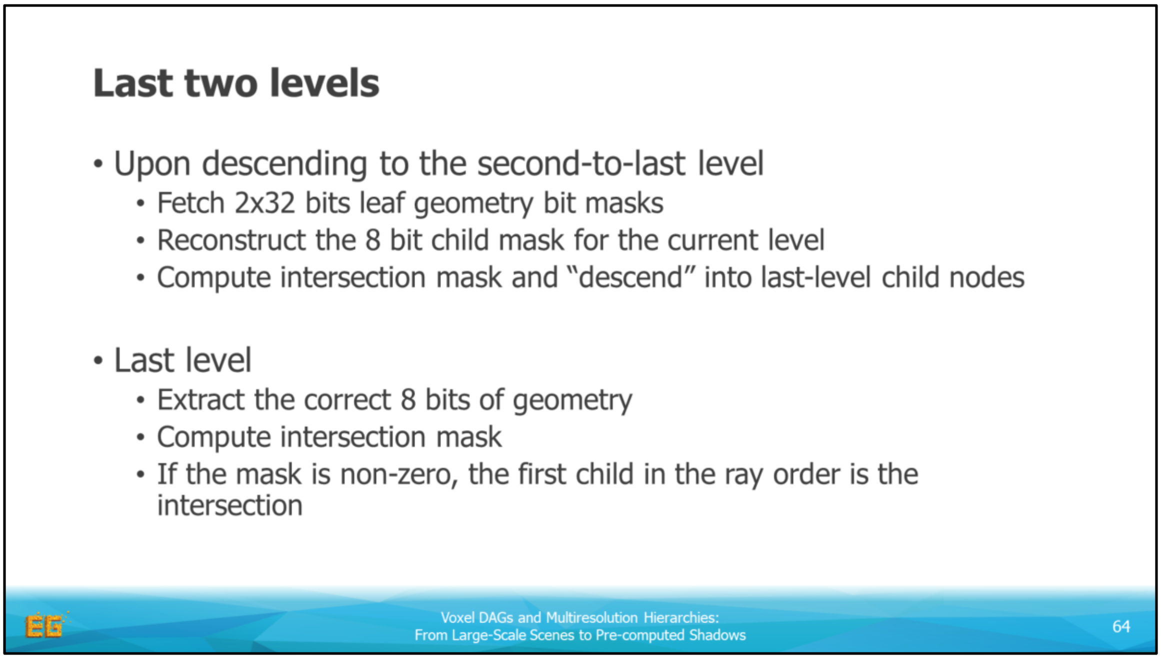
- First, we build a mask with the bits below the selected child set
- Second, we AND it with the child mask – this is where the cached child mask is required!
- Finally, we count the number of set bits using e.g., the __popc() intrinsic in CUDA. Similar intrinsics are available elsewhere (alternatively BitTwiddlingHacks has a reasonable implementation).



Whenever the child mask becomes zero, we need to ascend.

This is fairly simple: all the information we need is stored on the stack. First, we retrieve the intersection mask for the parent node from the stack. If it is zero, we will continue ascending. If it is non-zero, we want to descend into the next child, as indicated by the mask. We do this by passing the retrieved intersection mask to the lookup-table, get the child number, mask out the corresponding bit in the intersection mask … and so on, as described earlier.

If we ascend “out of the root node”, we’ve finished – there were no intersections with the voxel geometry.



If, during descending, we reach the second-to-last level, we need to something slightly different.

First, we fetch the two 32-bit words that contain the leaf geometry for the 4x4x4 voxel sub-volume represented by the current node.

From these two 32-bit words, it is possible to reconstruct a 8 bit child mask for the current level. By doing this, we can just reuse the machinery described so far … i.e., we can just plug the reconstructed child mask into the function for computing the intersection mask and then "descend" into the last level nodes. At this point, we don't need to touch the stack, which avoids some reads and writes to potentially slow memory.

At the last level, we do the same again – we extract the eight bits corresponding to the geometry of the selected child, and compute the intersection mask for this. If the mask is non-zero, the first child in the order along the ray is the intersection that we're looking for. If the mask is zero, we need to visit the other children from the parent node, or, if none remain, enter the ascending phase.

Summary

- DAGs behave a lot like trees when ray casting
  - Can't have backwards links encoded into the structure
  - But otherwise don't need to special case merged sub-trees
  - Many existing methods should work
- Presented one option
  - Fairly well tested at this point
  - Code will be made available

Voxel DAGs and Multiresolution Hierarchies:
From Large-Scale Scenes to Pre-computed Shadows

65

This concludes the presentation on practical ray casting against the DAG structure.

To summarize: DAGs behave a lot like an ordinary tree during ray casting – we did not consider merged sub-trees specially. Thus, if you have a traversal code, it shouldn't be too hard to adapt to a DAG (which is also one of the DAGs strengths). I also presented one possible traversal method that is fairly well tested at this point, and that should be fairly efficient on modern GPUs.

Additional Information:
Code will be made available in the coming weeks (i.e. after Eurographics 2018 concludes). I will link to the code from my homepage at
https://newq.net/publications/more/eg2018-voxel-dag-tutorial
(It will likely be on GitHub – but these slides might or might not get updated with the finalized information.)