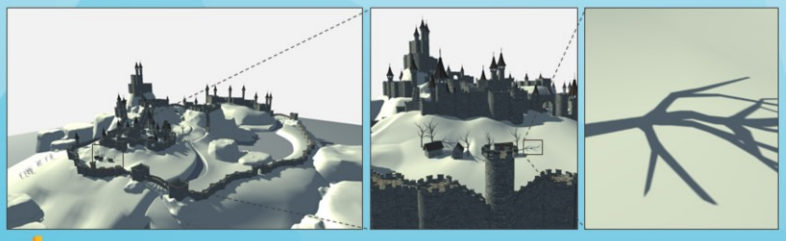
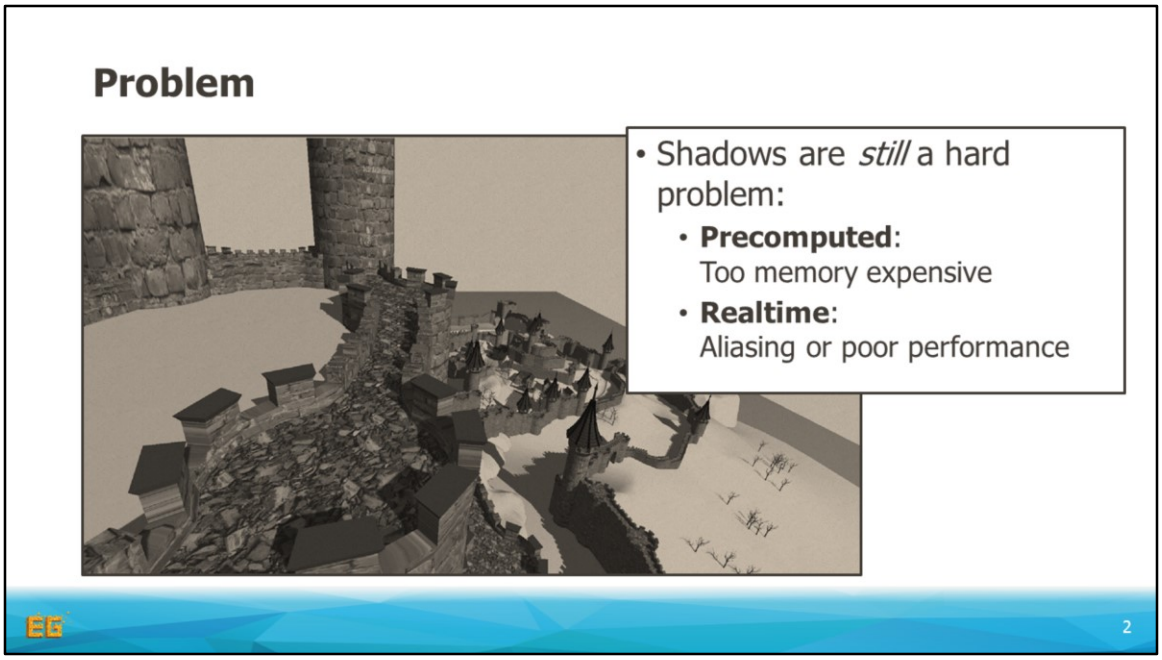
Compact Precomputed Voxelized Shadows

Erik Sintorn, Chalmers University of Technology



EUROGRAPHICS 2018

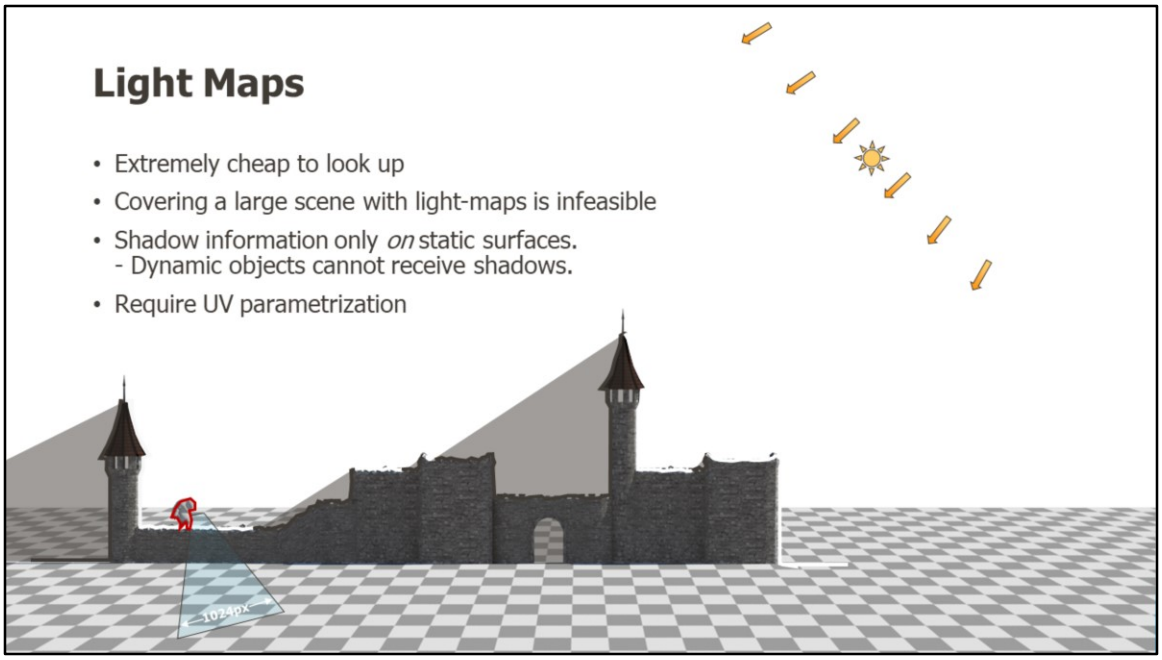
Hello,
Thanks for the introduction



In this paper we suggest an efficient data-structure for precomputed shadows from point light or directional light-sources.

Because, in fact, after more than four decades of research, rendering high quality shadows, even hard shadows, in large open scenes is STILL problematic.

Current precomputed solutions typically require much too much memory, and realtime solutions are either too slow for practical use or exhibit serious aliasing problems.

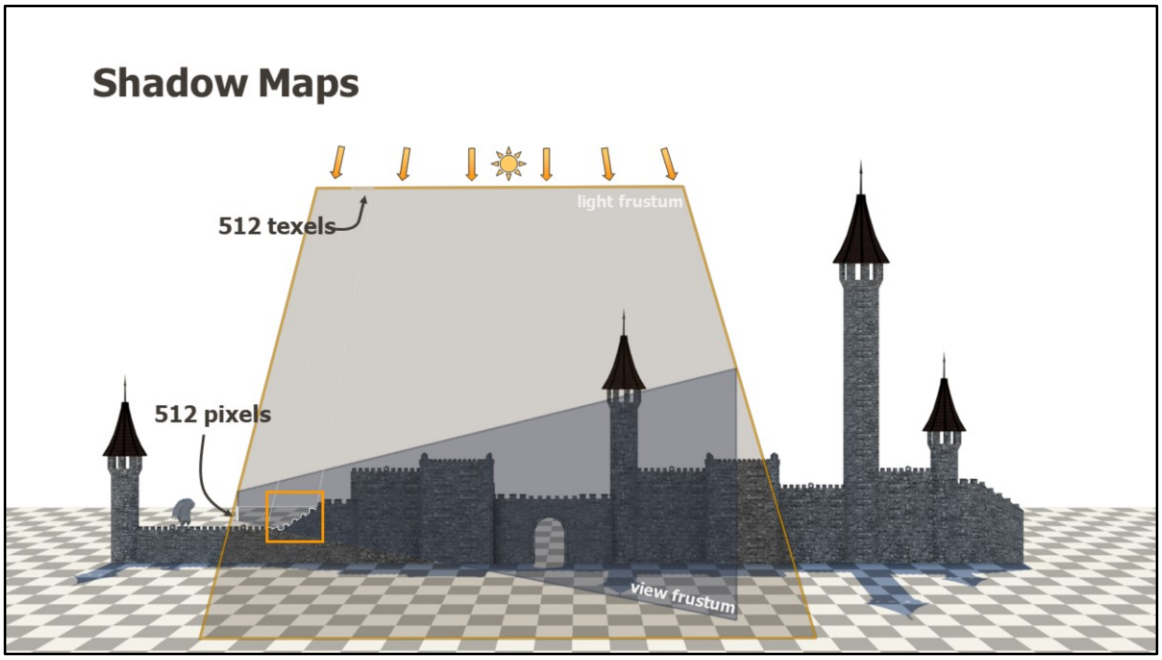


For static lights and geometry, visibility can be precomputed and stored in a large light-map. <click>

Now, let's say our player is standing here <click> and is looking down at the ground in front of him.
Since we need a resolution of the shadows that at least matches the screen resolution for this little part, the size of the complete light-map can be arbitrarily huge. Easily going beyond what is reasonable even for disc storage.

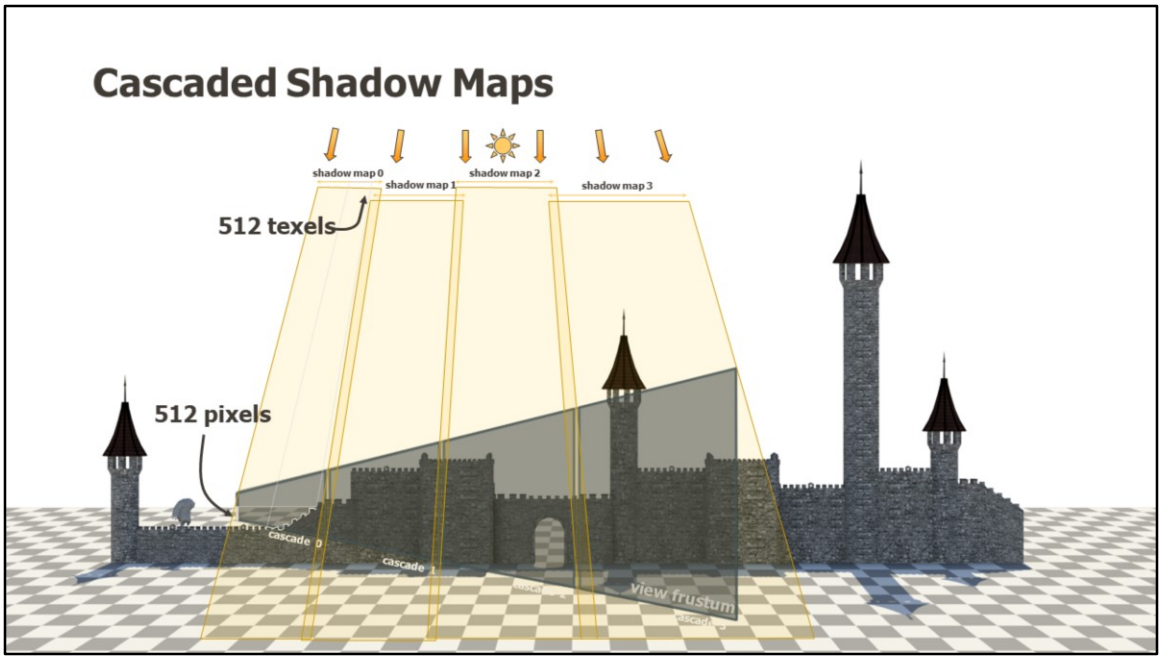
So, <click> while extremely cheap to perform lookups in, covering an entire scene in sufficiently high resolution light maps is infeasible. Also, light maps only provide shadows *on* static objects and cannot be used to cast shadows on dynamic geometry. Finally, they require a unique UV parameterization, which can be difficult to do well.

TODO: Previous work



Therefore, hard shadows are usually computed using some real-time method instead. Usually, some variant of the shadow mapping algorithm.

To avoid undersampling the shadow maps, we must have at least as many shadow map samples as we have pixels covering a small surface. So, a surface that covers, say, 512 pixels on screen must cover an equal amount of texels in the shadow map. If this surface only projects to a small part of the shadow map, as in this example, the entire shadow-map will have to be huge.

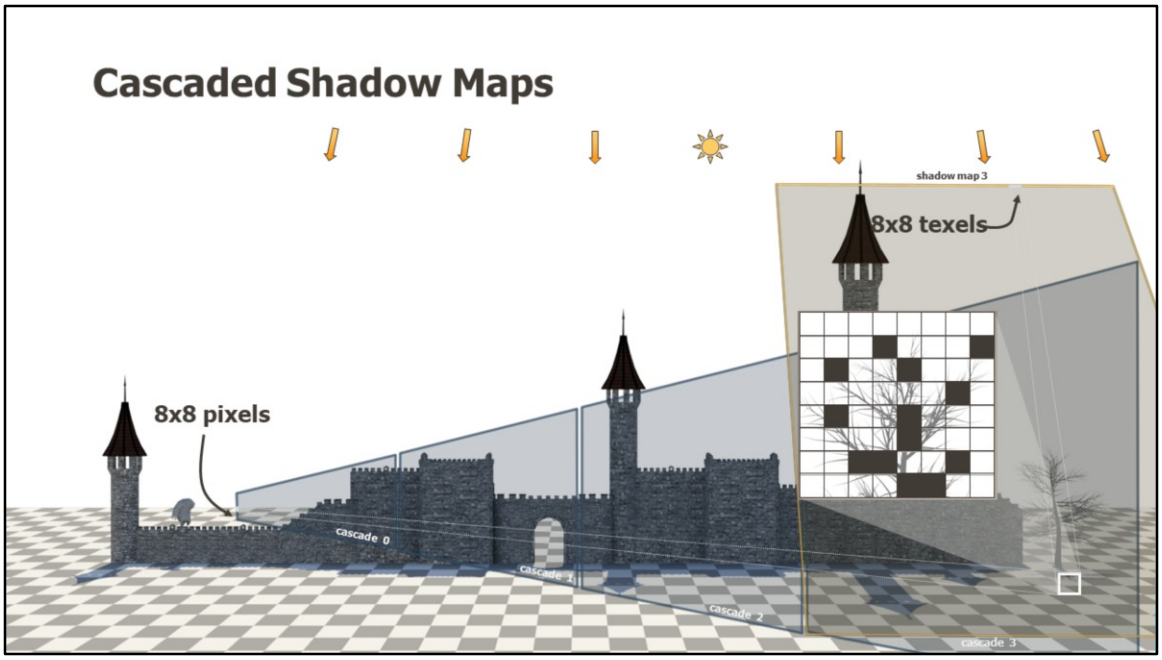


This problem is commonly alleviated using a technique called *cascaded shadow maps*. Here <click> the view-frustum is split in to several parts (or cascades).
<click>
And then, a shadow map is rendered for each of these cascades.
<click>
Now, the surface covers approximately as many samples on screen as it does in the shadow map of the first cascade, and all is well.

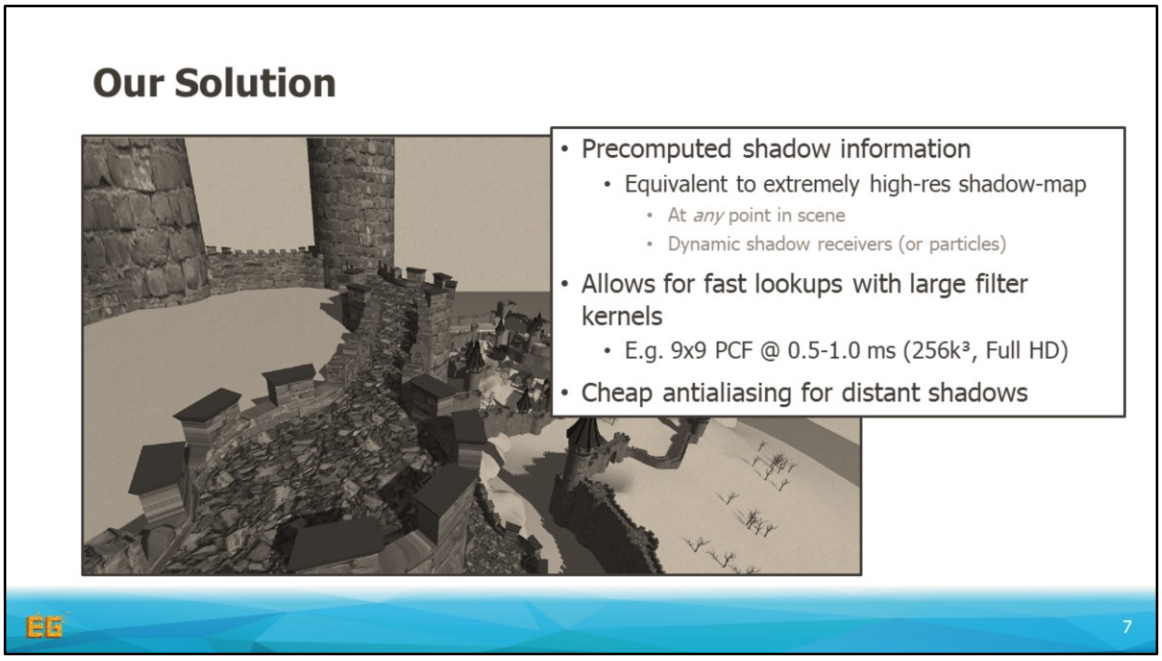
One problem is avoiding visible transitions between shadow cascades.

Another obvious drawback is that we now have to render several shadow maps each frame, which will significantly impact performance. Especially as we have to re-render *all* geometry, not just dynamic objects.

Note that there can be significant overlap between the shadow-maps, and that we might end up submitting the same geometry to every shadow-map.

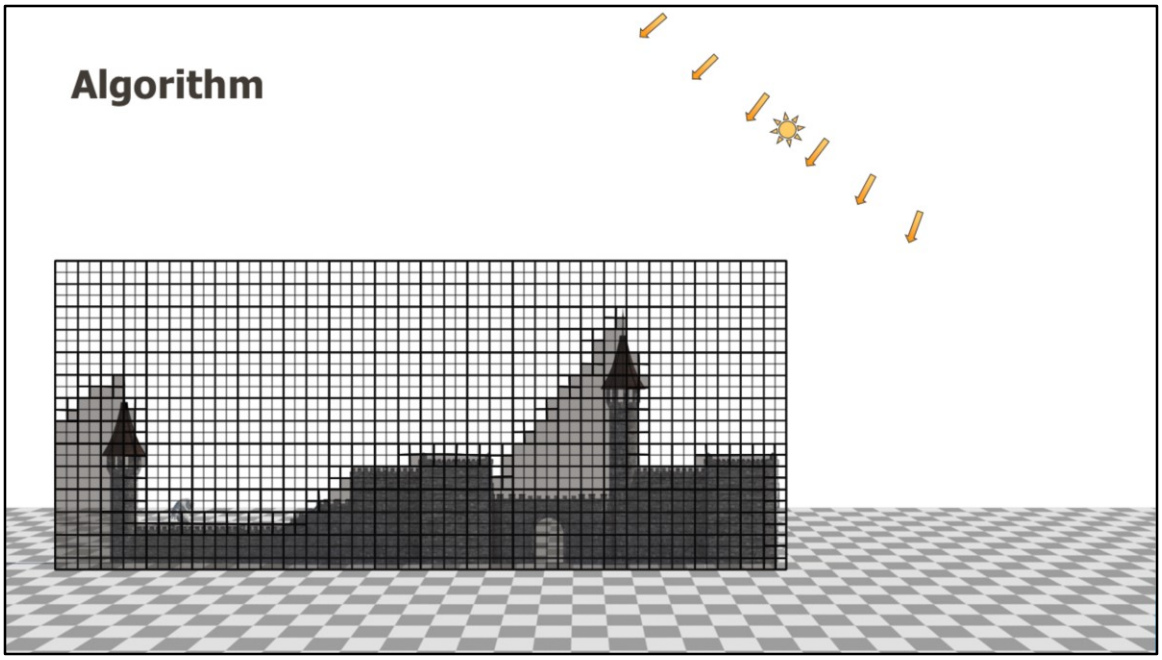


But we also run into the problem of *initial sampling aliasing*. Let's say that there is a tree way of in the background.
<click>
Using Cascaded Shadow Maps, we have made *sure* that this area is covered by just a few texels, since it only covers a few pixels.
<click>
Unfortunately, this resolution is much too small to capture the complex geometry, and we will see serious aliasing artifacts in distant shadows.

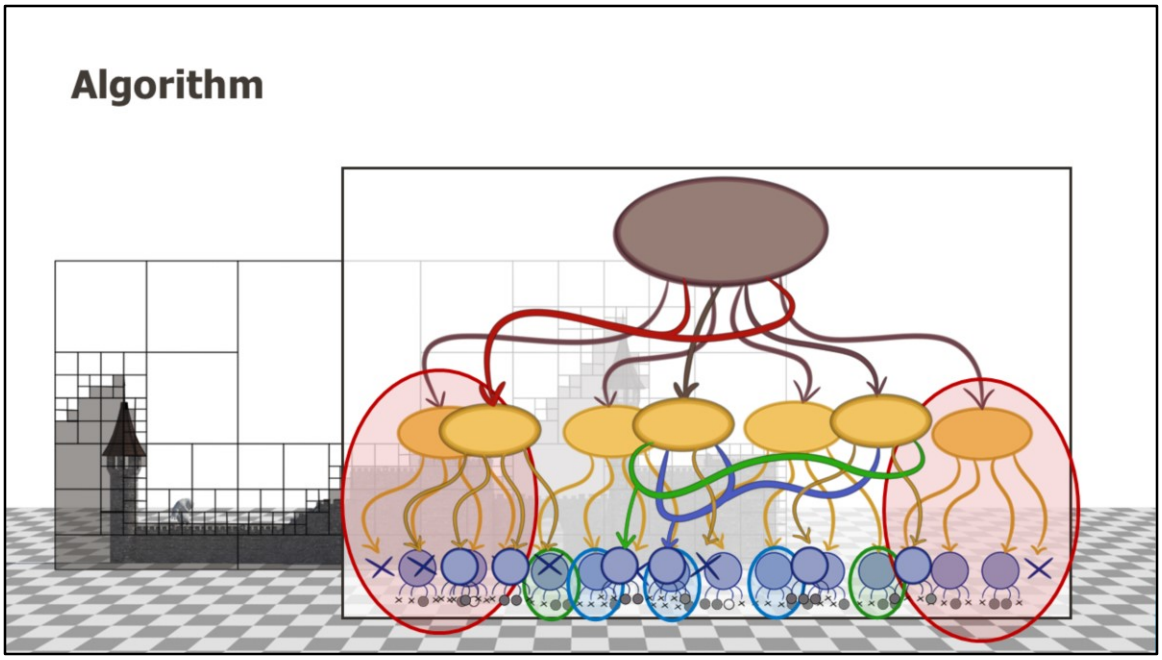


Which brings us to our solution.

We store precomputed visibility information at extremely high resolutions, for *any* point in the scene, so unlike light-maps, it can be used for casting shadows on dynamic objects or, for that matter, particles.
<click>
Our data structure is extremely compact, and can be used for fast shadow lookups, especially when large filter sizes are wanted.
<click>
Additionally, our data structure can be used for high quality antialiasing of distant shadows, alleviating the problem discussed earlier.



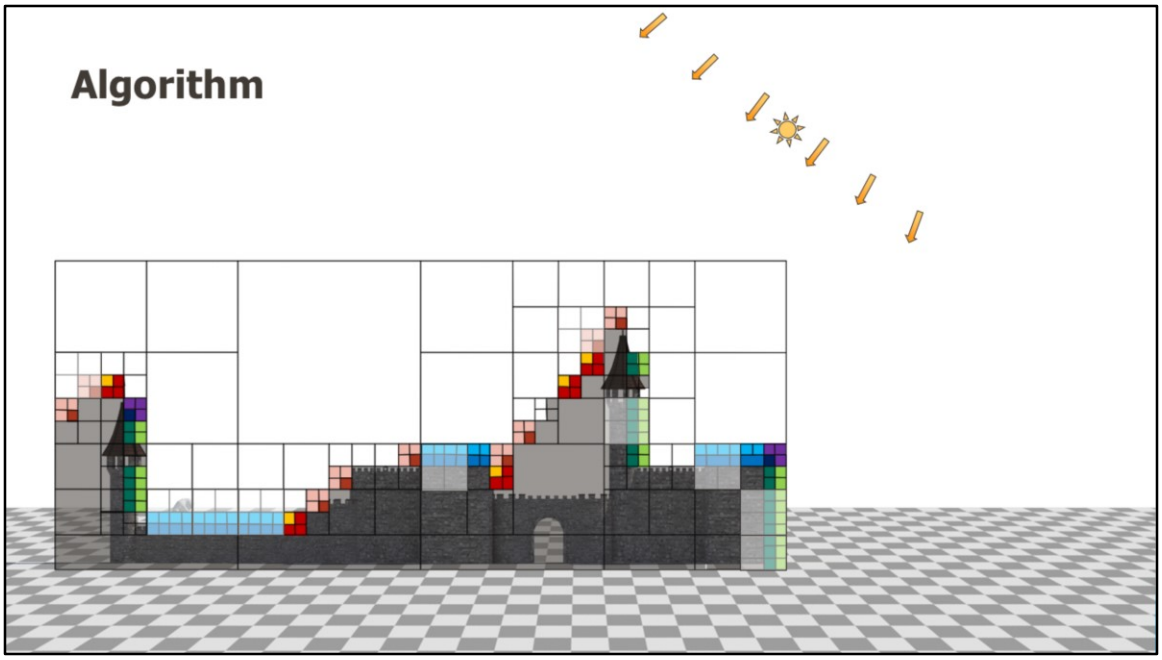
The basic idea is to store light visibility for every single cell in a voxelization of the scene. Every cell then only requires one bit of information, but storing a whole grid would still be madly expensive at higher resolutions.
<click>
It makes more sense to store the information in a tree-structure. The size of this data-structure will be proportional to the surface of the shadow volume of the scene, but will usually consume more memory than a simple shadow map.



But, since we have a sparse voxel octree, with binary values in the leaves, it makes sense to compress this SVO by merging common sub-trees, like we suggested at SIGGRAPH last year.

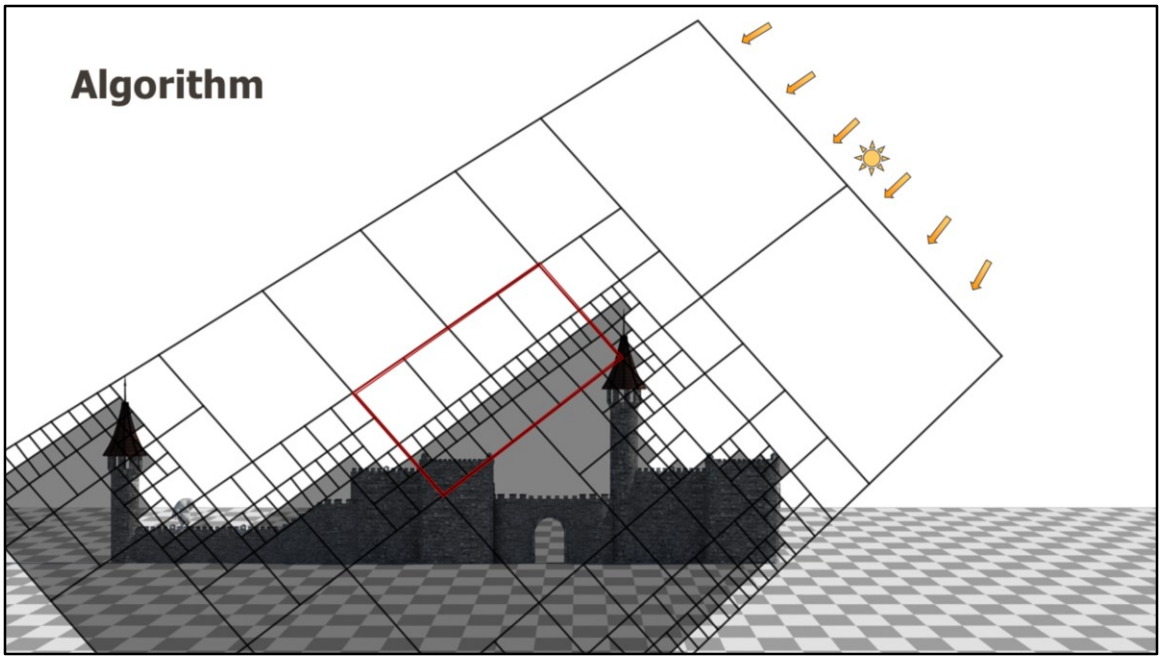
The idea is to find all identical sub-trees <click> and to remove all but one. The parents will then all point to the same unique subtree.

TODO: Ringar på andra bilden med



We end up with a data structure with much fewer nodes. At high resolutions, the compression rates can be very good.

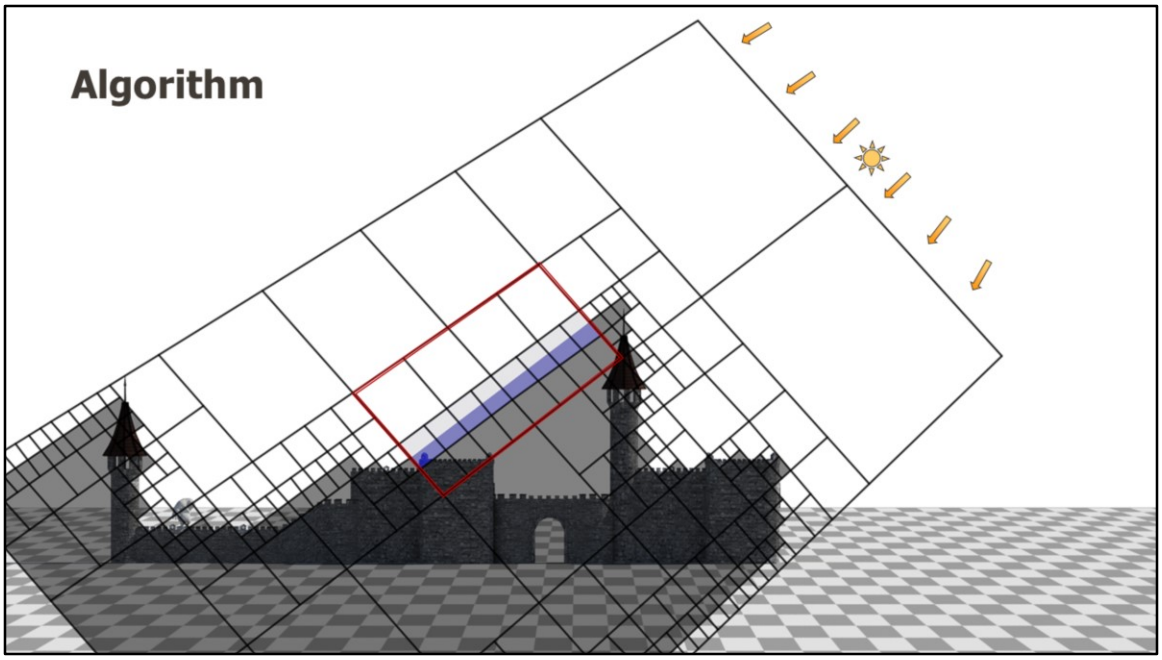
This produces much smaller data-structures than the corresponding shadow map at high resolutions.
However, we found that one small change could improve the situation tremendously.



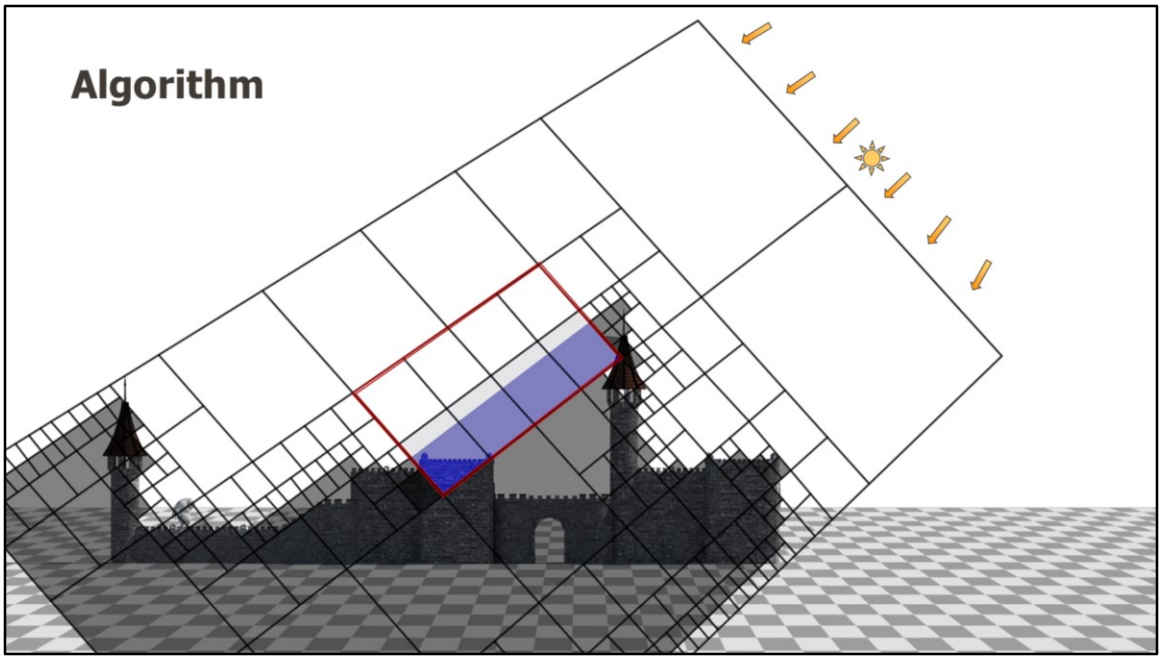
And that is to do the voxelization in light space instead.

Just voxelizing in light space and building a tree is not very interesting. The size of the resulting data structure is still proportional to the surface of the shadow volume, and the number of nodes is about the same. But <click>

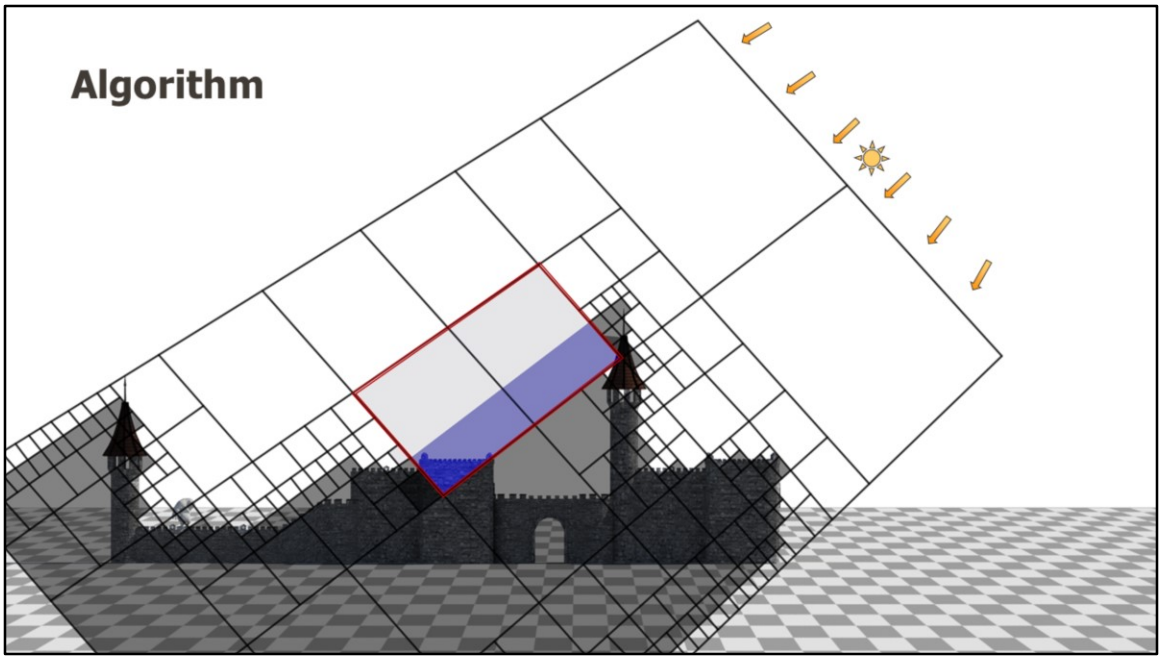
Consider what happens when we perform common sub-tree merging on this part of the tree



We start by finding all identical leaf nodes. In this region, they are all the same.

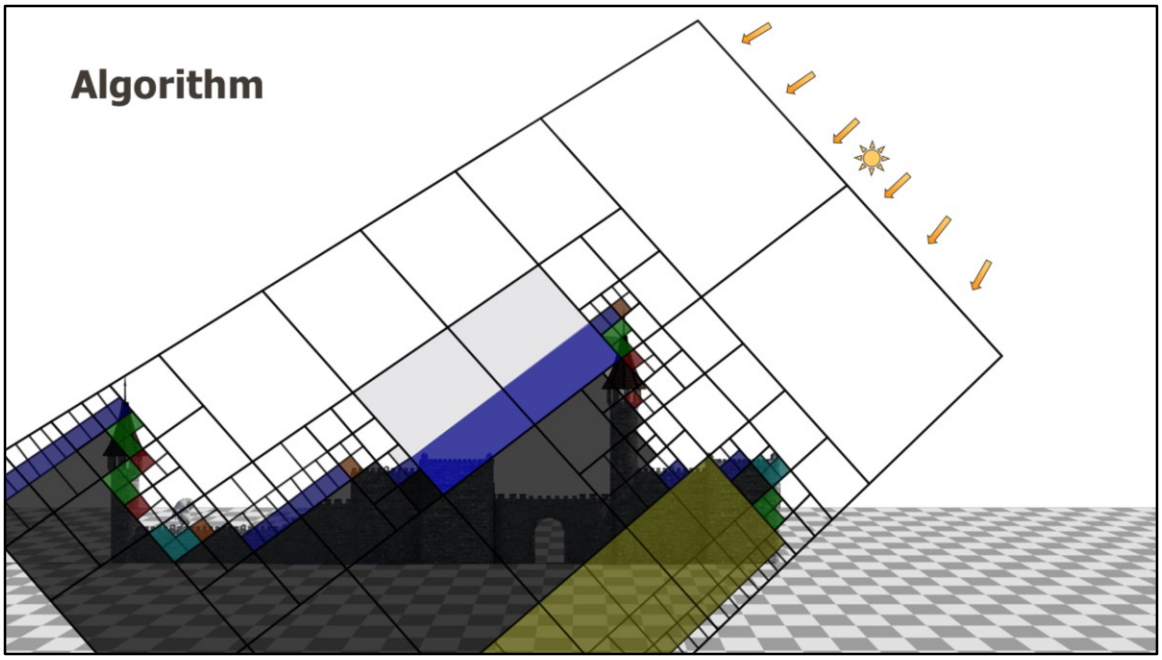


We then move up one level and find that all nodes are equal here too.



And at the next level, both nodes are also identical.

So, for this region, we only store *one* node per level. And this holds no matter how high the resolution is.

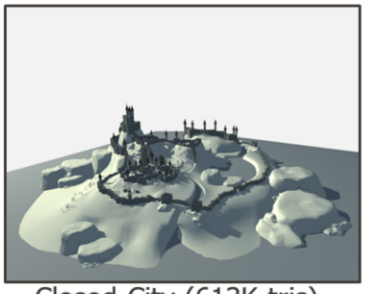


Here is the result for the entire tree.

It turns out that we only have to store unique leaf nodes along the *shadow casting surfaces*, which is usually a much smaller area than the area of the shadow volume cast by the scene.

Memory Consumption

Resolution:	1024^3	4096^3	$16K^3$	$32K^3$	$128K^3$	$256K^3$
16-bit SM	2 MB	34 MB	537 MB	2 GB	34 GB	137 GB
Voxelized	489 KB	4 MB	36 MB	96 MB	639 MB	1638 MB
Relative size	23.32 %	12.41 %	6.69%	4.48%	1.86%	1.19%

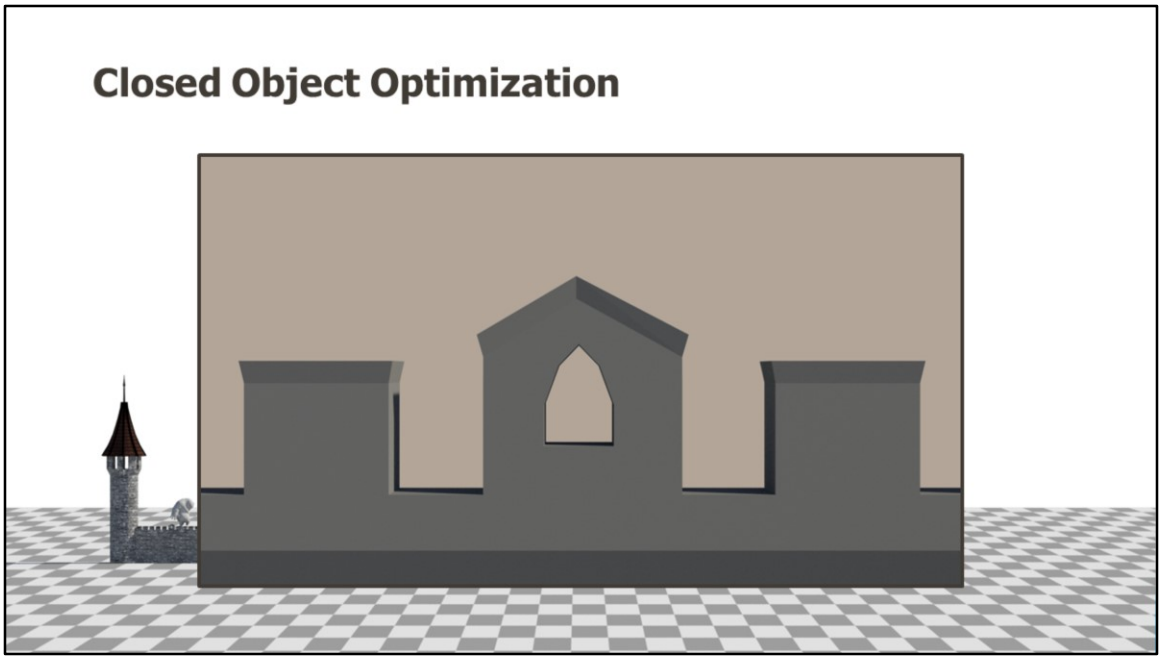


Closed City (613K tris)

For our main benchmarking scene, we now get a data structure that is about 10% of the corresponding shadow map already at resolution 4kx4k

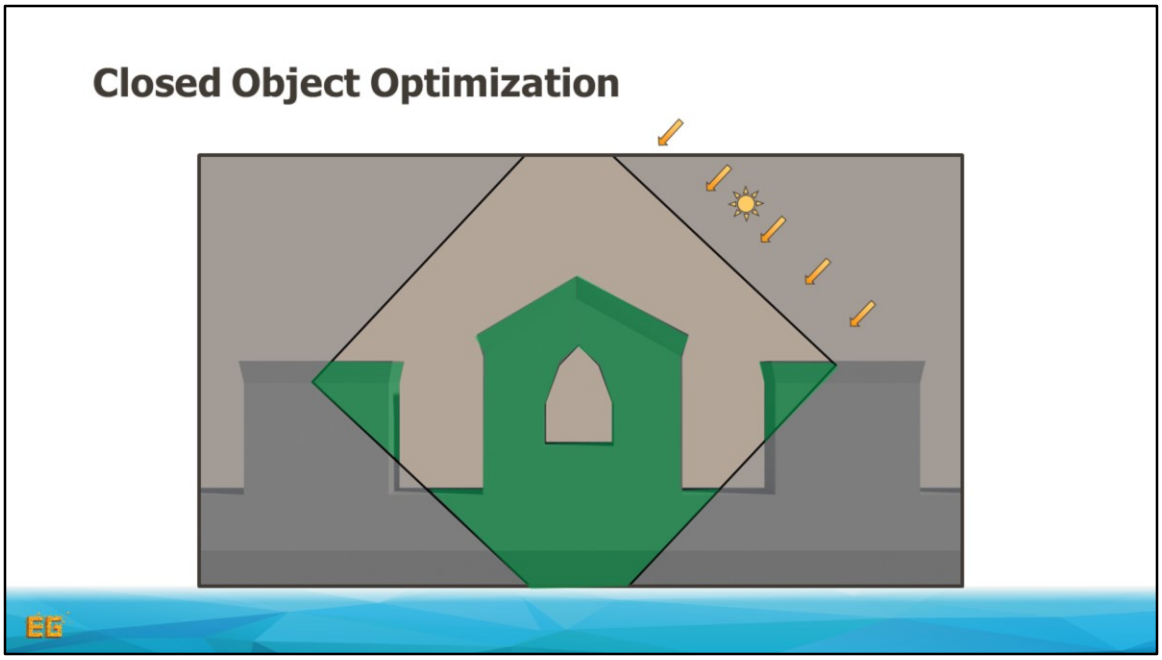
More importantly at high resolutions, for example 256k cubed <click> (which produces fairly sharp shadows in this scene) our data structure still fits in GPU memory at 1.5GB, while the corresponding 16 bit shadow-map would cost 137GB, which is really too large even for disc storage.

Next, I will talk about an optimization to the algorithm that improves compression further.

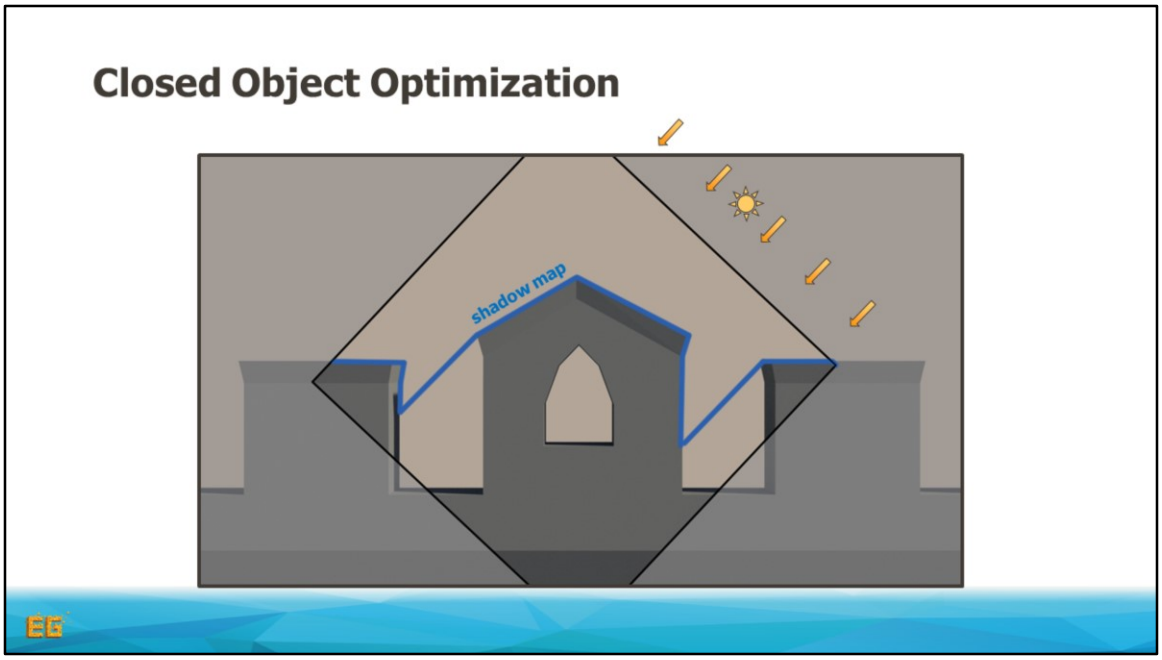


IF we know that some of the shadow casting geometry consists only of closed geometry, we can do much better.

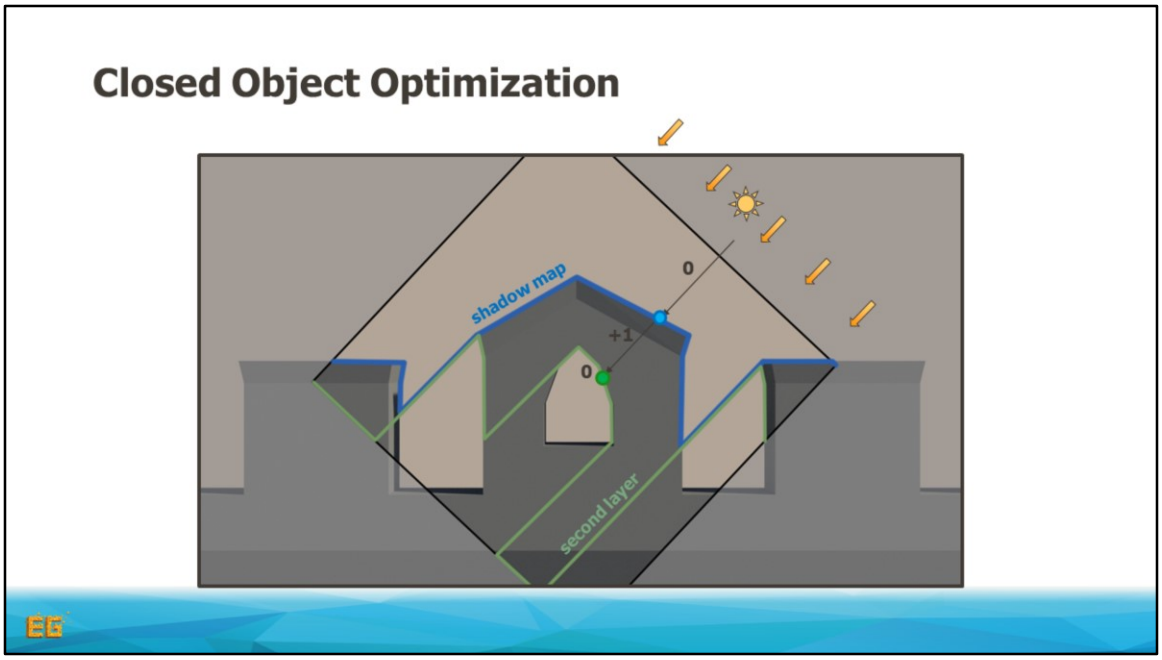
Let's consider this little part of the wall <click>.



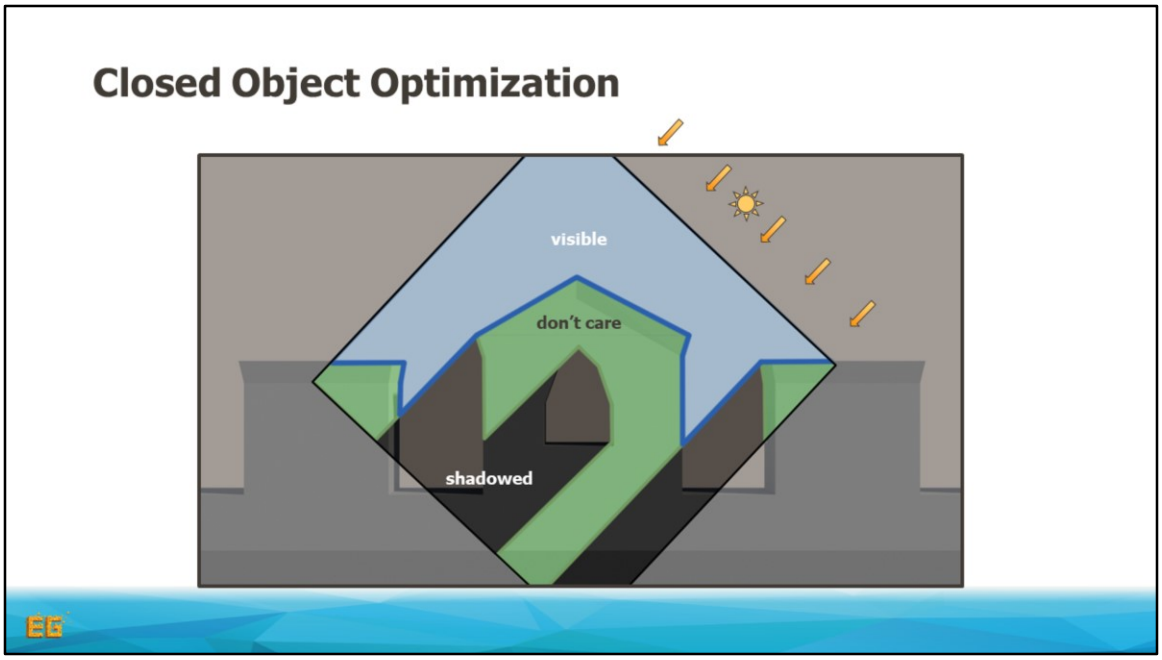
And let's say we want to build the subDAG corresponding to this region.
<click>
We assume that the user will never be interested in the visibility *within* a closed object, and so the actual value returned by our data structure in these regions does not matter.



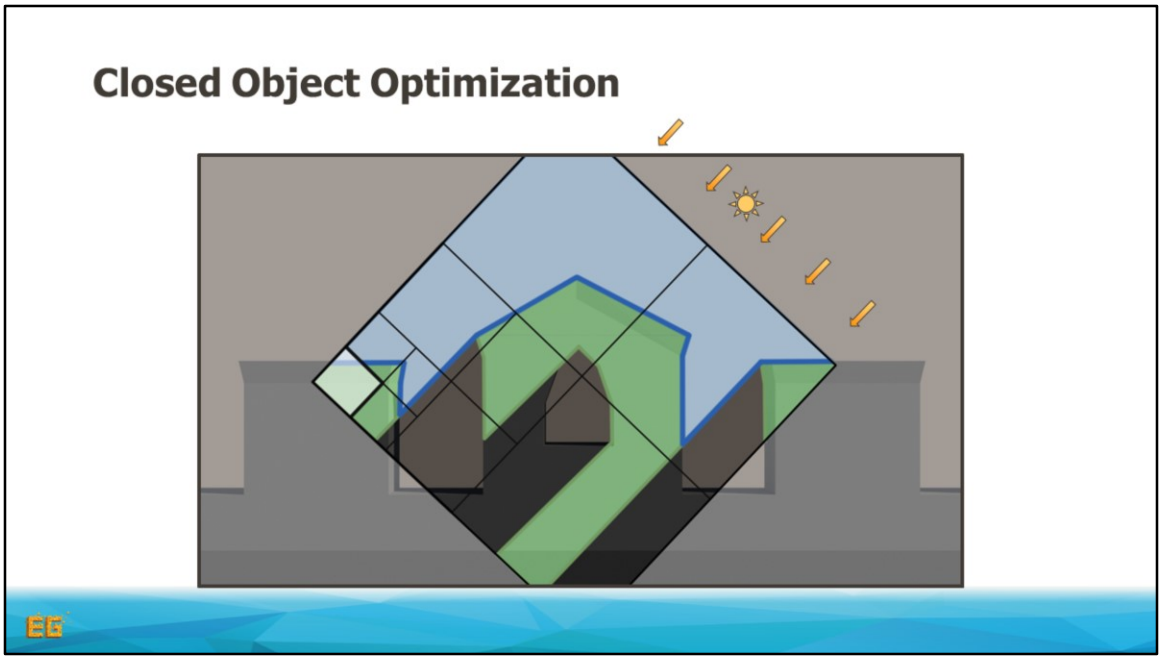
The first step of our implementation is to build a shadow map, with a resolution that matches the final resolution of our dag.



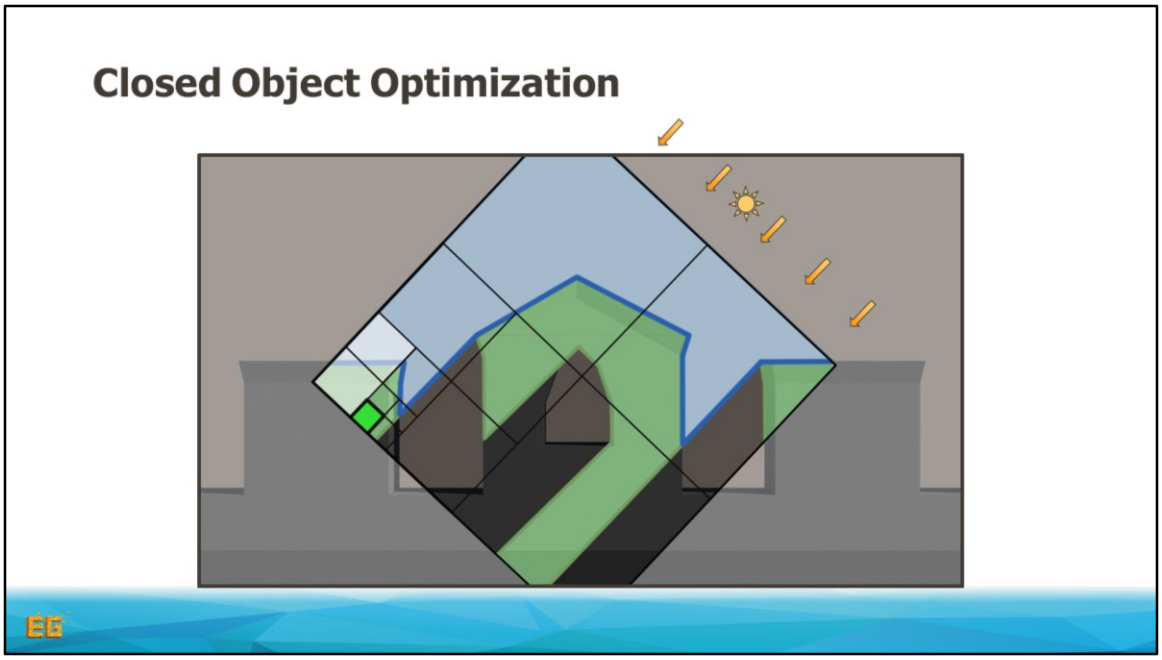
If our geometry is closed, we can then build a *second layer* depth map, which contains the depths at which a ray cast from the light would first leave all closed objects and enter empty space again.



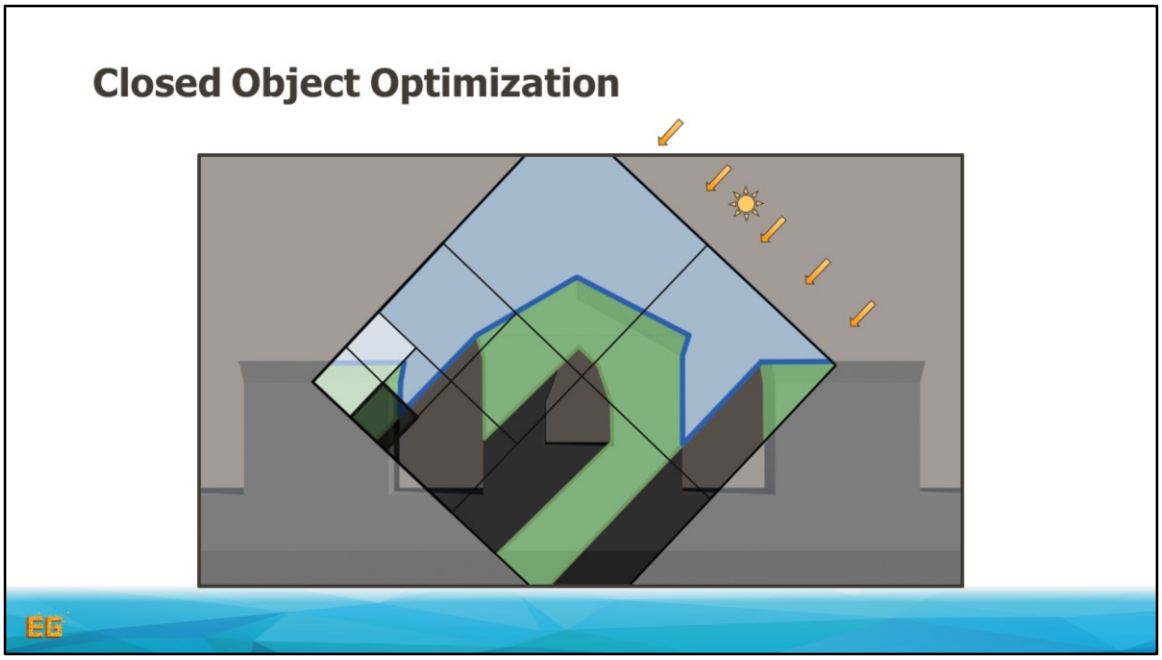
With these two maps, we can now divide the space into *visible*, *shadowed* and "I don't care" regions.



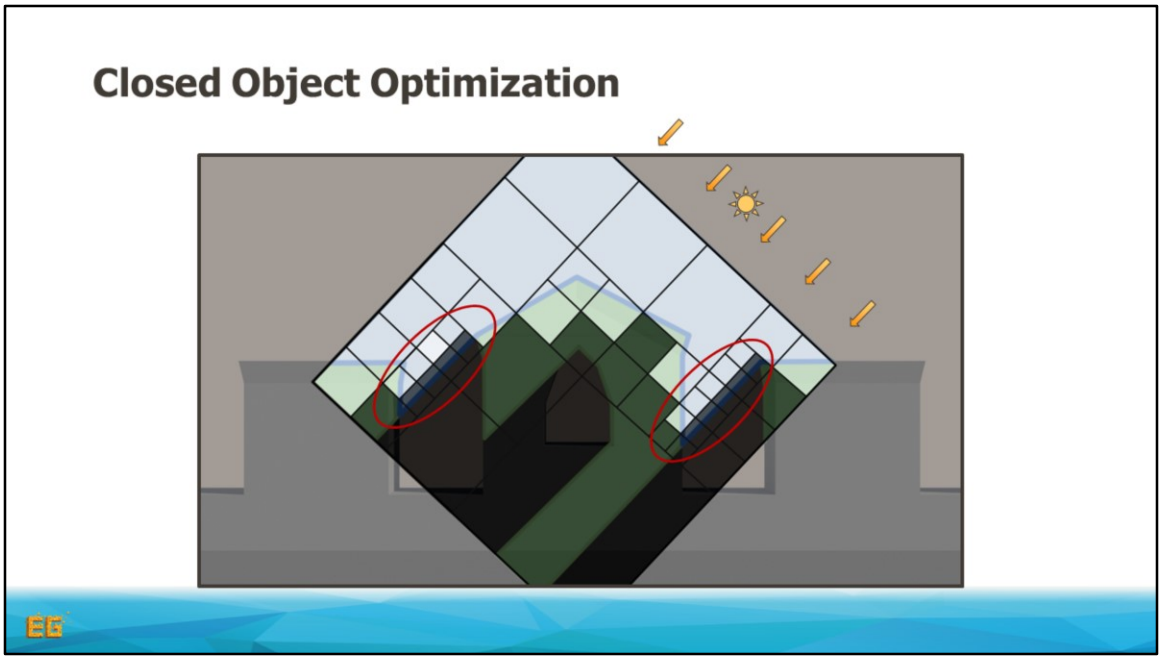
Now we start building the SVO, top down. We start with the left node here and subdivide <click> until we reach a node that intersects only the visible and don’t care regions. These nodes can safely be set to visible.



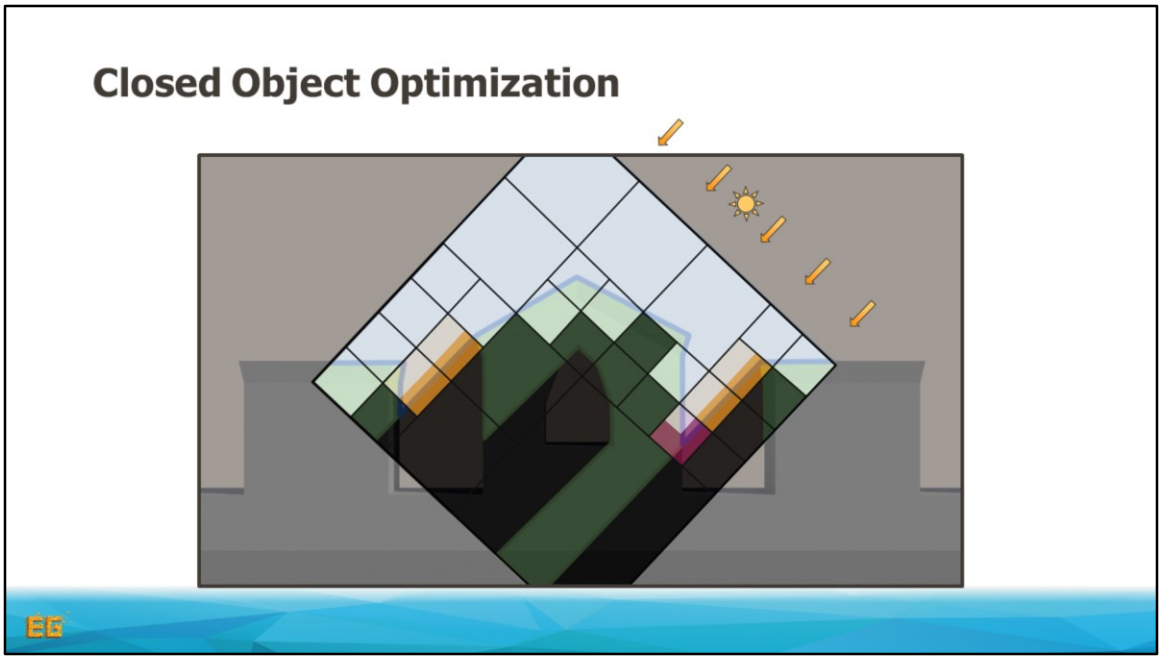
When we reach a node that only intersects the don’t care region, we mark it as such.



When we have processed all subnodes, we find that this node contains only shadowed and don't care subnodes, and so the whole node can safely be set to shadow.

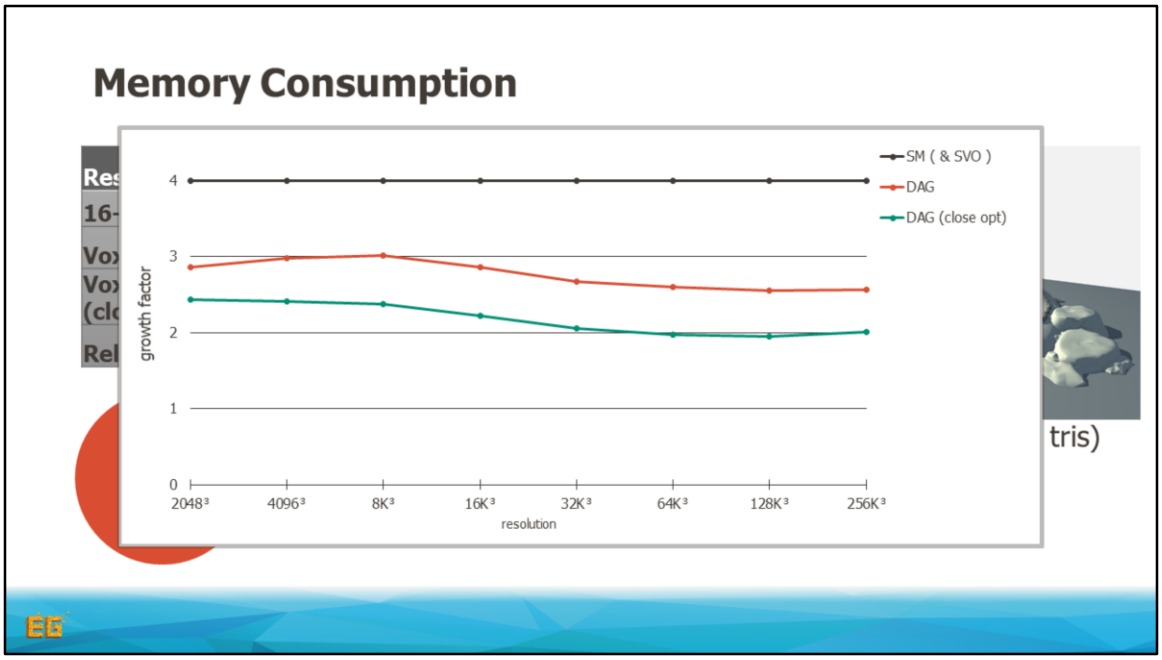


And then we continue like this, for a while, following the same simple rules. The result is much cleaner than what we get from a non-closed mesh, and <click> notice that we now only go down to the leaf level along the silhouettes of the shadow volume.

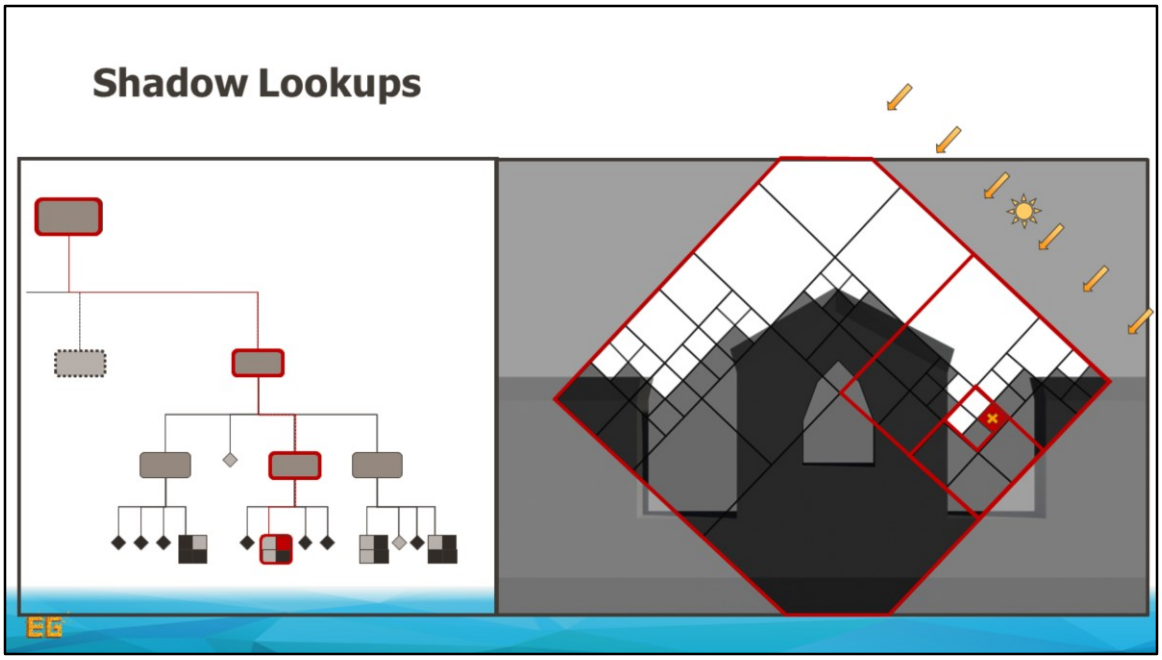


This is good news, because when we then perform the common subtree merging, these regions mostly go away!

For this example, we only have to store *two unique* nodes at the finest level. And in general, the compressed size will be proportional to the 1D silhouettes of the shadow casting surfaces.



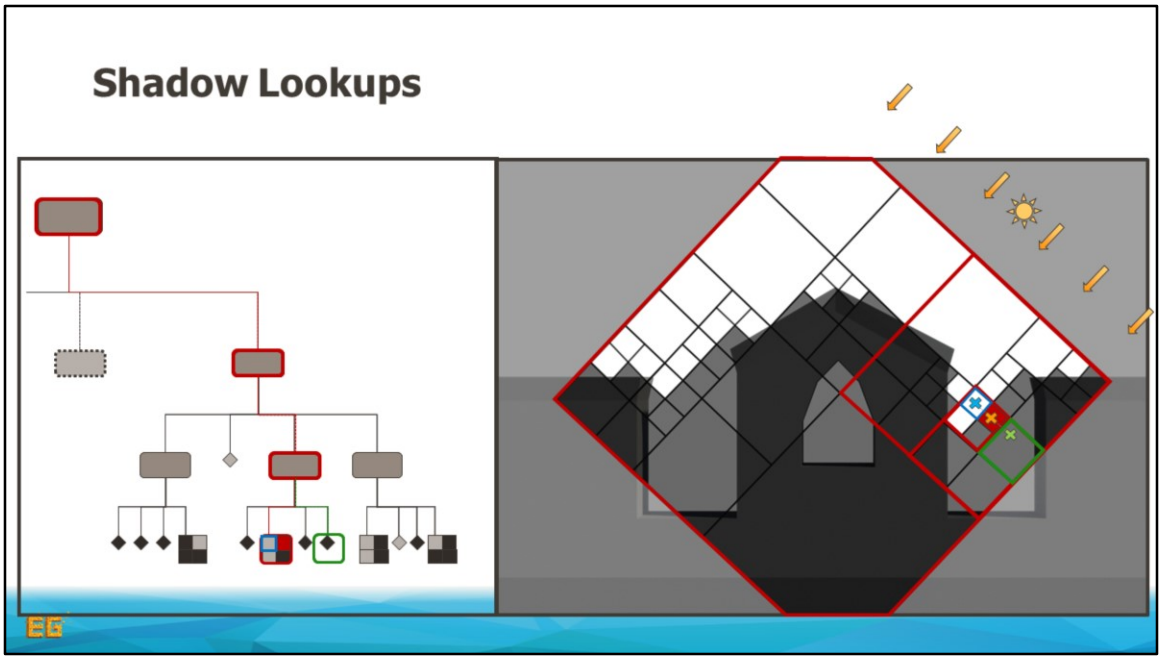
These are the resulting data structure sizes when we make use of the fact that all objects in this scene are closed. We now reach 100x compression already at 16k resolution and at the highest resolution, our data structure weighs in at 100MB, or less than a thousandth of the corresponding shadow-map.
<click>
This is a plot of how much the size grows as we move from one power of two resolution to the next, and as you can see, we quickly reach the state where it is proportional only to the 1D silhouette of the shadow casting surfaces.



The next step, after construction, is to use the structure for shadow lookups. When we do a single lookup for a sample position in the scene,<click>

we start from the root and traverse down to the corresponding node. <click>

In the leafs a small grid is stored as a bitmask, and we get the binary visibility value by masking out the correct bit. The multiple memory transactions of the traversal makes a single shadow lookup rather expensive compared to a simple shadow map lookup.



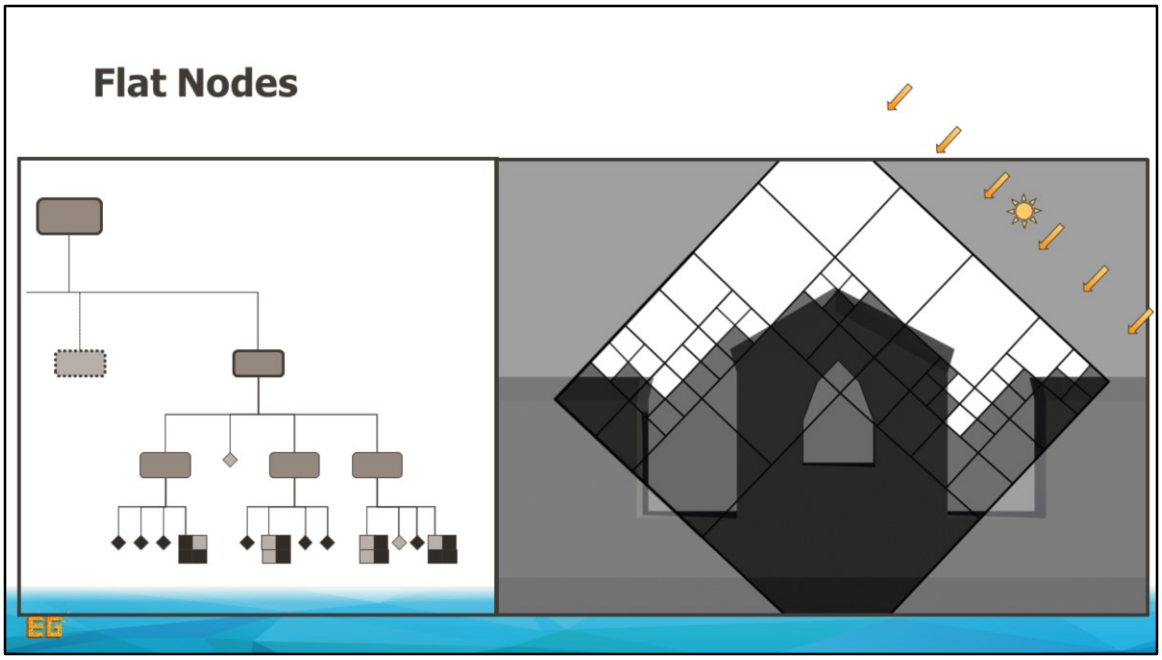
But when we perform many nearby lookups, as we do for percentage closer filtering, we can amortize some of the transactions over many lookups.<click>

The PCF sample positions have the same depth but are distributed in a small region around the original sample point. Here we illustrate this with three taps. When the red sample has been evaluated, <click>

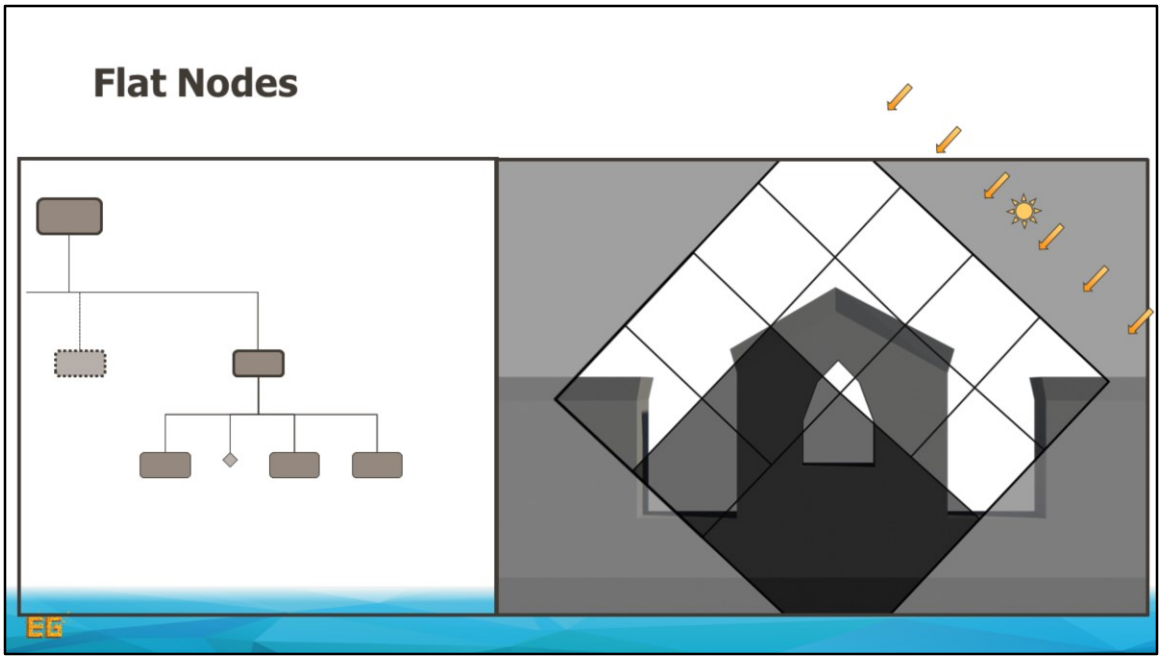
the blue is actually in the same bitmask, and more or less for free. <click>

The green tap is not in the same leaf, but shares a lot of the path and, here, only requires a single new memory transaction.

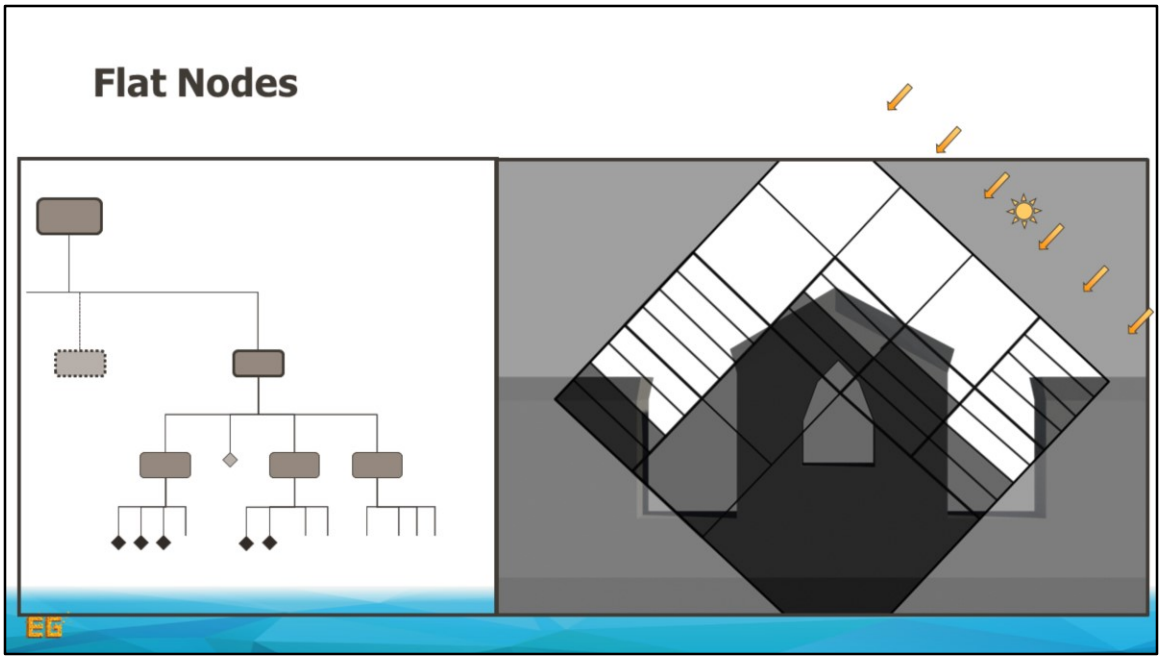
TODO: Flasha dessa också



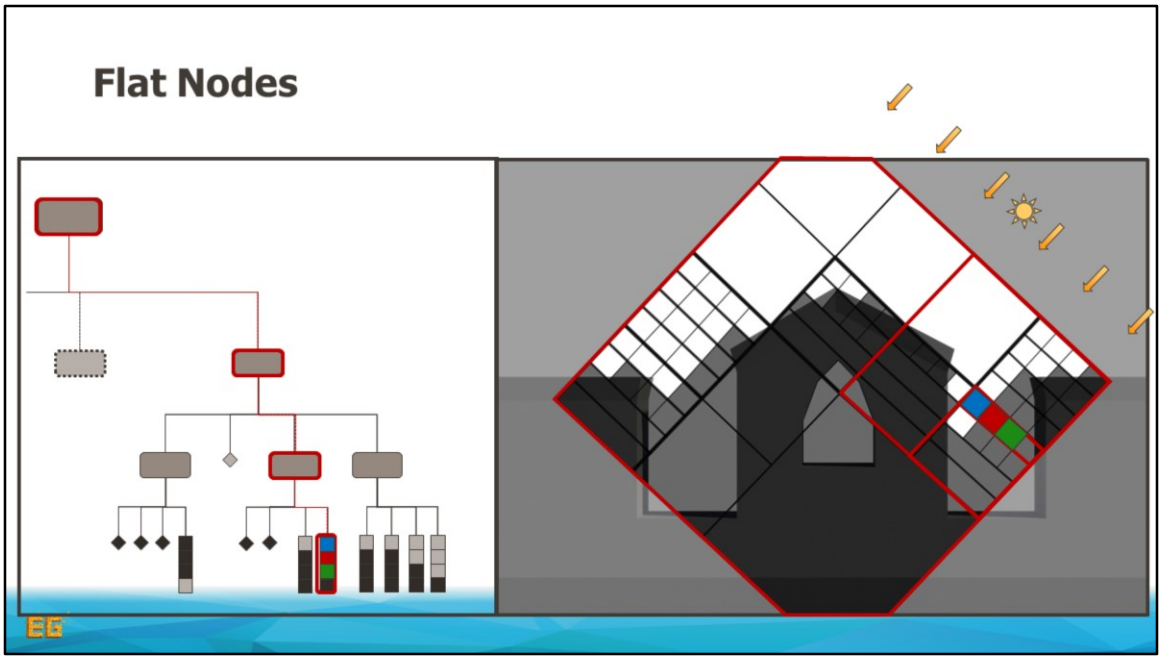
We have adapted the data structure to perform even better for PCF lookups.
We alter the order in which we subdivide the voxel grid in the last two levels, so that leaf nodes only contain voxels at a specific depth.



So, instead of subdividing these final nodes along both axes at once <click>



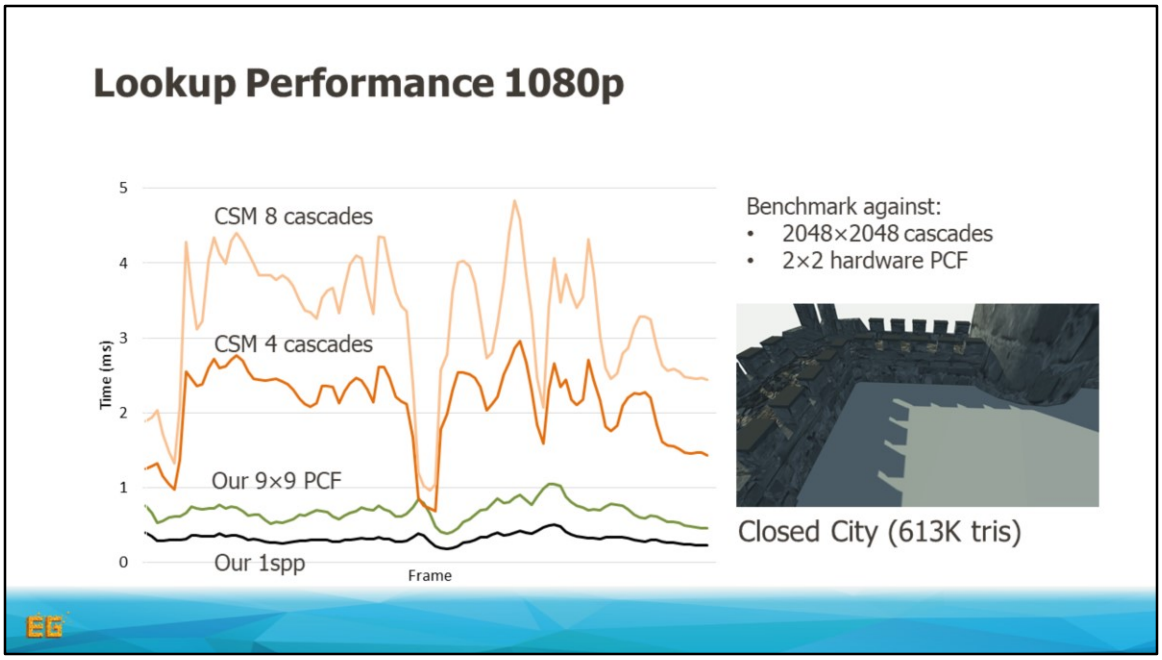
We first subdivide along the depth-axis



And then on the other axis.

PCF lookups are much more efficient with flat leaf nodes, since we can fetch many visibility samples simultaneously.

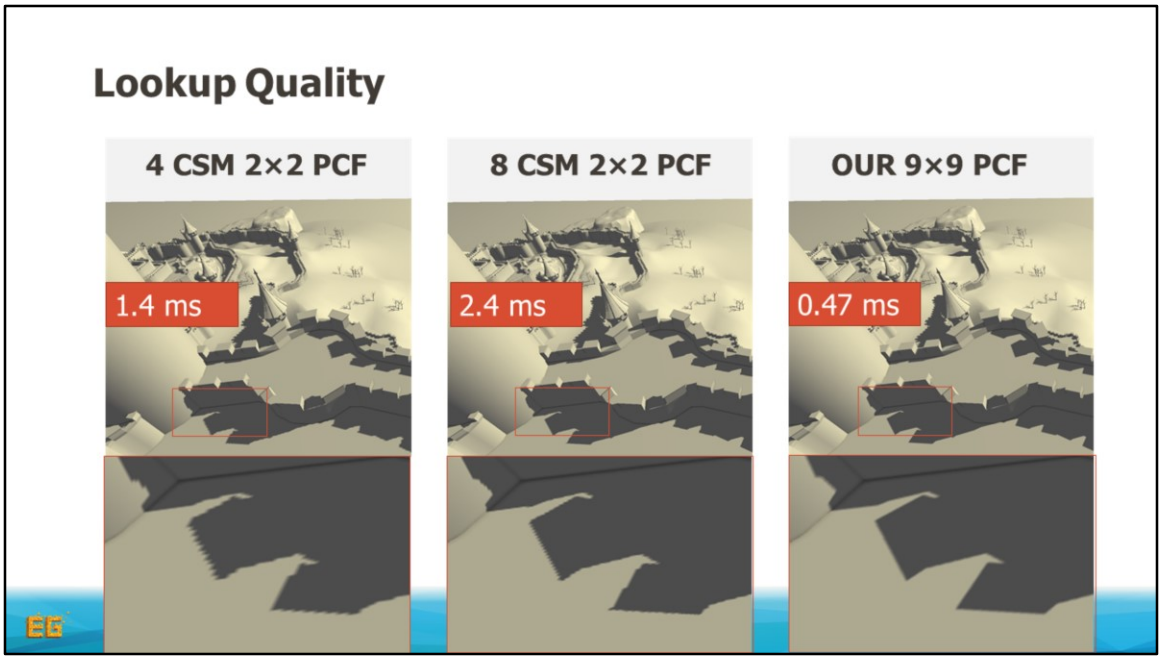
<click>
The same three samples
<click>
All now reside in the same bitmask



We compare the lookup performance against a implementation of Cascaded Shadow Maps tuned for fast performance and wide distance ranges, details are available in the paper.

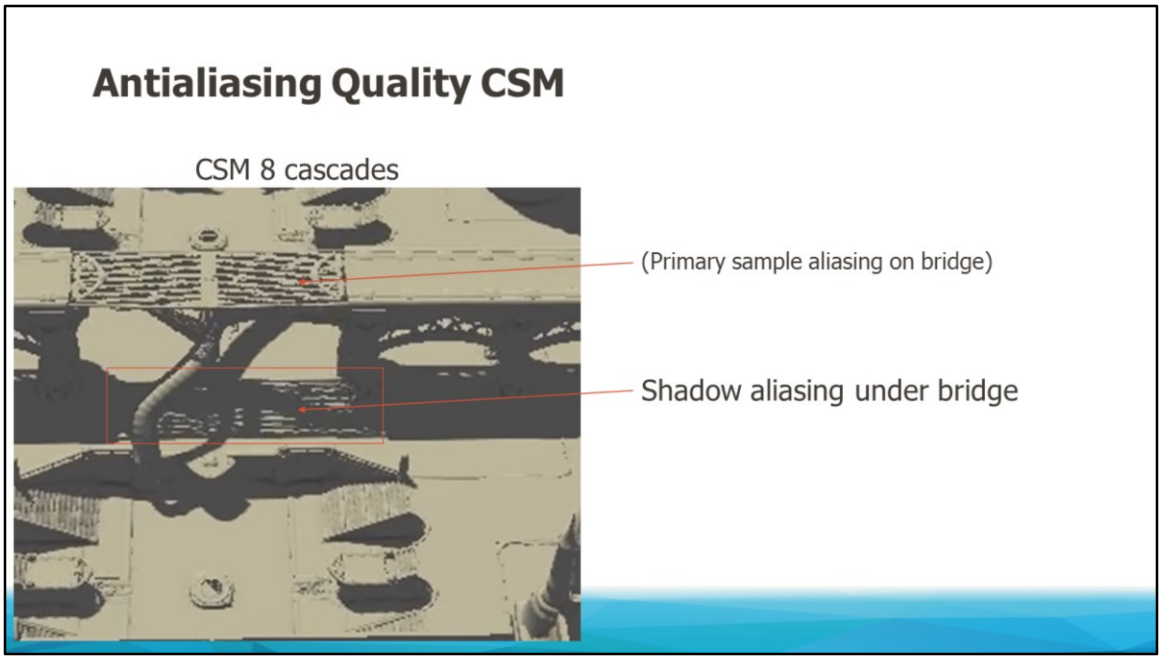
We can see how the performance varies along the camera path. Since the cascaded shadow maps are view dependent, performance includes the time taken to render them. We compare against 4 and 8 cascades, with simple hardware-accelerated 2x2 pcf lookups.

In our method we use a voxelgrid resolution of 256 thousand cubed. A single tap takes up to half a ms, and 9 by 9 taps only take up to 1 ms, due to the amortization of the traversal cost.



Even though we have carefully tuned the cascaded shadow map implementation, artifacts are clearly visible. Here we see the quality of 2x2 pcf of the cascaded shadow map close up. <click>
Even with 8 cascades there is aliasing present. <click>
With our method, we have extremely high resolution and can perform 9x9 PCF filtering while still outperforming even 4 cascades.

Even with large filters like 9x9 we can get aliasing in the far distance. We could further increase the filter size, at a cost, but we have two prefiltering methods to more efficiently coupe with this aliasing.



Here we examine the antialiasing quality by zooming out from this view.

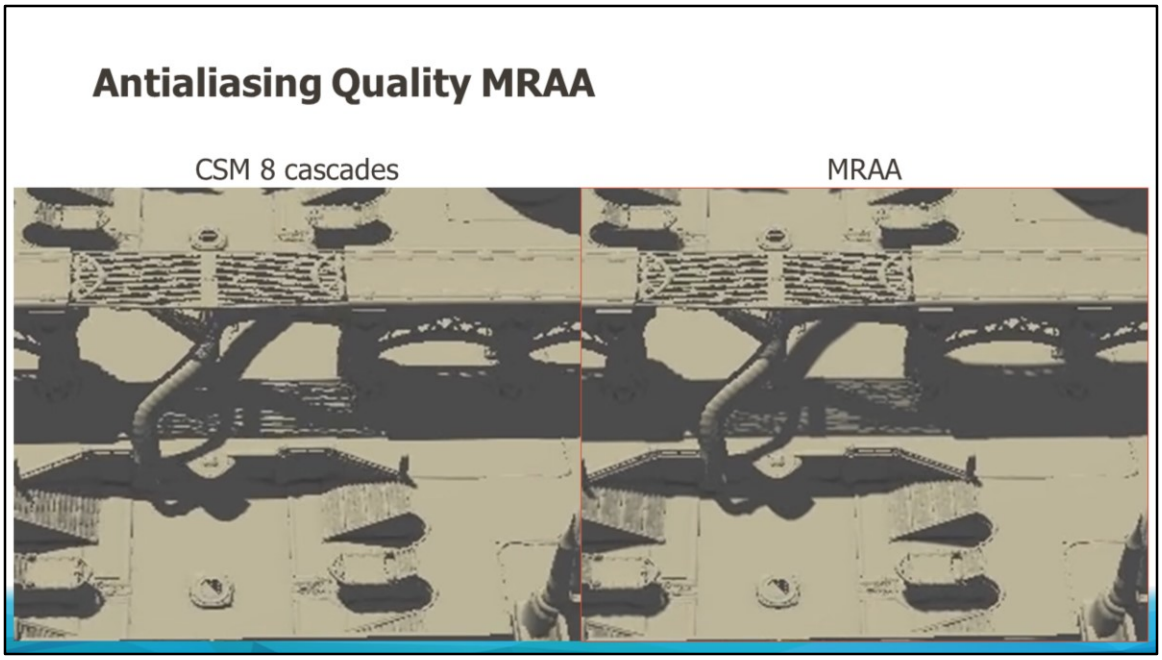
Focus on the aliasing in the shadows below the bridge.

<click>

This is Cascaded Shadow Maps with 8 cascades.

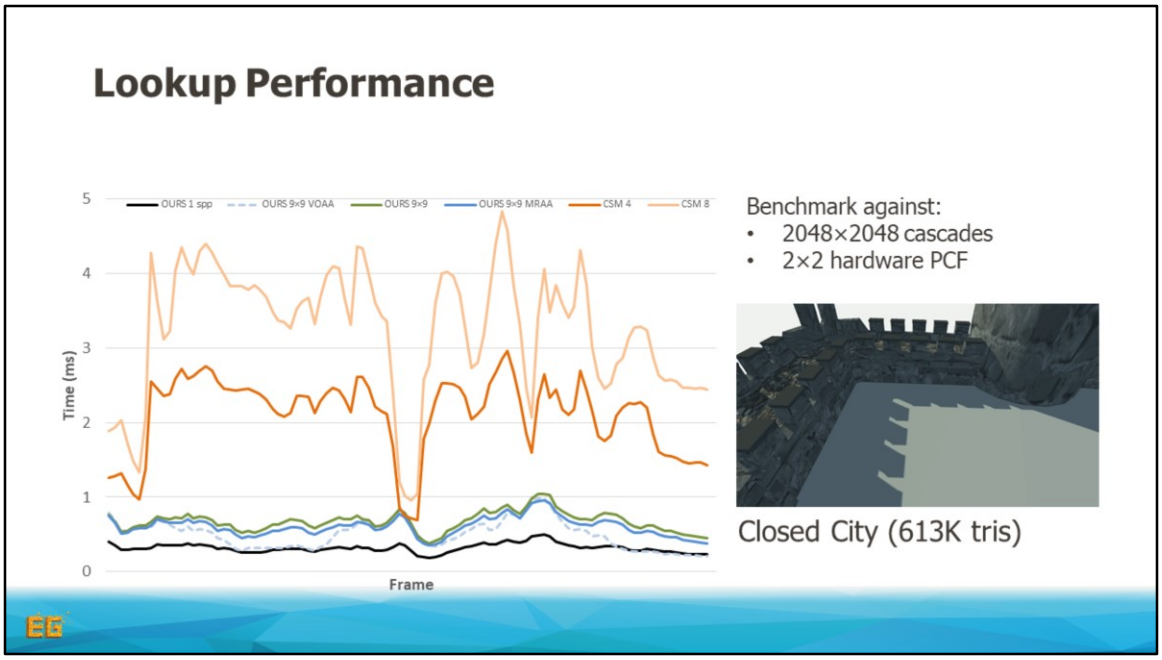
<click>

TODO: Primary

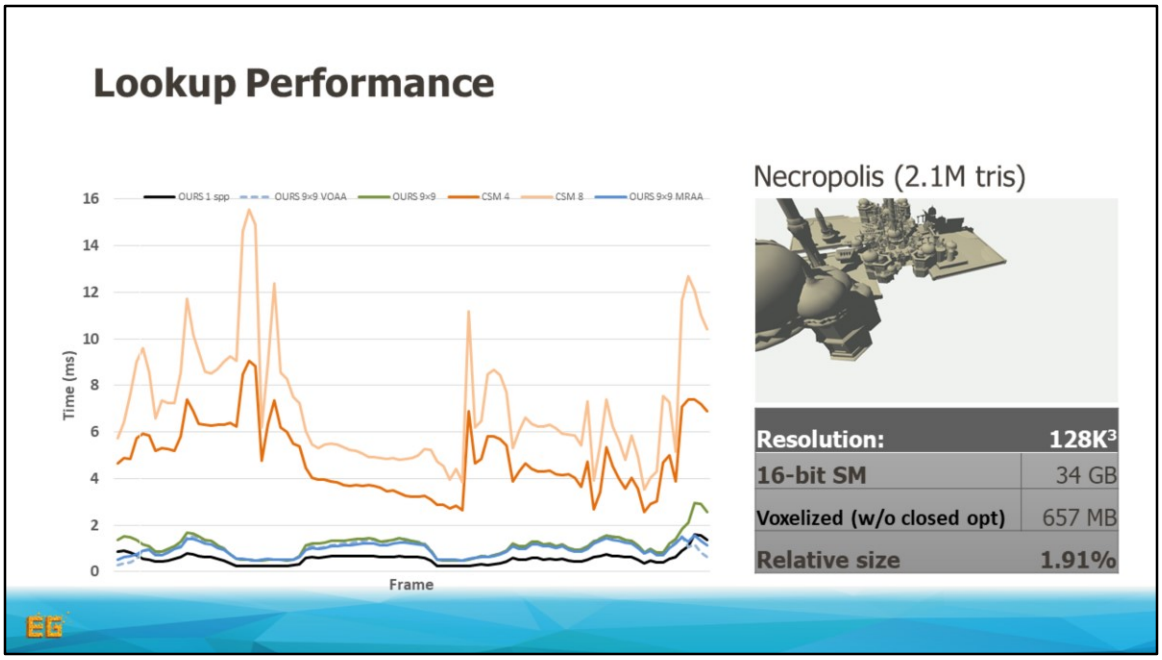


And here, we use Multi Resolution Antialiasing.

As can be seen, the problems with initial sampling aliasing can be more or less removed by precomputing visibility at very high resolutions and prefiltering.



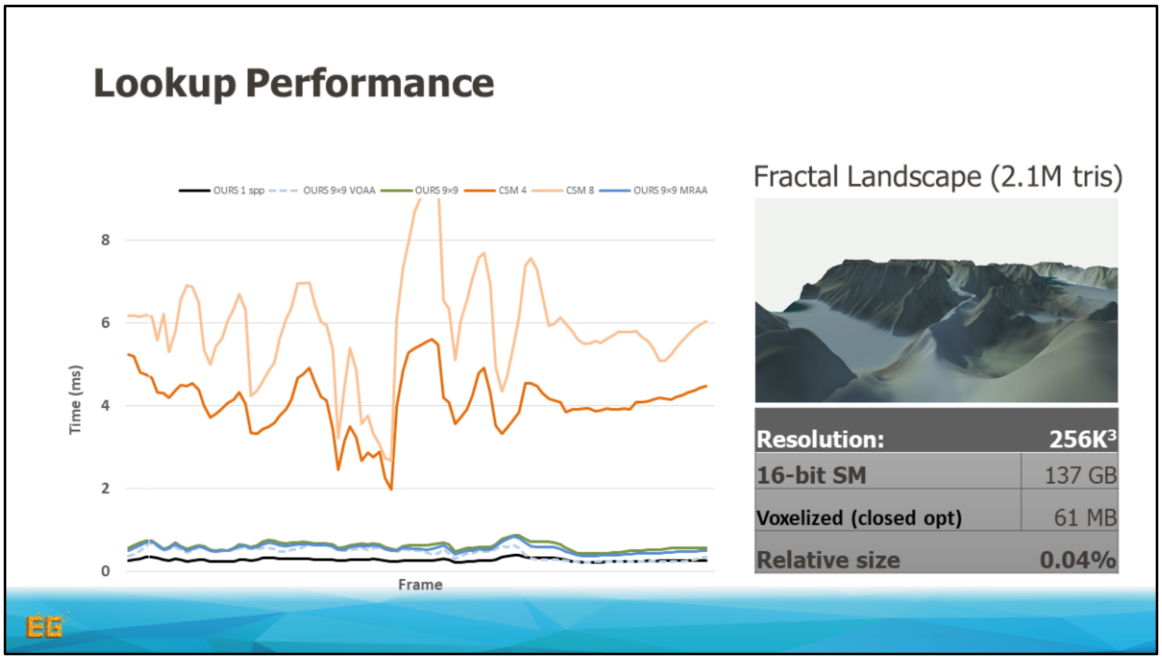
Performance only improves when using these prefiltering methods, as traversal can often be terminated earlier.



Here we show the performance in a proper game scene, the Necropolis level from the Unreal Development Kit. It consist of roughly 2 million triangles.

Even without being able to use the closed object optimization, our data structure compresses to roughly 600 MB at 128k cubed.

The higher triangle density increases the cost of rendering the shadow maps, but as our solution is precomputed, we see performance almost the same as before.



Here we have a fractal landscape of roughly 2 million triangles. We have also inserted a dynamic object that recieves shadows from the landscape.

Since the landscape is closed geometry, our data structure consumes only 61 MB at 256k cubed, a small fraction of what a shadow map would cost.

Summary

Executable and source code at:
http://www.cse.chalmers.se/~d00sint/

Our solution provide high quality precomputed shadows

1) for large and open scenes
2) With high resolution and fast filtering
3) And with very efficient antialiasing for distant shadows

And this within a few hundred megabytes.

Note that, while our method does not help with shadows from dynamic objects, it can be combined with a simple cascaded shadow map implementation, which will be very cheap as only a fraction of the triangles now need to be rendered.

The CUDA traversal code is provided as supplementary material.