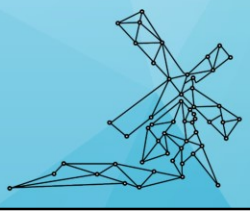
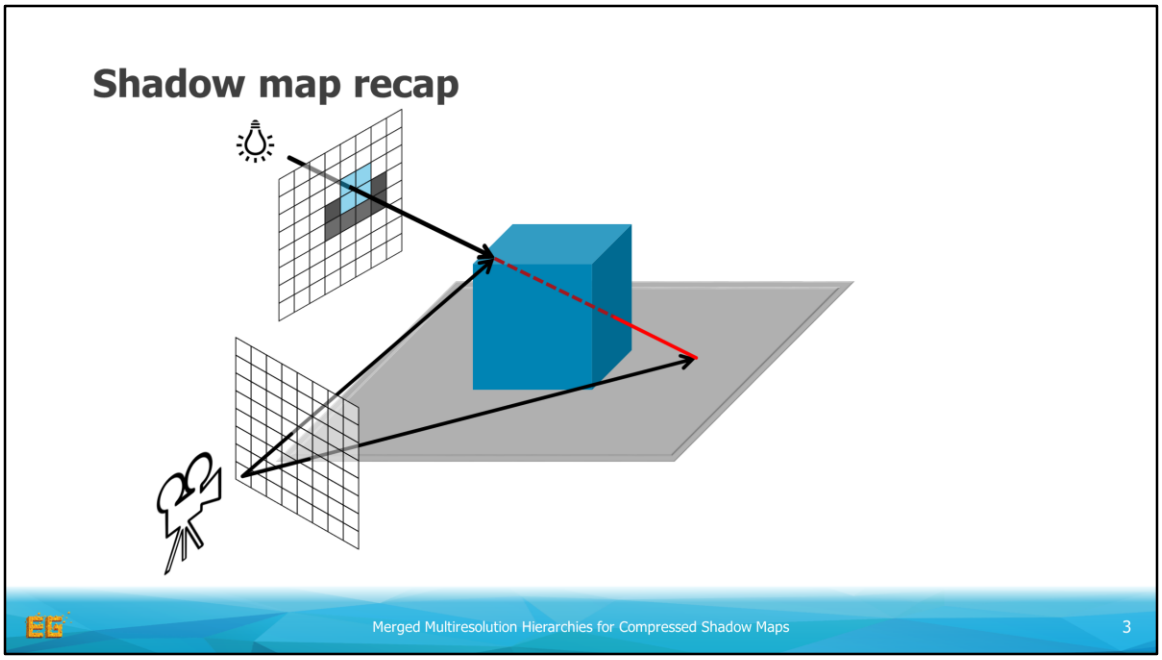
Merged Multiresolution Hierarchies for High Quality Compressed Shadow Maps

Leonardo Scandolo

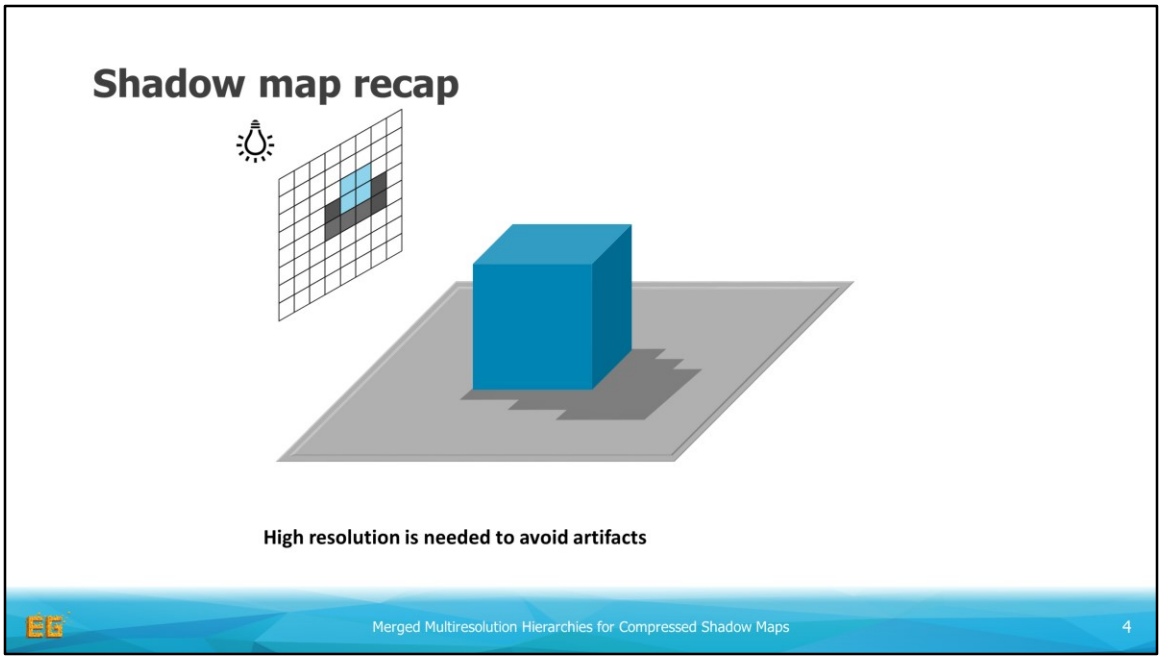
Delft University of Technology, The Netherlands



Background



Shadow maps create shadows by first rendering the scene from the viewpoint of the light, this results in a shadow map.
Afterwards, each pixel in our main view is reprojected to the light space and compared to what we stored in the shadow map.
Points visible by the light are in light, and those not visible by the light are in shadow.



Shadow mapping presents artifacts due to the quantization of depth information. Unless very big resolutions are used, artifacts usually occur.

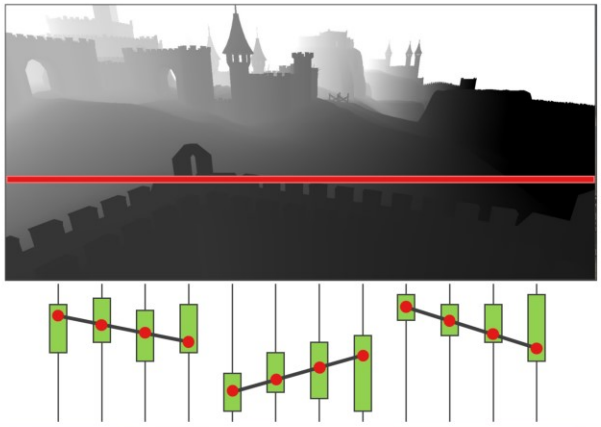
Shadow map recap

How much resolution is needed for a large scale scene?

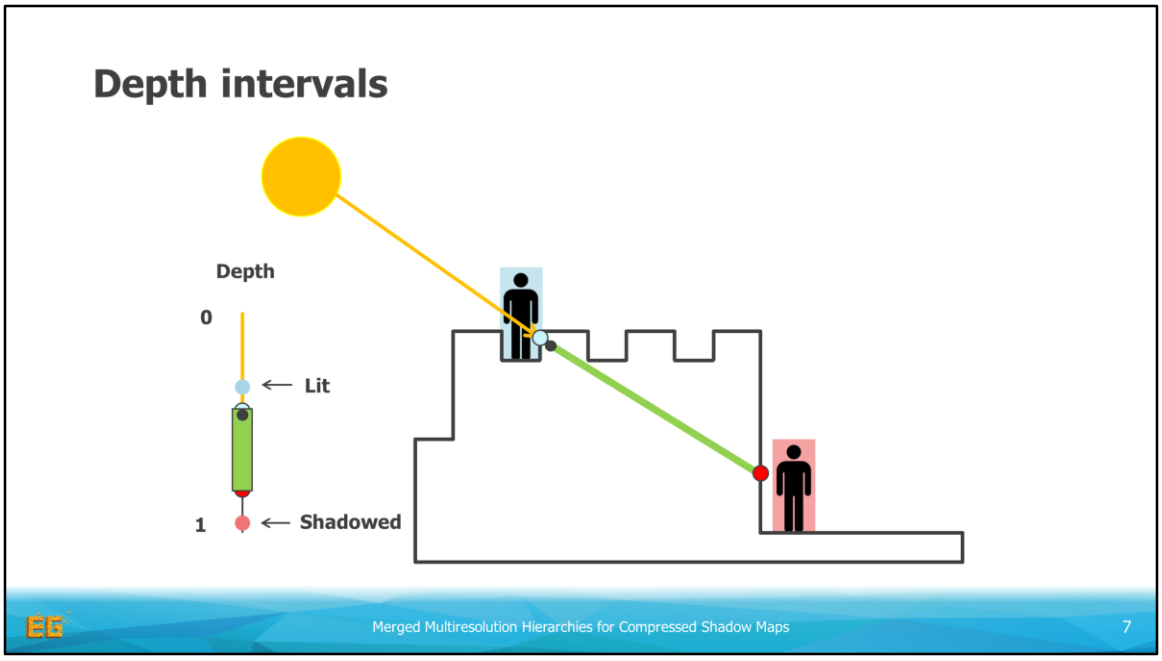
$4K^2$	64 MB	Not enough
$16K^2$	1 GB	Still not enough
$1M^2$	1 TB !!	

Previous work – Scanline based compression

[Arvo & Hivvikorpi 2005]



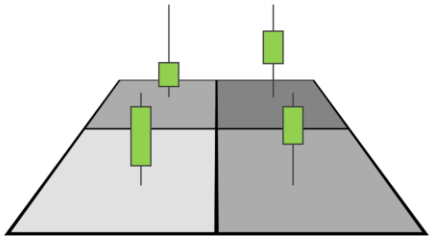
Scanline based compression [Arvo2004] compresses shadow maps by compressing each scanline separately as a series of lines that fall within the depth intervals.



Each texel in a shadow map represents a ray shot from the light towards the scene.
Normally we save the depth of the first intersection point.
After that intersection, the ray will enter an object.
Any depth that represents a point within that object is a valid depth to store, since we will only compare that depth to points outside of it.
This results in a valid depth range, or interval, for that pixel.
Other objects can only be outside of this range.

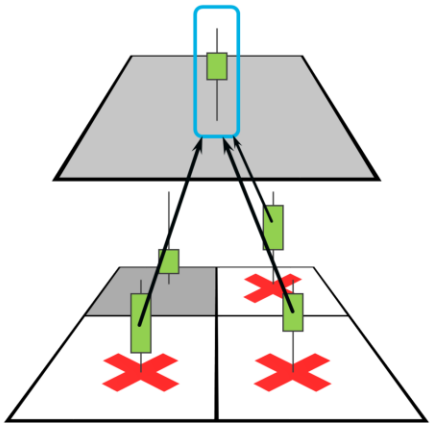
Merged Multiresolution Hierarchies

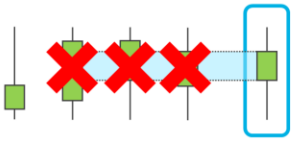
Multiresolution Hierarchies



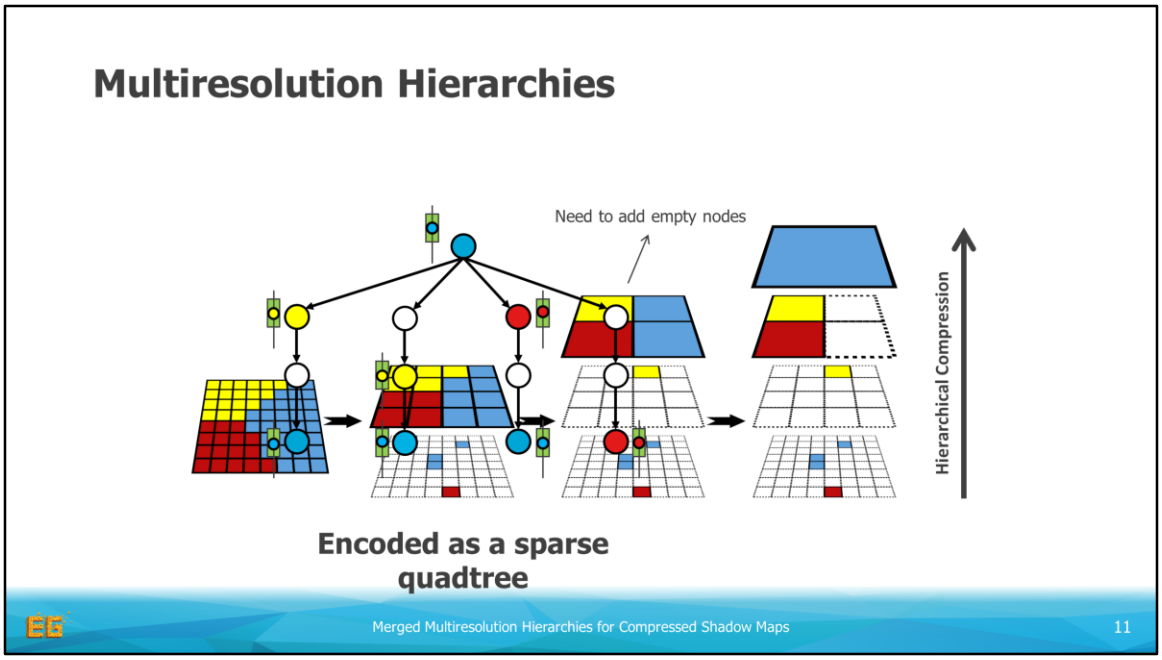
Another method we presented at EG this year is MH, which creates a sparse hierarchy of intervals to represent the original high resolution depth map.
Groups of 4 intervals are examined, and the larges intersecting set of intervals is identified.
A new interval is created that can represent the original intervals at a lower resolution, creating a hierarchy.
The redundant intervals are then eliminated, thus sparsifying the hierarchy.

Multiresolution Hierarchies

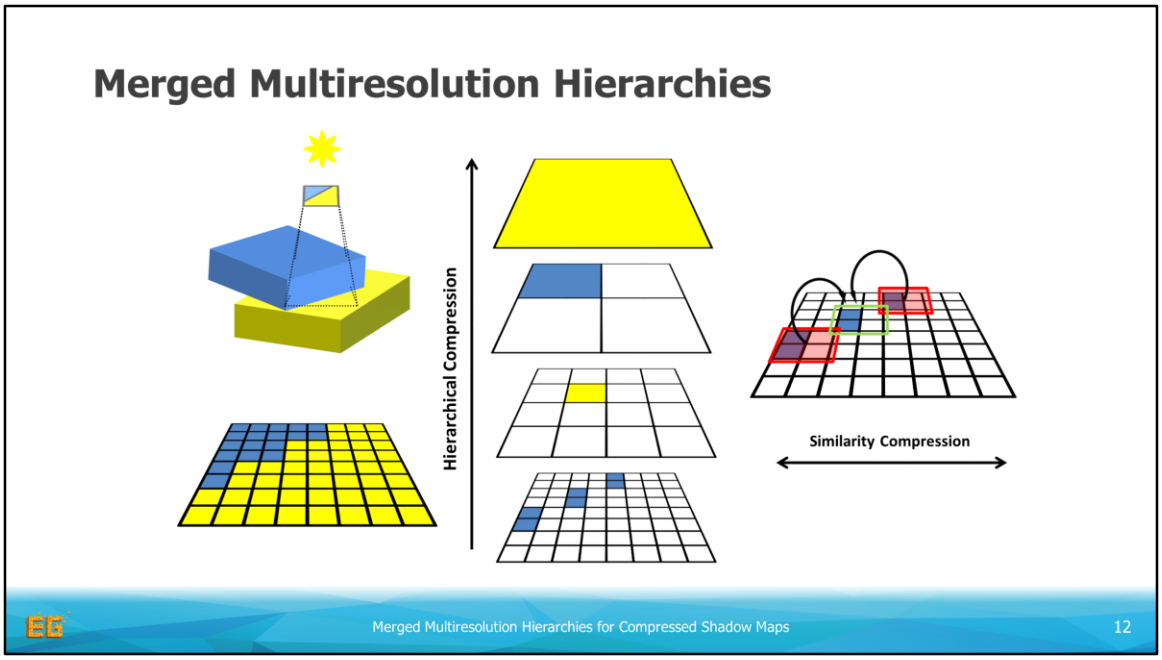




Largest intersecting subset



This method is repeated from the highest resolution until one single interval remains. Then, the hierarchy is encoded as a quadtree for fast access.
The quadtree does not need to store the interval, it simply stores the midpoint of each interval.
Compression in this case is achieved by representing many values as a single one at lower resolutions, a sort of hierarchical compression.

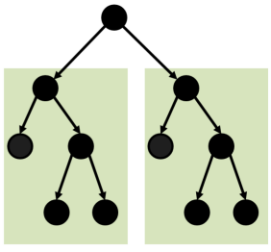


So, the idea of this work is to merge both previous approaches.
We want to create a sparse hierarchical representation of the data and also identify redundant parts in that hierarchy that can be represented by a single instance.

Merging subtrees

Why is this difficult ?

Typical DAG
1-bit value



Easy to find matches using a dictionary

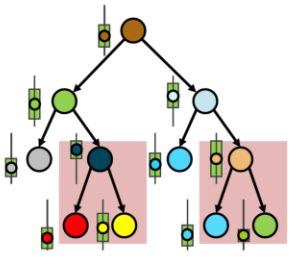
Multiresolution Hierarchies
Depth values (32 bits)

Exact matches are very rare

At this point you may wonder why wouldn't this be straightforward.
The problem is that in DAGs, we usually store 1 or 2 bit values at nodes, so exact matches are abundant and easily detected by using a dictionary.
In our case, we want to save full 32 bit values, so exact matches are almost non-existant.

Merging subtrees

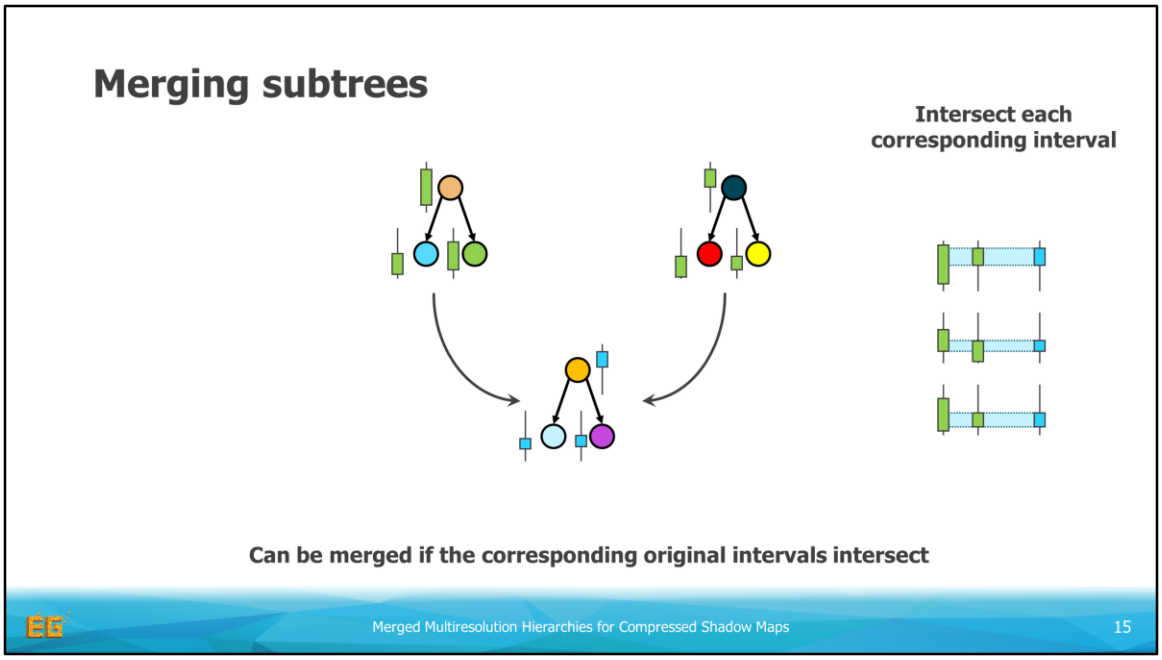
Can they be merged?



Insight: Depth intervals guide the subtree matching

So, what we do is this:

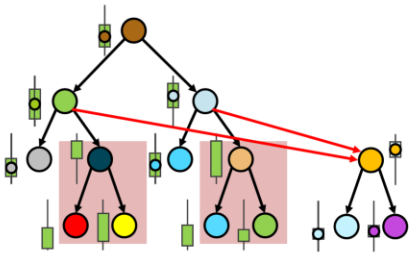
- We create a multiresolution hierarchy quadtree, as explained before.
- Instead of choosing a single value for each node, we keep the original intervals.



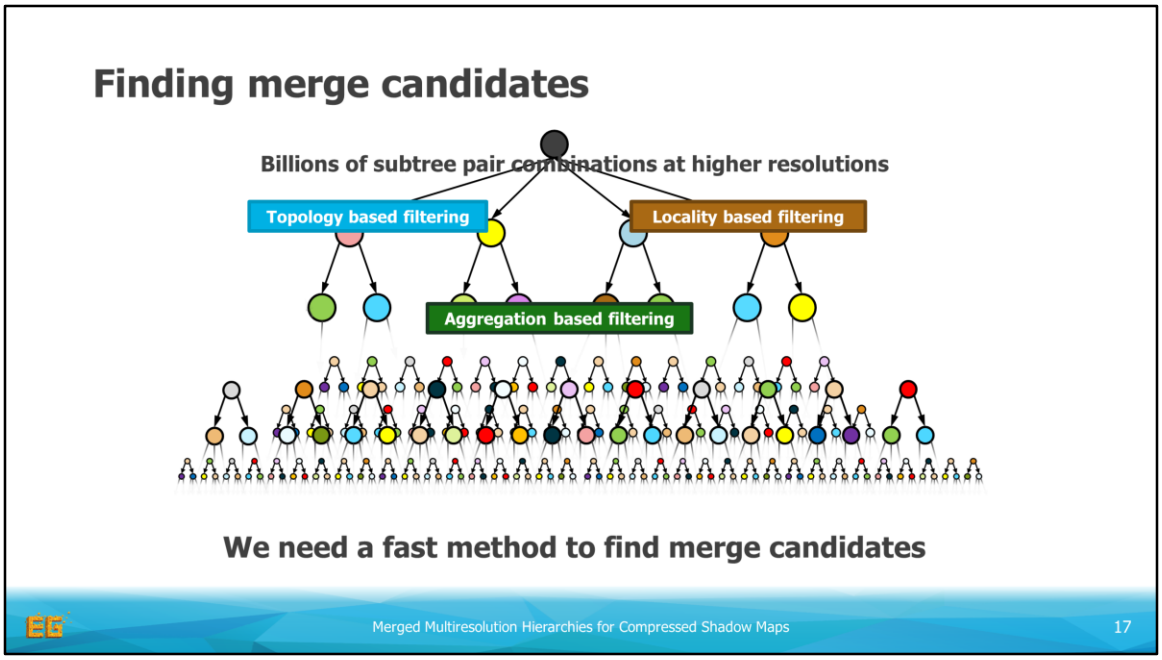
- We can then compare 2 subtrees by trying to intersect each corresponding interval.
- If each interval pair is intersecting, then the intersection represents all values that are allowable for both subtrees.

Merging subtrees

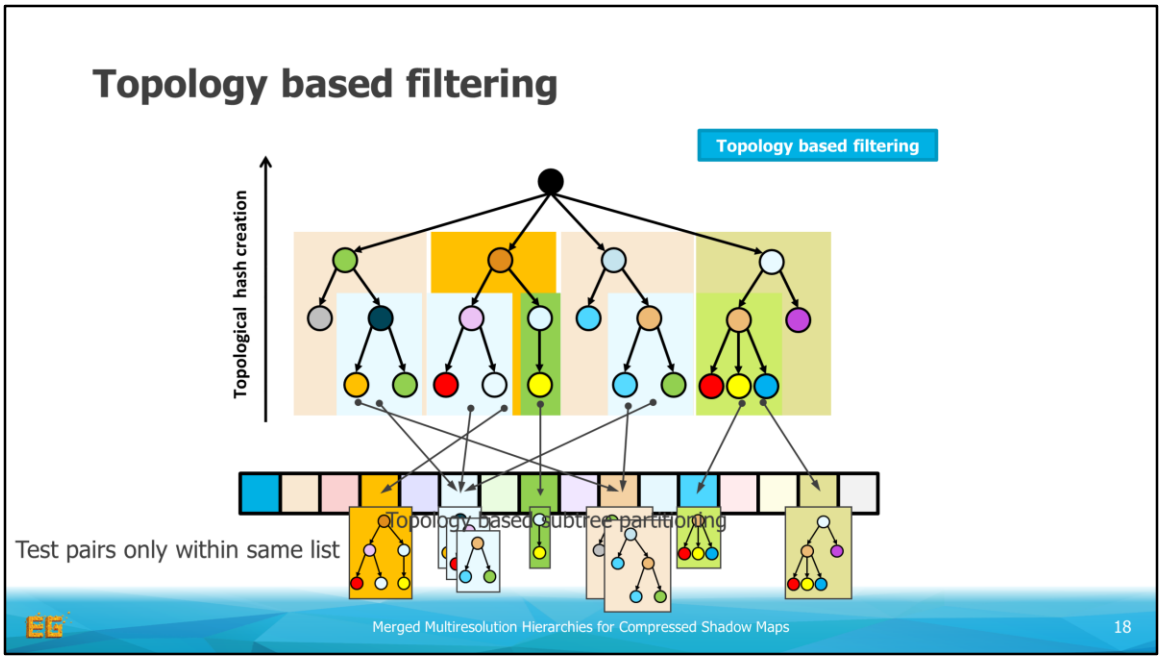
Intervals make merging possible



Finally, we can replace both subtrees by the intersected one.

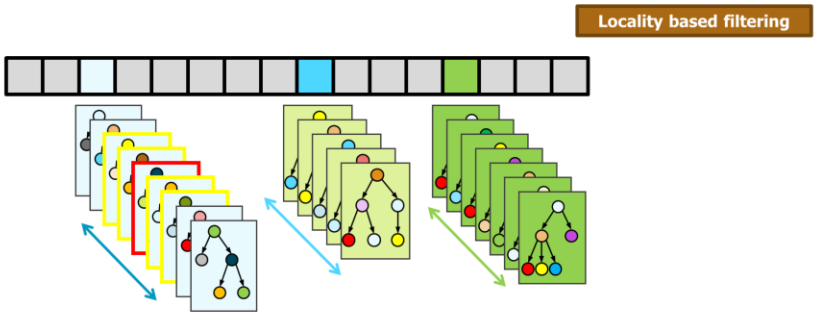


Since we cannot rely on a dictionary, we must test subtree pairs to determine if they can be merged.
The problem is that there are billions of combinations in high resolution MH quadtrees.
Therefore, we need a procedure to reduce the number of tests as much as possible.
We do this by filtering the possible merge pairs based on topology, locality and aggregated statistics.



A typical subtree may contain hundreds of thousands different subtrees, so we cannot compare every subtree pair.
In order to speed up merging, we will partition subtrees by topology, so only equal-topology subtrees will be tested.
We do this by creating a hash code for each inner node that encodes the topology of the tree in a bottom-up sweep.

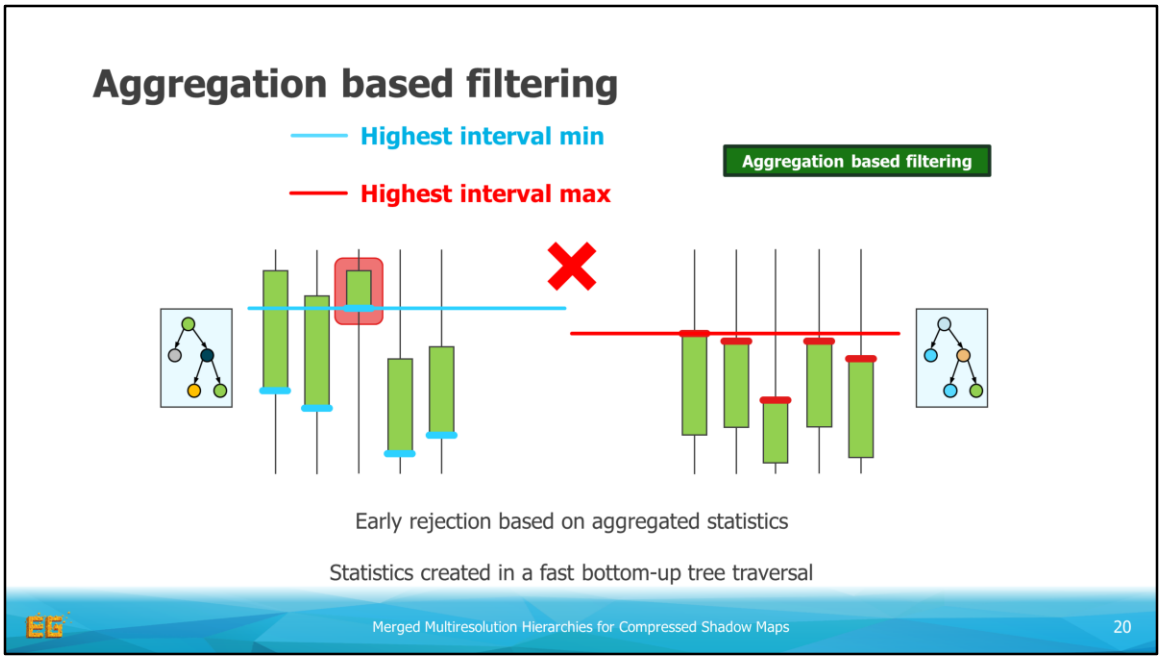
Locality based filtering



Test only against nearby subtrees in the list

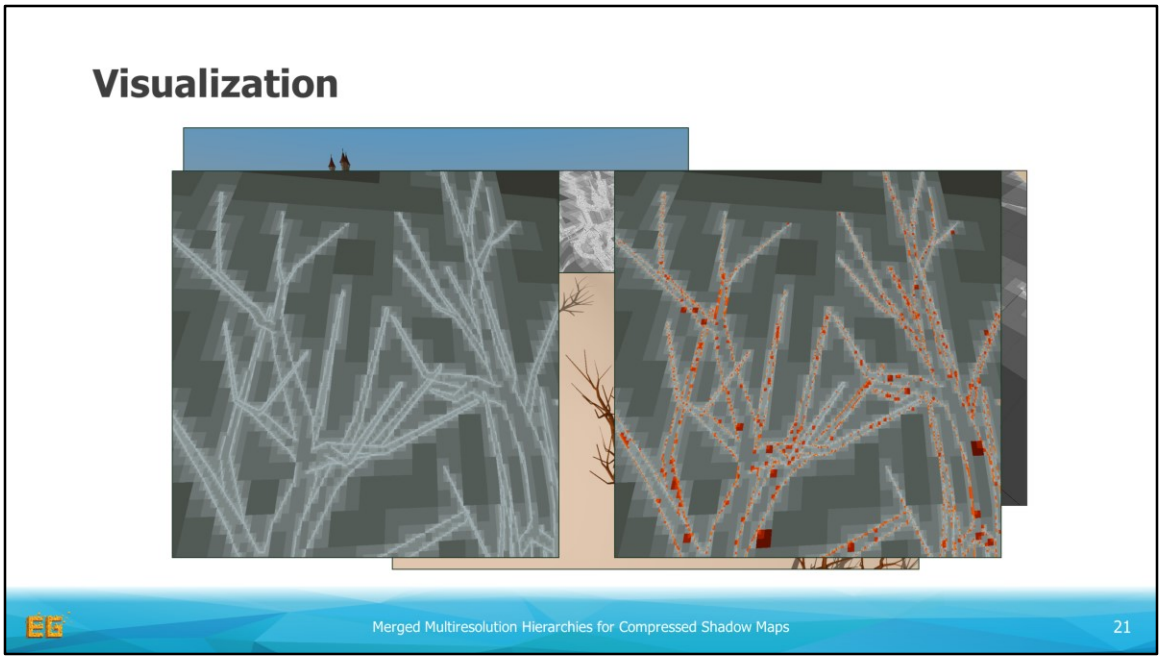
Corresponds to testing locally in image space

Once we have the hash table, each collision list contains all subtrees of equal topology.
We can therefore test subtree pairs within collision lists.
Some lists will grow very large, especially for small heights were topology types are not so many.
We restrict testing to a local neighborhood in the list. This corresponds to spatially near features, since subtrees are inserted in a depth-first manner.



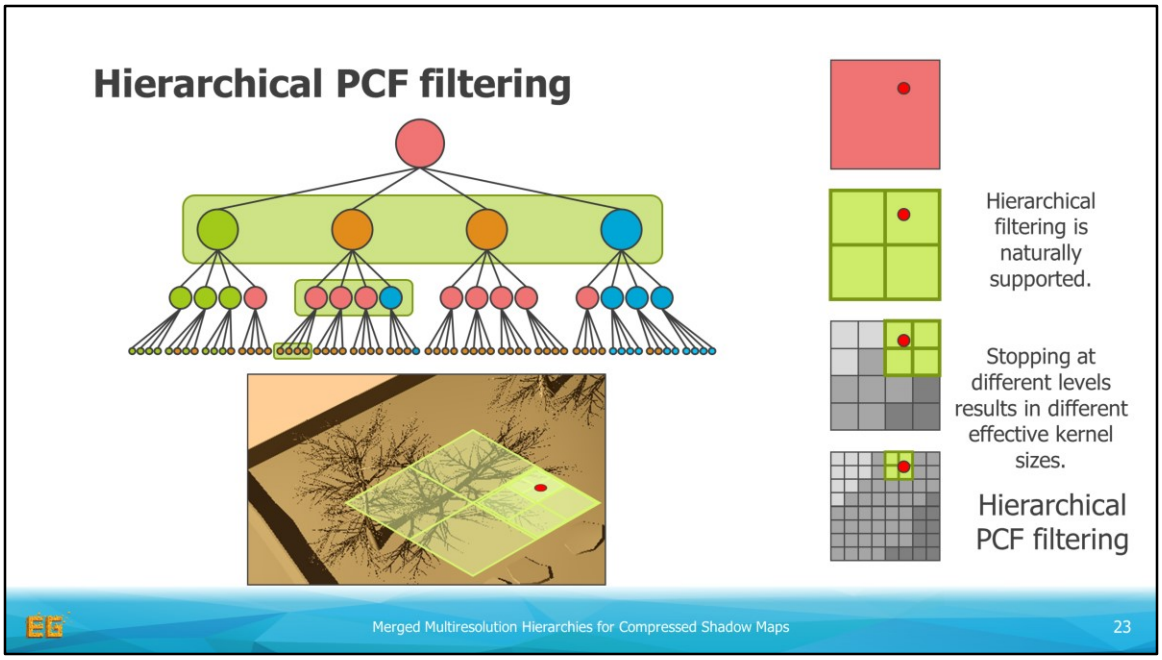
Once we have selected a pair of subtrees to test, as we mentioned, we need to verify if each corresponding node interval intersects.
This still is a lot of floating point comparisons, and we want to speed this up.

If I show you these two sets of intervals, you immediately see that they cannot all intersect.
The reason is that there is an interval whose minimum is higher than any maximum of the other set.
We can quickly reject merges by inspecting the lowest interval minimum and highest interval maximum of the sets.
These aggregated statistics can be computed very fast for all subtrees in a single bottom-up traversal. We do this at the same time as the bottom-up hash creation.

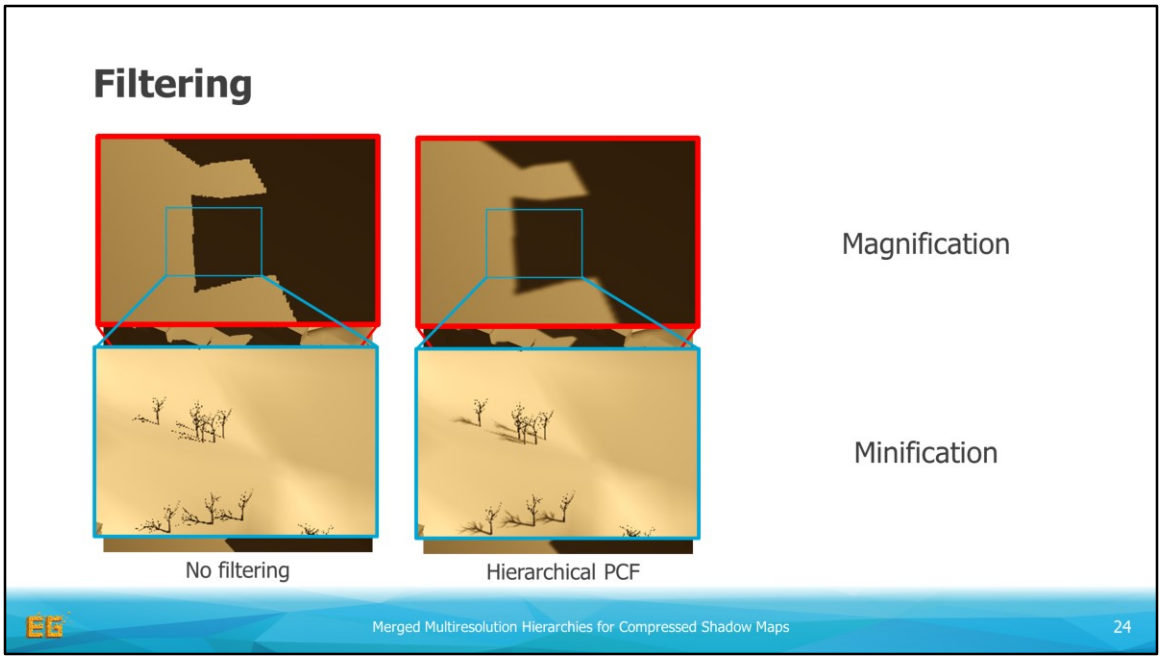


This is a small visualization of how the hierarchy looks.
On the left is the original hierarchy, where the lighted pixels denote deeper nodes in the tree.
On the right, the same visualization but we mark in red the merged nodes.

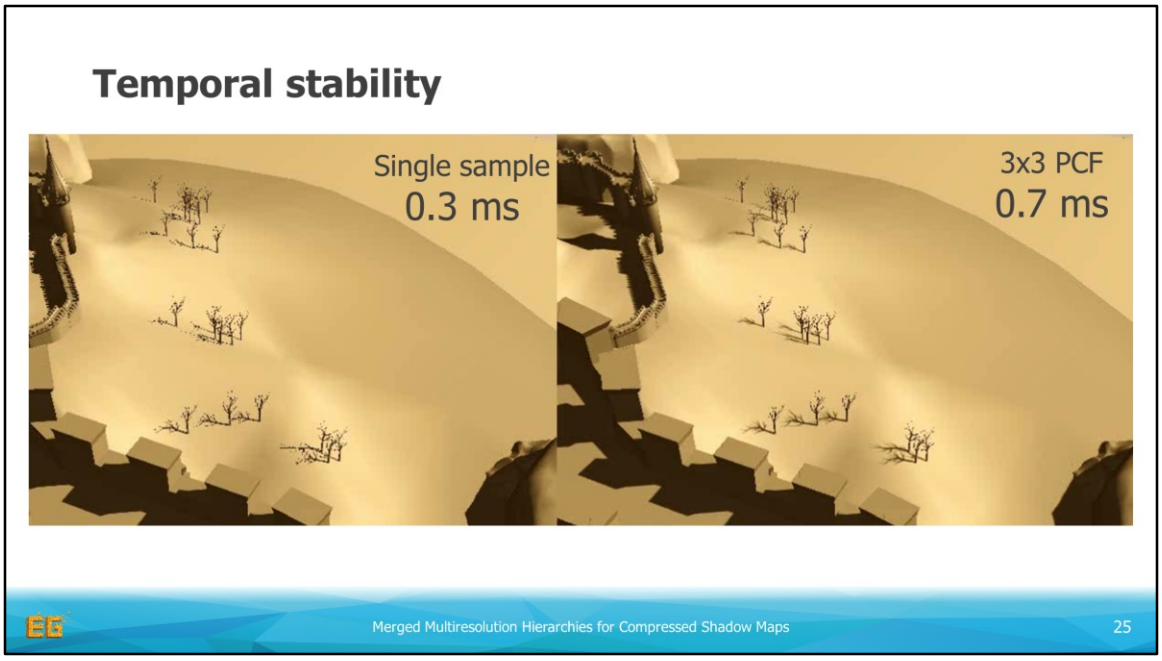
Filtering



Let’s look at an example of a small PCF kernel of 4 texels.
We can decide which level to sample to make the texels match the pixel size.

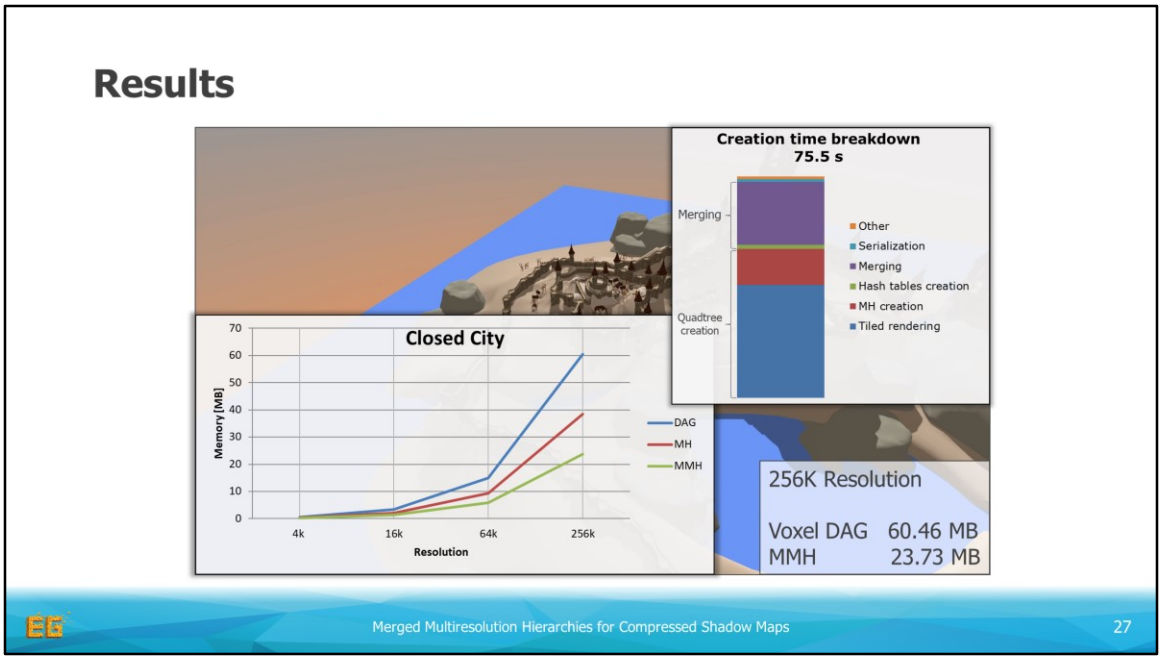


Let’s see an example in a case where we have a view where we need magnification for the close objects and minification for the trees in the background *click*.
As we saw, magnification works by regular PCF filtering. *click*.
For minification, stoping at the correct level gives us smooth looking shadows at any distance.
In this example the trees where not antialiased, so you can compare what happens when you have too small features.



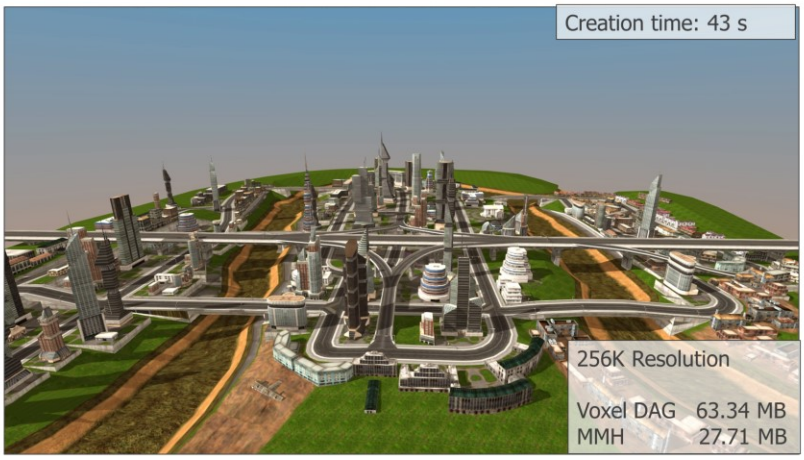
The end result are very stable antialiased shadows.

Results



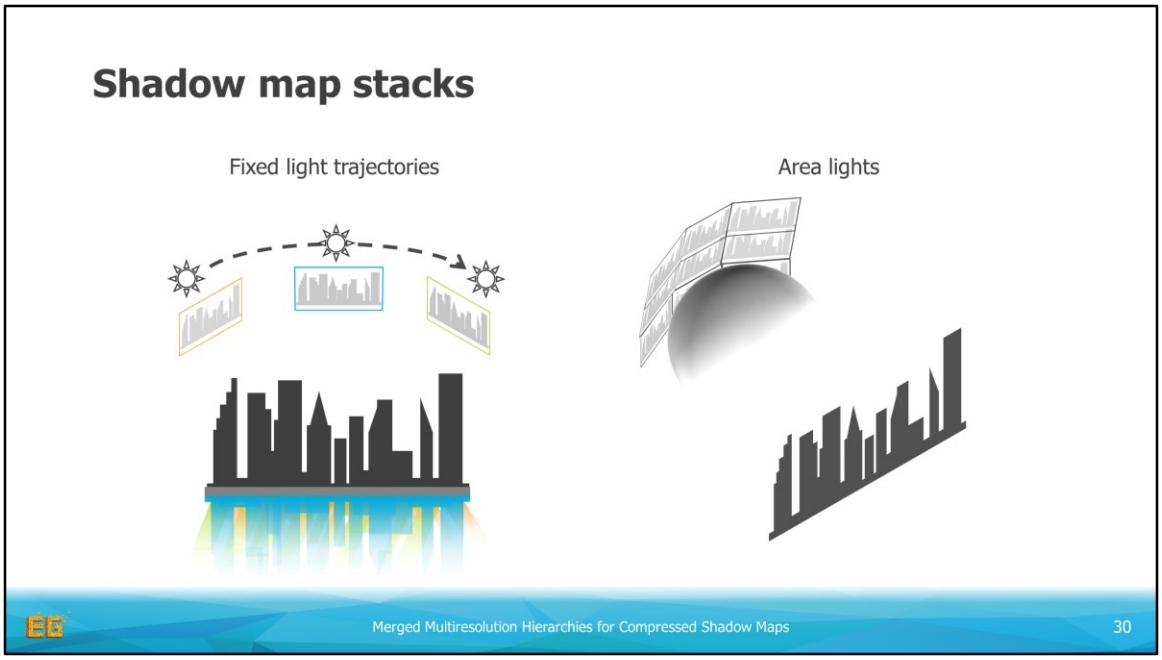
This other example is a more typical use case where there are detailed features, but they are smaller in scale.
Again, our method outperforms previous ones.
In this case most time is spent rendering the original shadow map tiles, while merging is now around 30% of the total time, which is more reasonable.

Results



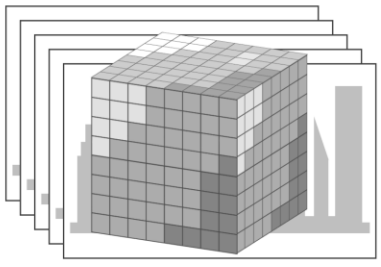
This is an example of a large scale urban scene.
Again, we are able to achieve much higher compression rates than competing algorithm.
This scene is pre-processed in only 43 seconds.

Shadow map stacks



In some cases, we may want to precompute and compress a set of similar shadow maps.
For example, we may have a light moving in a fixed trajectory and want to store several SM along the trajectory.
Or an area light, where we may want to store several SM for different points in its area to produce soft shadows.

Shadow map stacks



One extra dimension

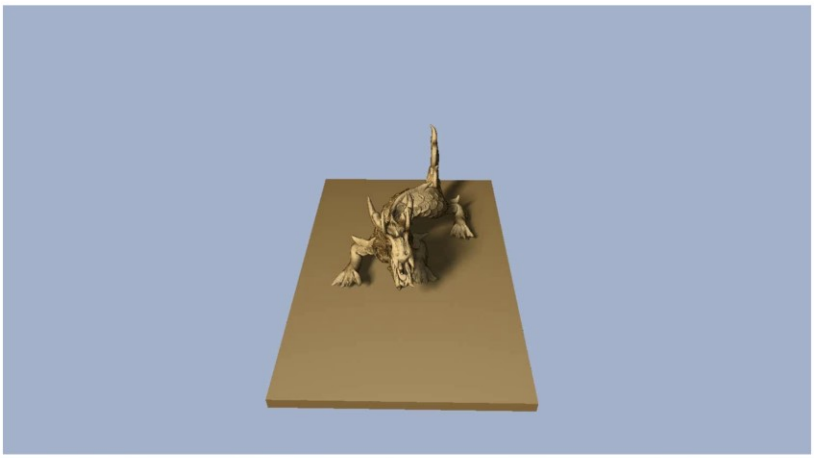
Encode it with an octree!

By using our same approach we begin with a set of shadow maps, which we can think of as a cube of texels, each with one depth interval. *click*
So we can do the exact same procedure we have showed, but with one extra dimension, and we will end up with an octree instead of a quadtree.

Fixed light trajectories

This is an example of storing a set of SM for a moving light, where we sample different slices over time.

Area lights



By storing several SM from an area light and sampling a subset for each pixel, we can get soft shadows which have geometrically correct penumbra sizes.

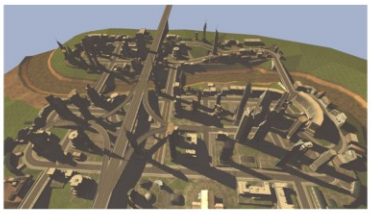
Conclusion

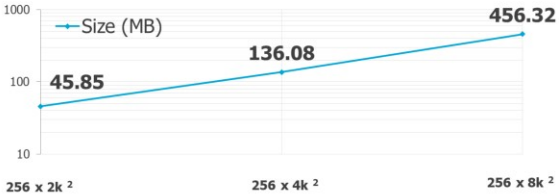
- Practical fast method for compressing and using high resolution shadow maps
- Compression rates up to **40%** higher than competing approaches (not tested against newer DAG variants)
- Highly efficient hierarchical PCF
- Reasonably fast creation times (~ 1 min for 256k resolution)
- Can be sampled/filtered directly or used as input for other methods (e.g. cascades)

Thank you

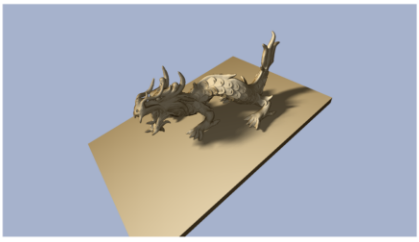
Extra slides

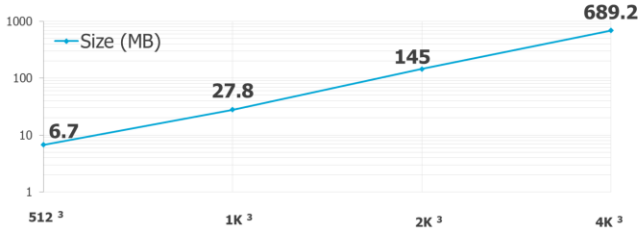
Fixed light trajectories memory consumption





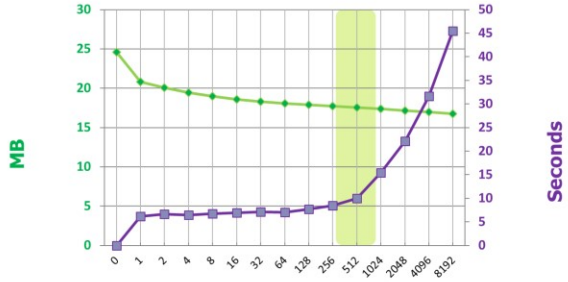
Area lights memory consumption



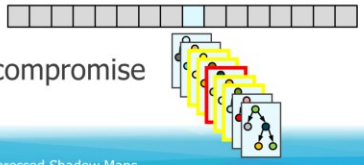


Results

Increasing local search distance results in diminishing returns



Distance of around 500 is a good compromise



Augmenting the size of the neighborhood results in diminishing returns. The graphics shows that beyond around 1000 neighbors, testing more subtrees does not increase compression significantly.