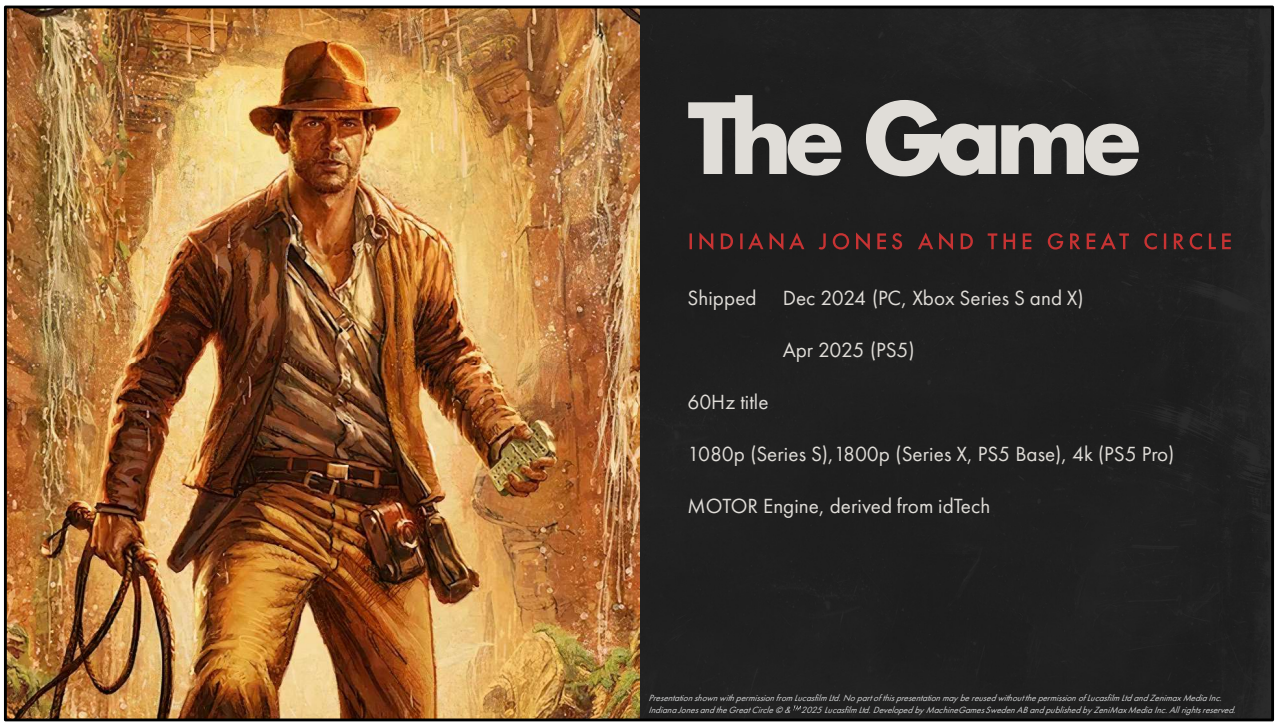
Hi, my name is Joel de Vahl and I work as a Senior Render Programmer at MachineGames.
Welcome to this presentation about shipping Indiana Jones and the Great Circle

MachineGames is a triple A studio located in Uppsala, Sweden, just north of Stockholm. We also have a small satellite office a bit further up north in Sundsvall which I work out of.
The studio has previously mostly been known for the Wolfenstein games starting with 2014s The New Order



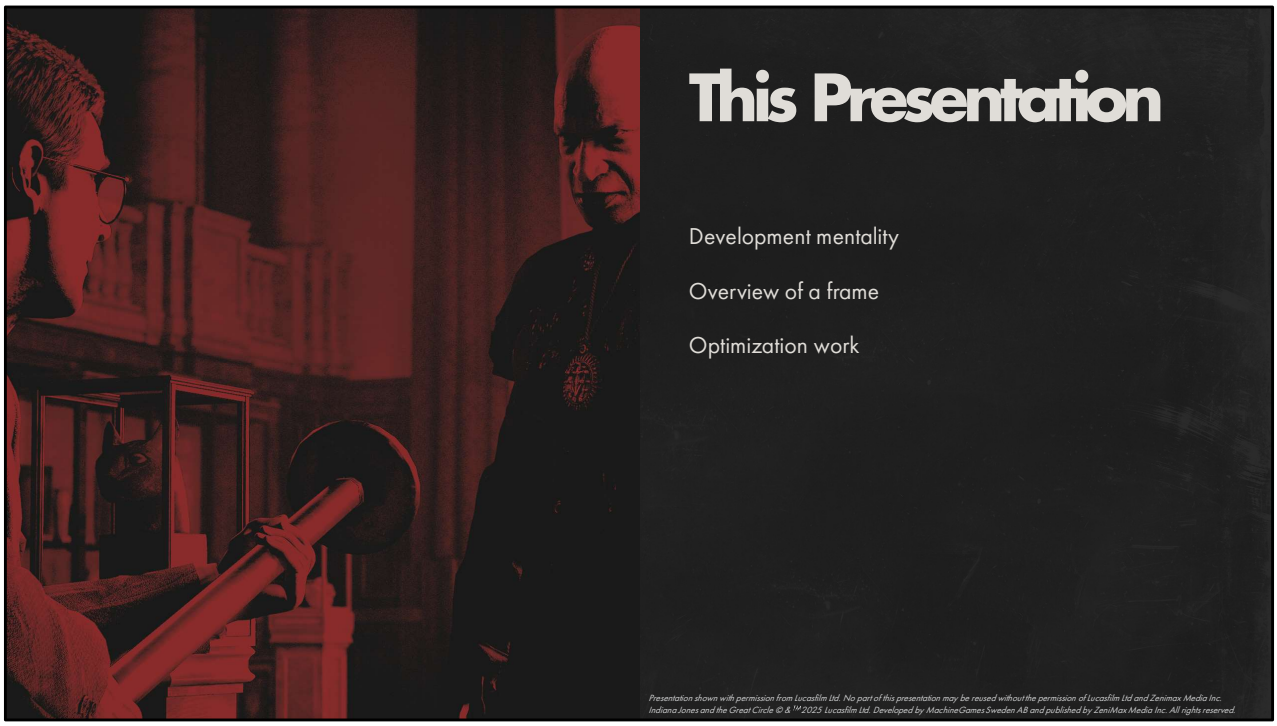
Announced in 2021, Indiana Jones shipped late 2024 for PC and Xbox, during the spring for PS5, and as the title of this presentation states the game is a 60Hz title on all platforms. With a bit of difference in internal rendering resolution.

The engine “MOTOR” (which is the Swedish word for engine) is a fork of idTech 7, and we do quite a lot of technology sharing with id around rendering. The gameplay side on the other hand is quite different but that’s not the scope for this presentation.

Team size combined with production timeline meant that we really had to focus our efforts where it mattered. What in the engine could be kept the way it was, what did we need to replace or improve to ship the game we want to.

A thing I want to stress is that what I will go over in this presentation was a highly collaborative effort by a big team both internal and external to the studio. I will try to answer questions in the Q and A but can probably only give very general responses on areas I personally did not work directly on.

Big thanks to the rendering team for the help collecting all the info to be presented here.



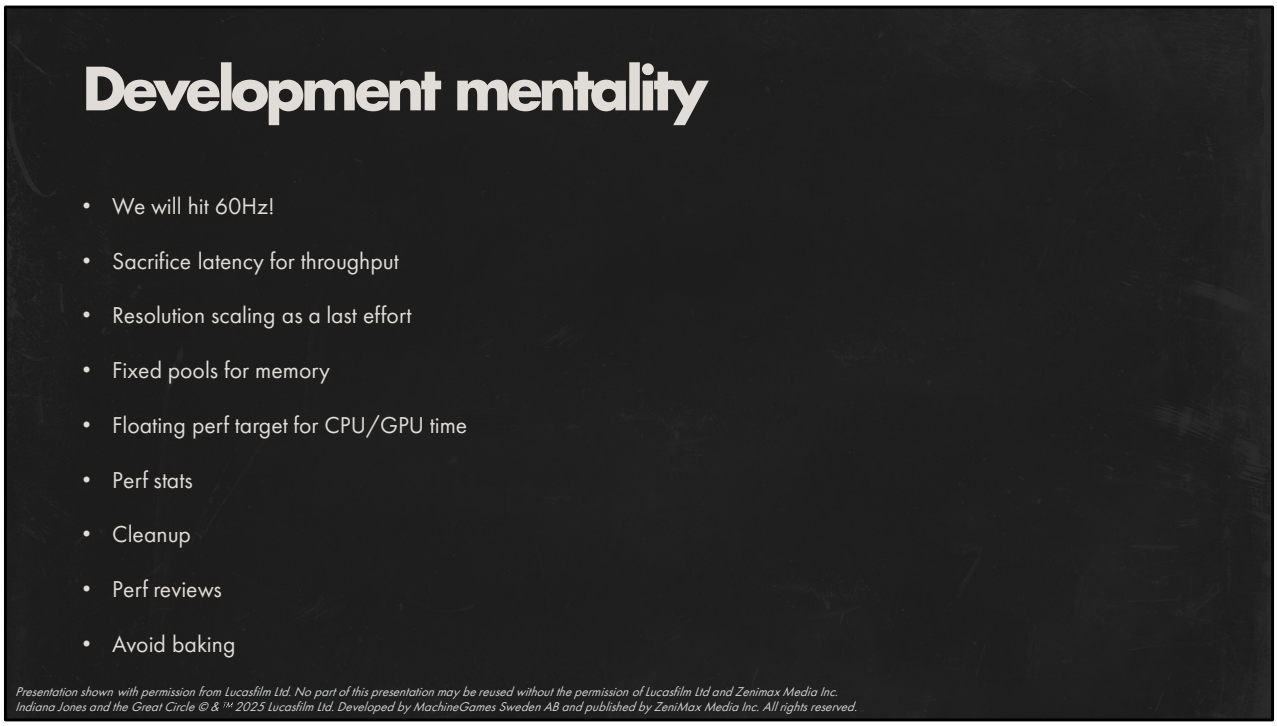
In this presentation we will go over the development mentality the team had from the start so that we would be confident in shipping a 60Hz title.
Then we will go over the anatomy of a frame, followed by case studies of specific optimizations that were done during the final stretch of development to hit the 60Hz target at as high quality as possible.

Also, a small note on what this presentation is not about. I'm not going to talk about the path tracing implementation or our hair rendering during this presentation. For path tracing see the GDC talk by Ivan from NVIDIA and for hair please be patient.
I'm also going to be a bit light on the GI implementation we used, which is the first iteration of what was shipped in Doom the Dark Ages. I suspect more information will follow on that as well.

I thought I'd start things off by showing the launch trailer from last fall so that those of you who have not played the game could get an idea of what it's about.


INDIANA JONES
and the
GREAT CIRCLE
LAUNCH TRAILER
Preseptation shown with permission from Lucasfilm Ltd. No part of this presentation may be reused without the permission of Lucasfilm Ltd and Zenimax Media Inc.
Indiana Jones and the Great Circle © & ™ 2025 Lucasfilm Ltd. Developed by MachineGames Sweden AB and published by ZeniMax Media Inc. All rights reserved.

So, now that you have seen what the game ended up looking like, on with the development mentality



First of all, the overarching principle is that we will hit 60! This is not something to compromise on and even if things did look bleak at times this was a fixed requirement for us.

To do this we sacrifice latency for throughput in a lot of places. One frame delay is not as much at 60Hz as it is in 30Hz. We also try to amortize work whenever possible. And while this adds latency, it also spreads out load.

We run with resolution scaling on, which allows us to iron out some drops and uneven frame rates to keep the game running at a good pace.

When it comes to budgets, most data in the game is placed in fixed sized pools. These are allocated up front with a worst case bound. They cannot grow at runtime.
For development builds these are bigger. LODS and MIPS are streamed in on demand.

For performance we have a few soft targets on how long passes or systems are allowed to take, but overall perf budgets can vary wildly between different kinds of scenes. In the end phase we focused on optimizing hotspots we saw were under 60Hz or used too much memory.

To track that we used a series of automated tests that was run nightly that tracks performance and resource usage. With this we get a set of perf metrics from several locations in each game map as well as all cut scenes and can track progress on these over time. This was very helpful during the end phase to pinpoint where our efforts were needed the most.
Data here include pass timings within a frame, overall GPU/CPU time, load times, number of models as well as per shader stats such as VGPR usage.

We also work actively with removing unused content and systems from the game. Fridays are designated “cleanup day” where people are encouraged to take some time to remove unused systems or data.

Additionally on the rendering side we have worked with regular perf reviews, preferably one per week, where we focus on a scene digging into the performance to see if it is reasonable or requires additional work, either from art or code side.
Here we try to work together with art, ensuring that what gets built can reach 60, but also try to be on the lookout for places where it turns out code is the limiting factor and needs extra work to be performant.

I also want to add that for rendering the only thing we bake on a per map basis are reflection probes. Everything else is runtime systems to speed up iteration time for artists.

To recap from the trailer, here is a selection of our heaviest scenes.

Treehouse in Sukhothai. High vantage point, lots of trees and very little in terms of effective closeup occluders

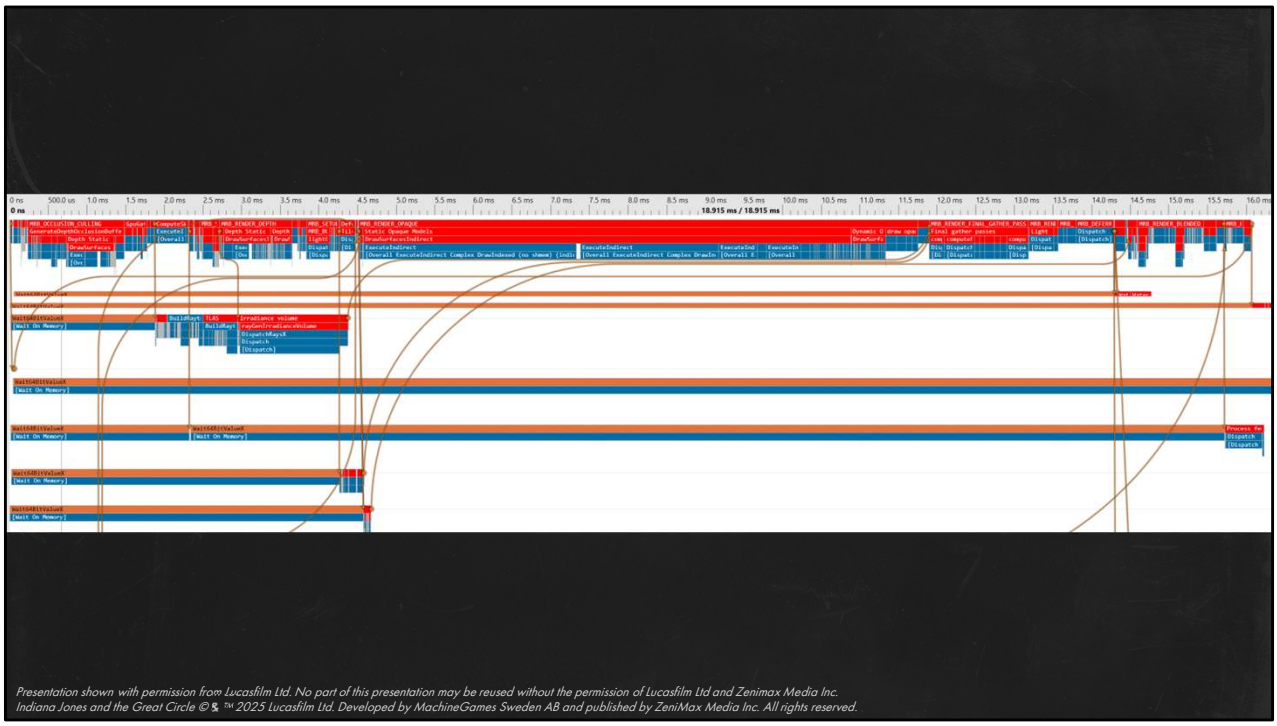
Nepal cave. A lot of transparent things such as ice and particles, as well as dither blended snow and heavy rock shaders

Gizeh lookout, lots and lots of instances and very few sectors so almost everything out in the open is there all the time.
Sectors are our way to load/unload parts of the map, but that did not work out here as the player can see everything

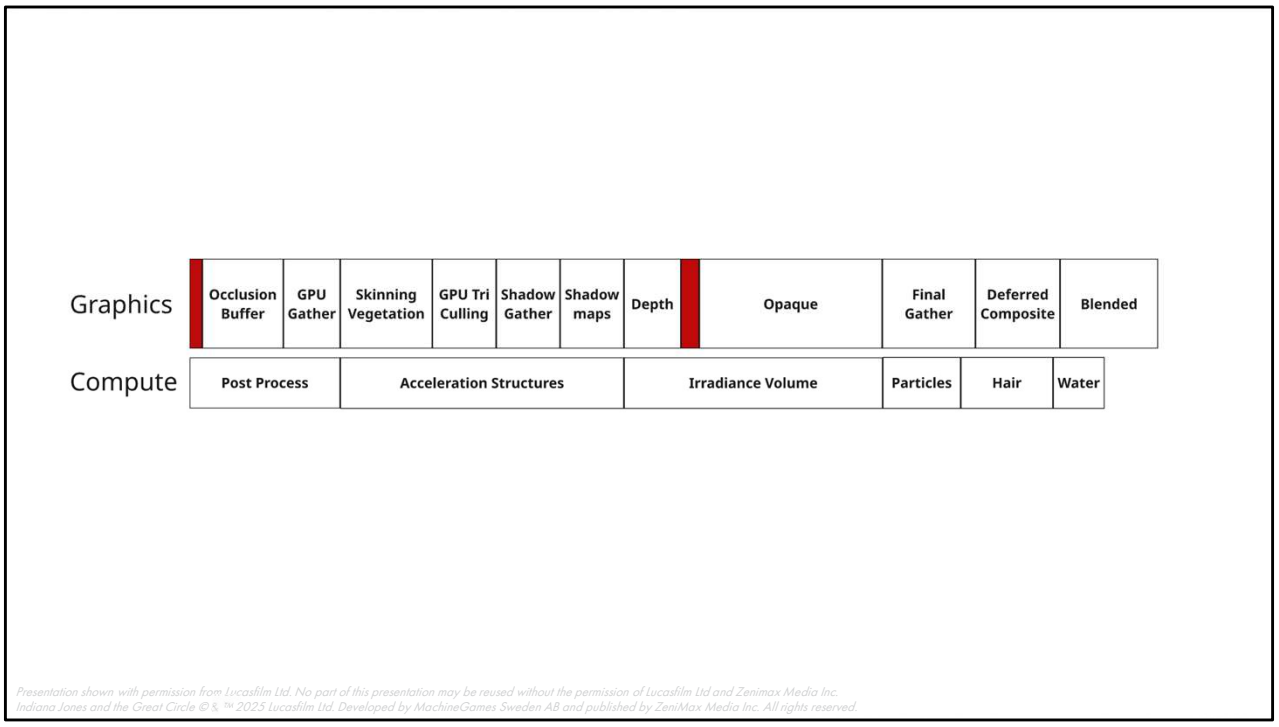
And finally, cutscenes, which often are heavy overall due to everything being dynamic and set up with lots of cut scene only cinematic lights.

For the CPU side we use the common pattern of overlapping one game simulation frame with one render frame, synchronized with a “commit” that copies game state data into the renderer.

Let's take a look at Gizeh to see how a frame is rendered on the GPU.



So here is a timeline for the same Gizeh image I showed before. It’s heavily opaque bound as you can see by the big block in the middle of the frame.

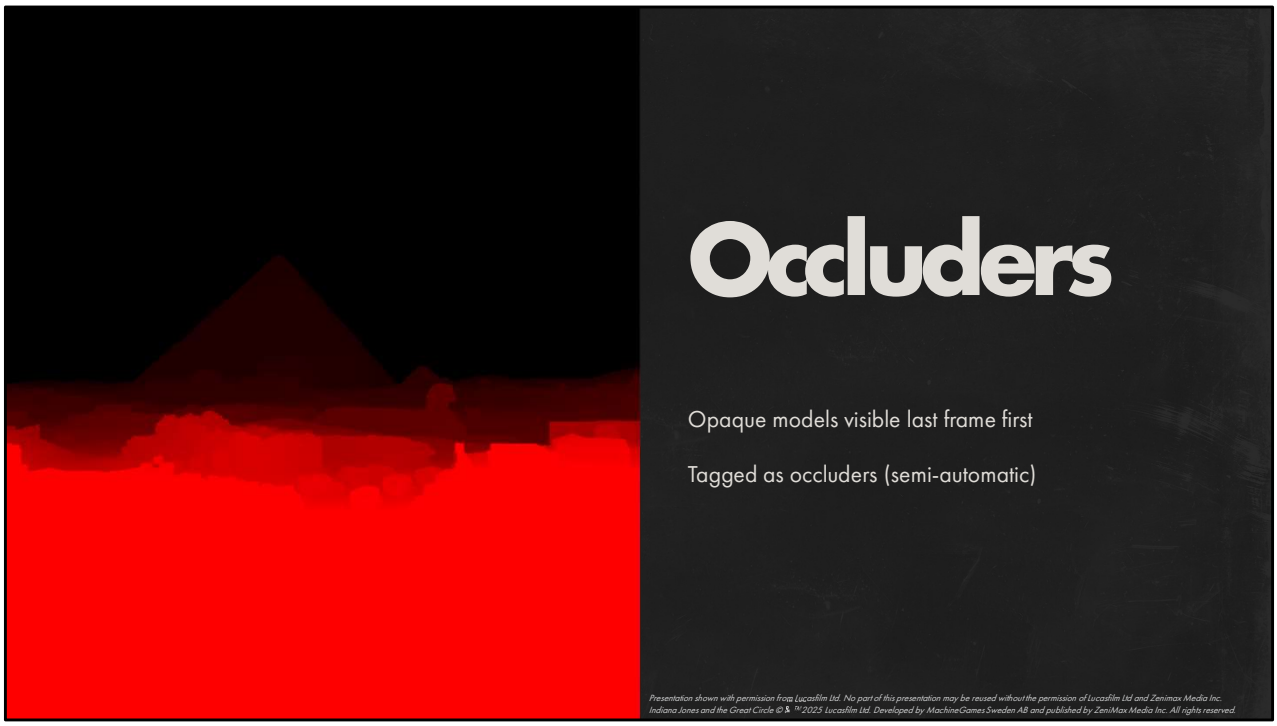


A simplified version would look something like this. Sizes of the passes not to scale, but a general idea of the timeline at least.

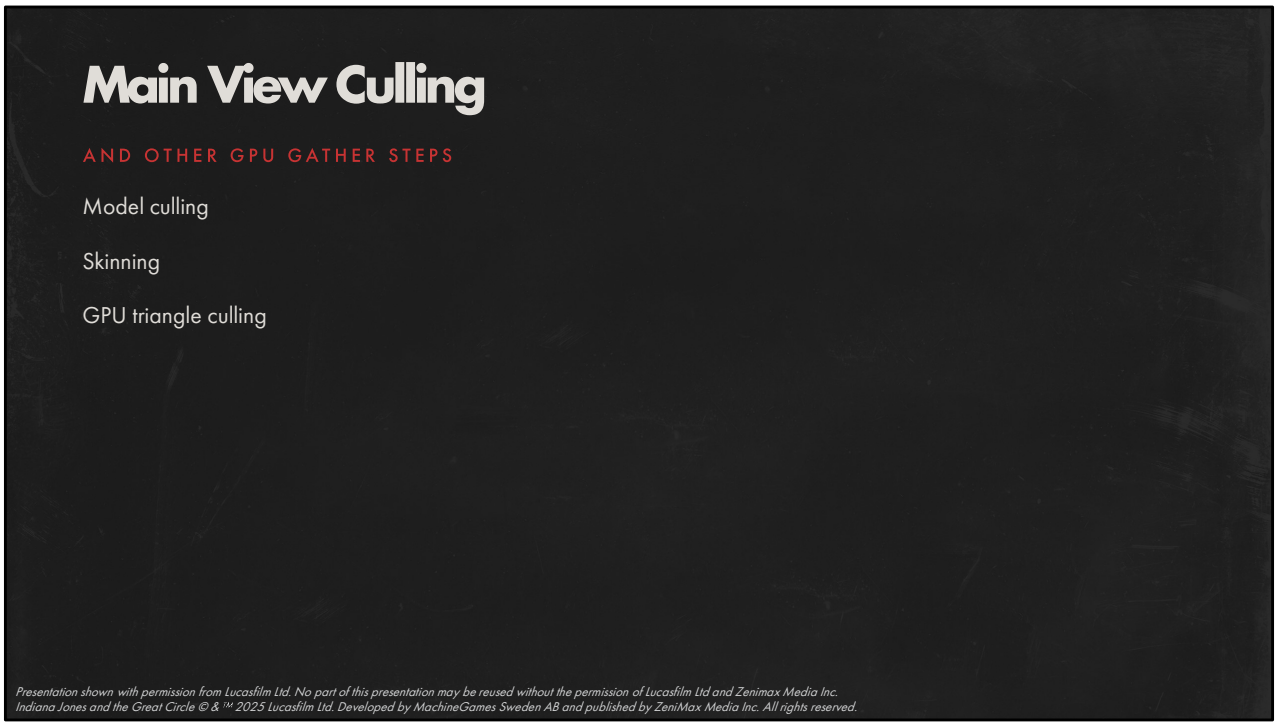
The red boxes here are where we do a lot of setup passes, preparing other work. Adding all that detail here would make the timeline a bit too cluttered.

The async work might also not look exactly as it did on the previous slides. The boxes positions are scaled to fit into the time frame in which they have the ability to run. Often scenes to not require all of them to be that heavy.

In the first red box there to the left we start off the frame by doing a bit of setup: terrain, wind and ecotopes. The latter being our GPU based vegetation placement systems which runs periodically and updates visible vegetation instances



We then generate an occlusion buffer by drawing occluders that was visible last frame.
We don’t treat everything as occluders. A system selects what seems to fit best but artists and designers can go in and force this on or off for specific models



After the occlusion buffer has been created and then downsampled we enter the culling phase for the main camera view. For the majority of the models and surfaces this is a GPU only path. Only a select few things take a CPU path.

All models are tested, giving us a list of “surfaces” to test. This can be seen as submeshes of a model with a specific material.
Surfaces are then tested using their tighter bounding box and a draw call is set up.
At this point we know what to draw and can start compute skinning of skinned models.

Lots of draws also goes through the GPU triangle culling setup, where we build a new index list after testing each triangle against the occlusion buffer as well as doing frustum, back face and pixel center culling. This can reduce the amount of geometry to process in the vertex program quite a bit.

Shadows are then rendered into an atlas using a combination of CPU and GPU generated draws. The GPU generated draws here are very similar to the main view culling described earlier.

A lot of times we can skip rendering anything that has not moved by simply keeping a copy of the static part of the shadow map in a separate slot and rendering dynamics on top of a copy of that. Trading rendering time for memory.

At the same time as shadow drawing runs, we build dynamic bottom level acceleration structures and updated top level acceleration structures async.

After shadows has been drawn we render the full depth buffer using the output from the culling step. This is rendered on to the existing occlusion buffer from the previous steps.
Here motion vectors are also written out for objects that needs it.
Full depth is needed for some setup that happens next

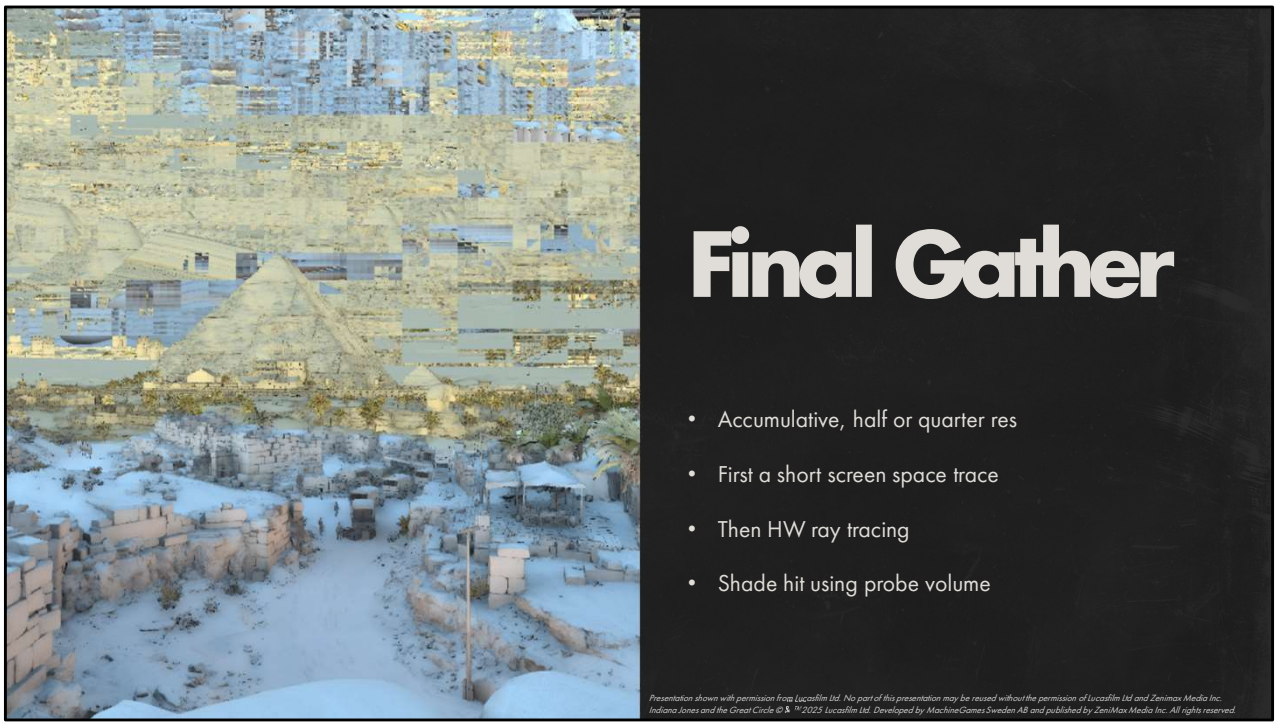
Also, at the same time we do probe tracing for GI async, as we now have the shadow atlas prepared.

Once we have a full depth buffer we can start setting up things that rely on it.

Visible geometry decals and water setup can be calculated.
We calculate light tiles for the forward+ setup.
Also particles and hair is setup.

Its now time for the heaviest pass for this scene at least.

Opaque renders out full forward+ lighting as well as storing out a mini g-buffer to be used for some subsequent passes.
Deferred might have been an option to look towards here but we have a few different BRDFs, as well as some shaders doing light calculations in custom ways.
Sticking with Forward+ which was what the tech was already running on was the safe option.

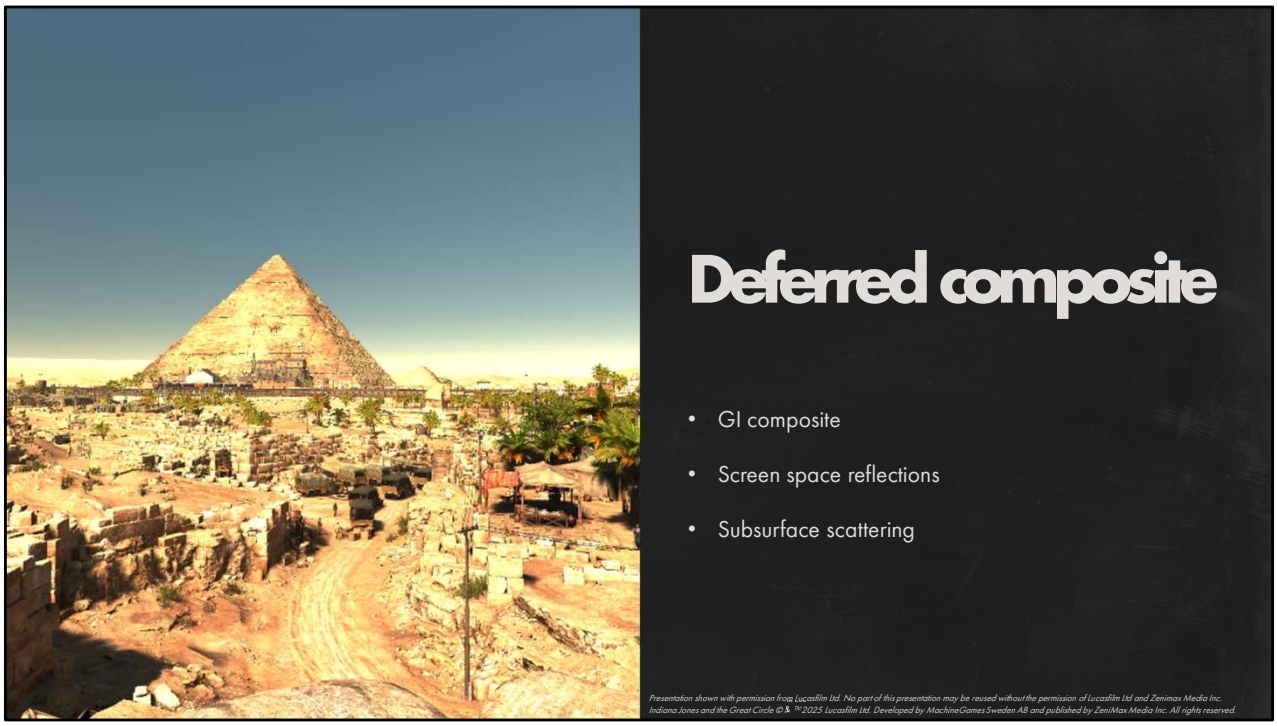


Once a gbuffer is set up we can use the probes we traced async before to do a GI final gather pass, storing out indirect diffuse light contributions.

The sky part of the image here is garbage as that is masked out by another buffer.

We start from a world position and trace a short screen space ray. If that does not hit something we shoot a HW ray.

Hit is diffuse only shaded using probe and accumulated into a spherical harmonic stored per pixel. This is then denoised



Even though we render most things forward we still have a few things that needs a deferred composite.

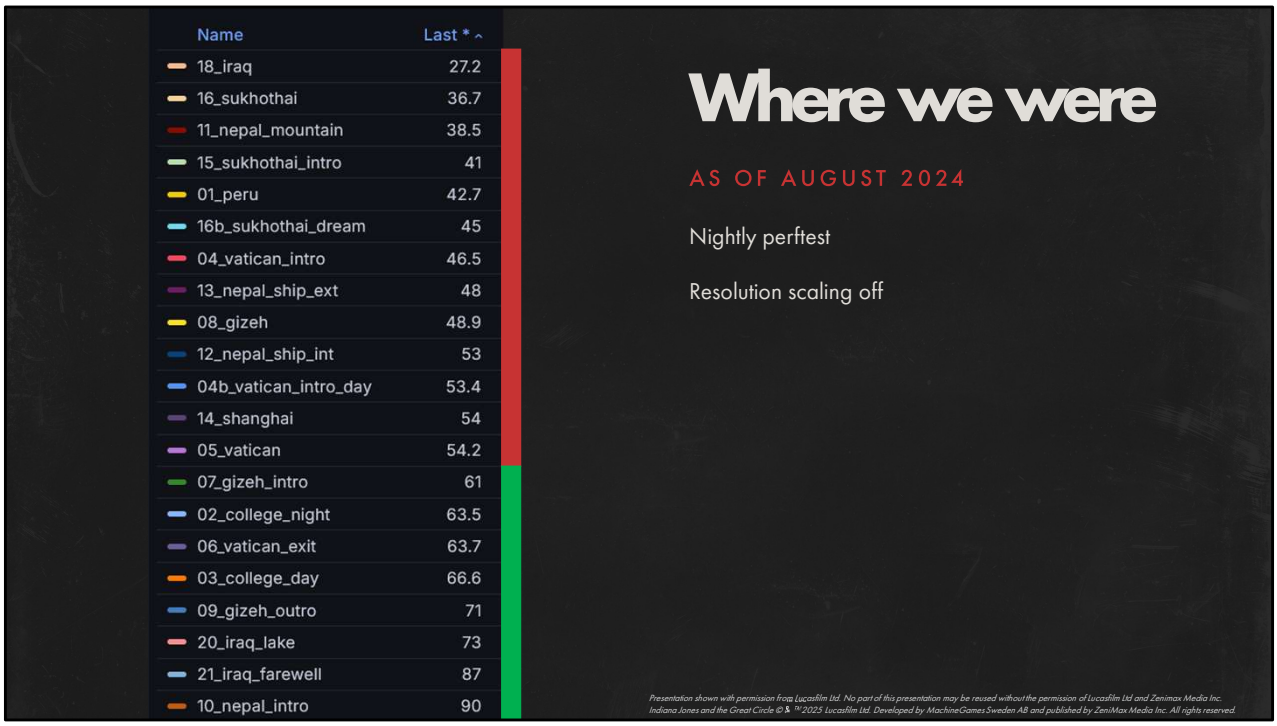
Here we apply the denoised GI from the previous pass, as well as screen space reflections and subsurface scattering

After everything opaque as been drawn we blend in additional transparent surfaces such as water, ice, glass, particles and other transparencies together with distortion. There were very little blended in the capture I used for the other slides, so here is an alternative image showing some particles.

And finally, the image is post processed.

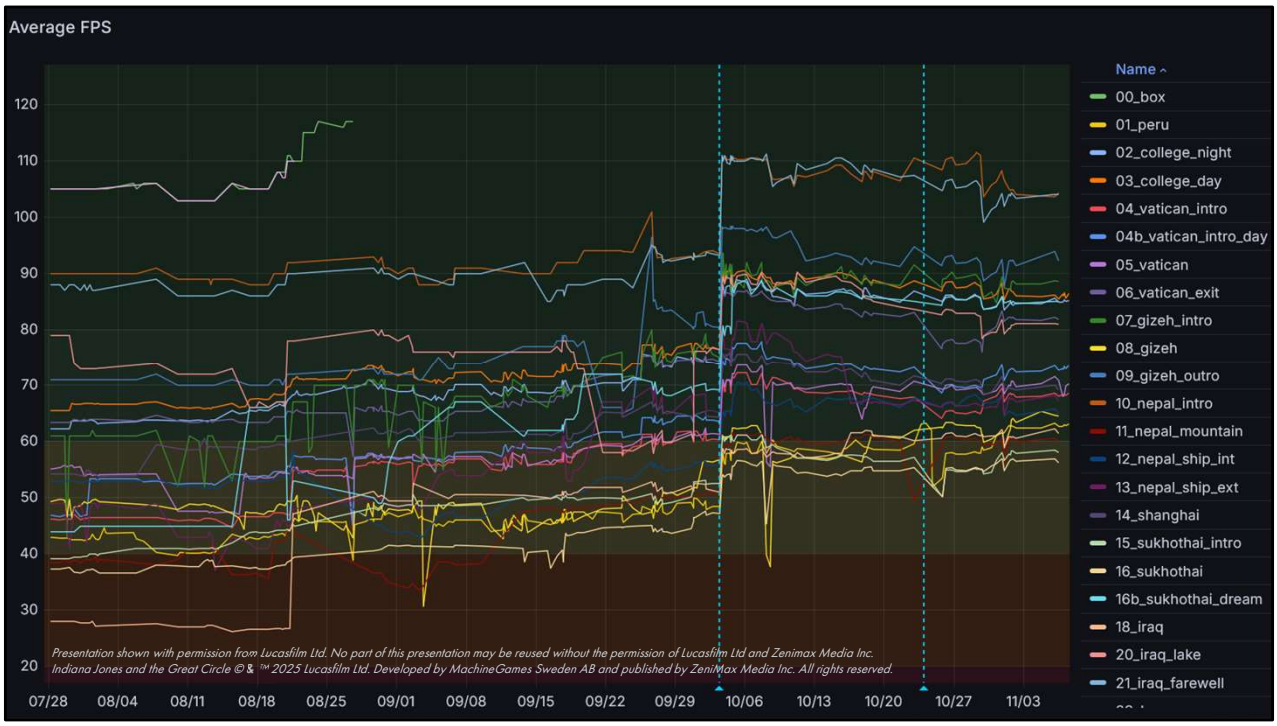
This pass happens async and overlaps into the next frame on the GPU, which means that we often ends up with the actual present happening somewhere before opaque pass starts.

So now that you have seen how a frame is built up and what kind common scene setups we have, let’s take a look at the final stretch of the development.



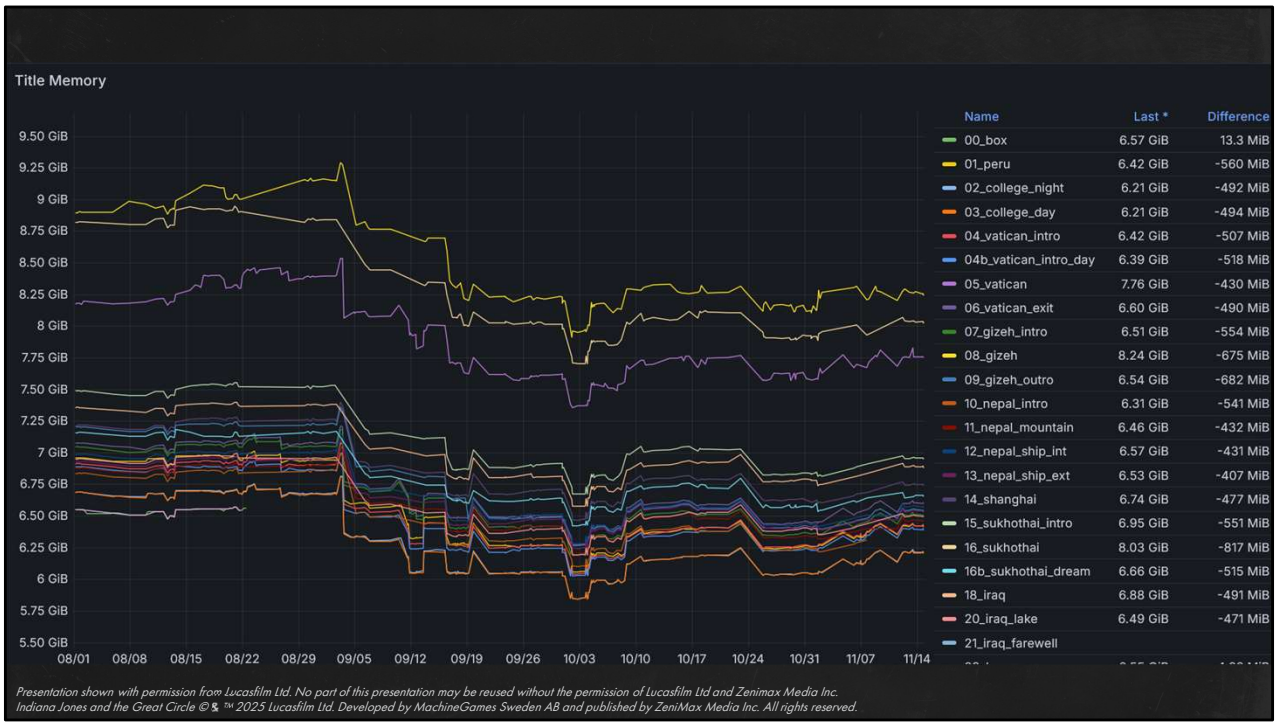
Here is the result of the nightly test on Xbox Series S as of August 2024, roughly four months before the game would be in players hands.
These tests were made with resolution scaling off.

As you can see the majority of the levels struggle. Some by just a tiny margin but some with quite a bit. I've marked the levels under 60 with red, and there you can see all the main levels of the game.
The green marked levels that average over 60 are all the smaller intro/exit levels.

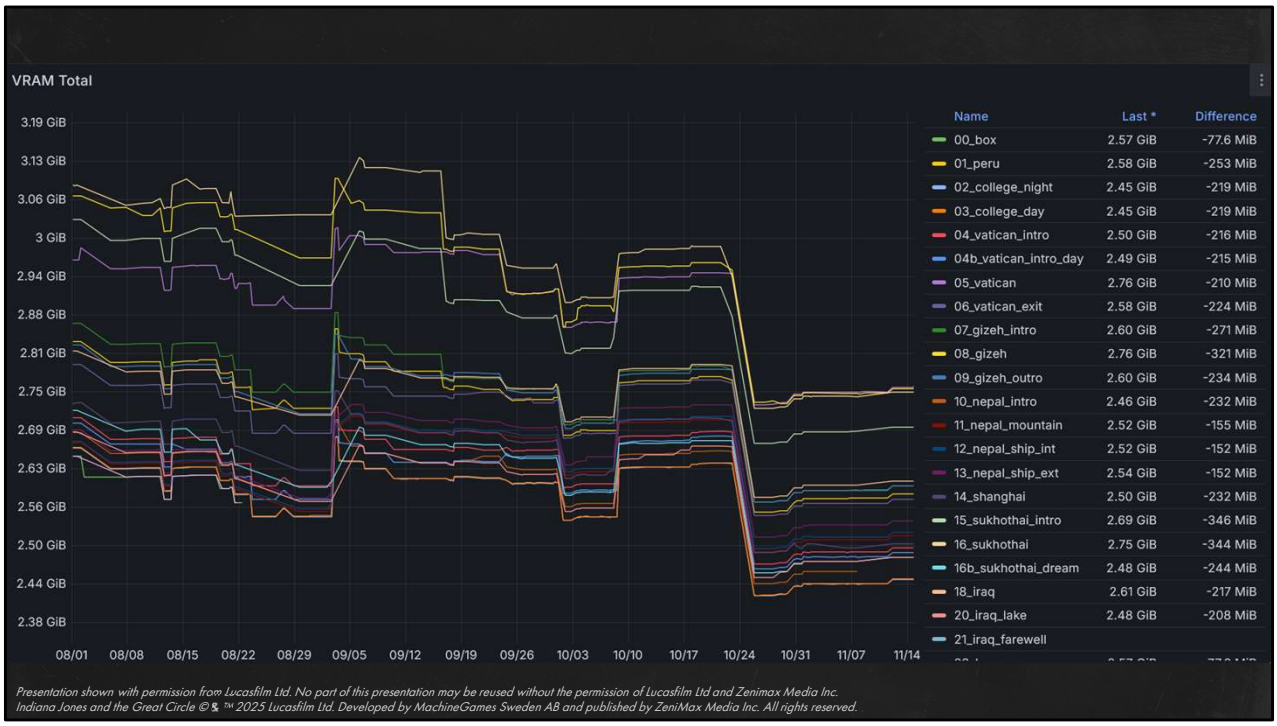


And here is a perf timeline showing the average FPS and how it changed over time towards release.

Worth noting here is that this is not just a struggle to get things as fast as possible. We want things to also look as good as possible, so graphs go up and down quite a bit as content is getting added, tweaked, removed, readded and code optimizations come in.



Total memory used

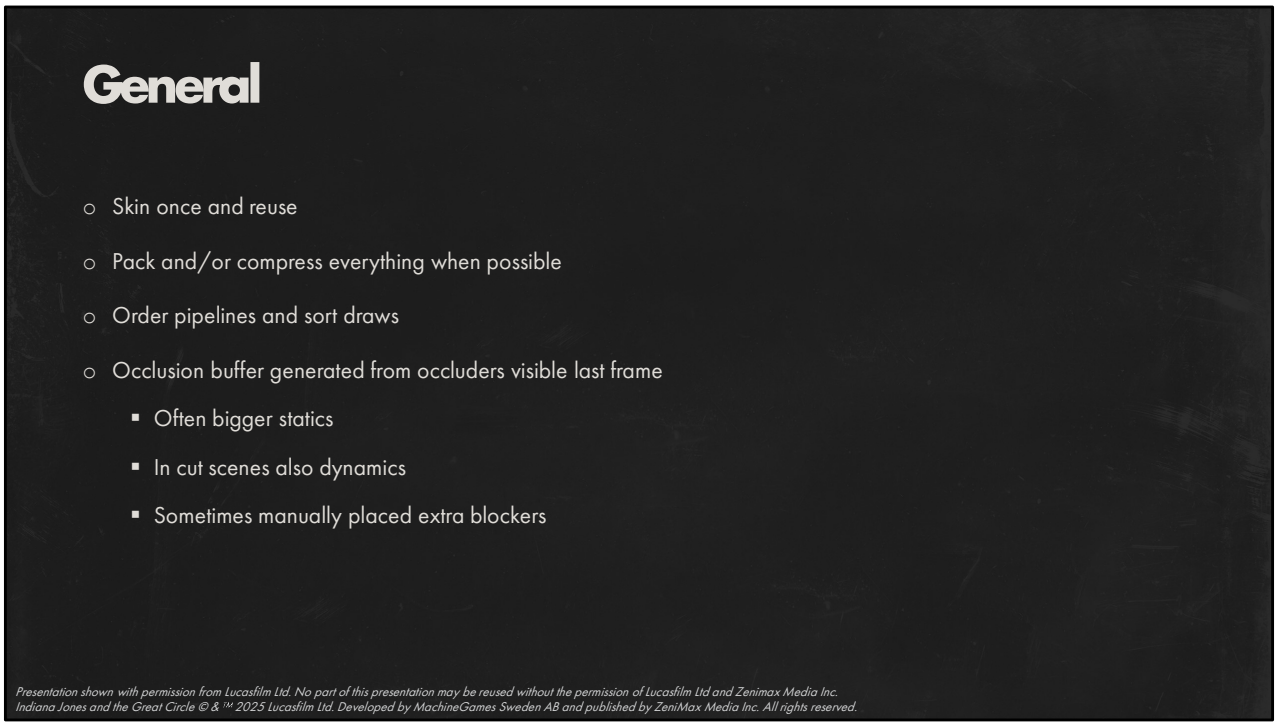


Total VRAM used.

Big jumps downward here are mostly pool size changes and changes to the streaming systems to allow that.

So what did we do optimization wise?

What follow might be a bit all over the place and mostly focuses on what we did from the code side, but in a lot of places taking captures, looking at expensive areas and talking to artists about what was costing helped us get very far. Taking that extra time to look through scenes in PIX or something similar and give direct feedback tend to be appreciated in our experience, and it helps the studio align on perf.
I'm also going to talk a bit more on how certain systems work and what we did there specifically for performance and quality.



In no particular order some general things that helps us save some time.

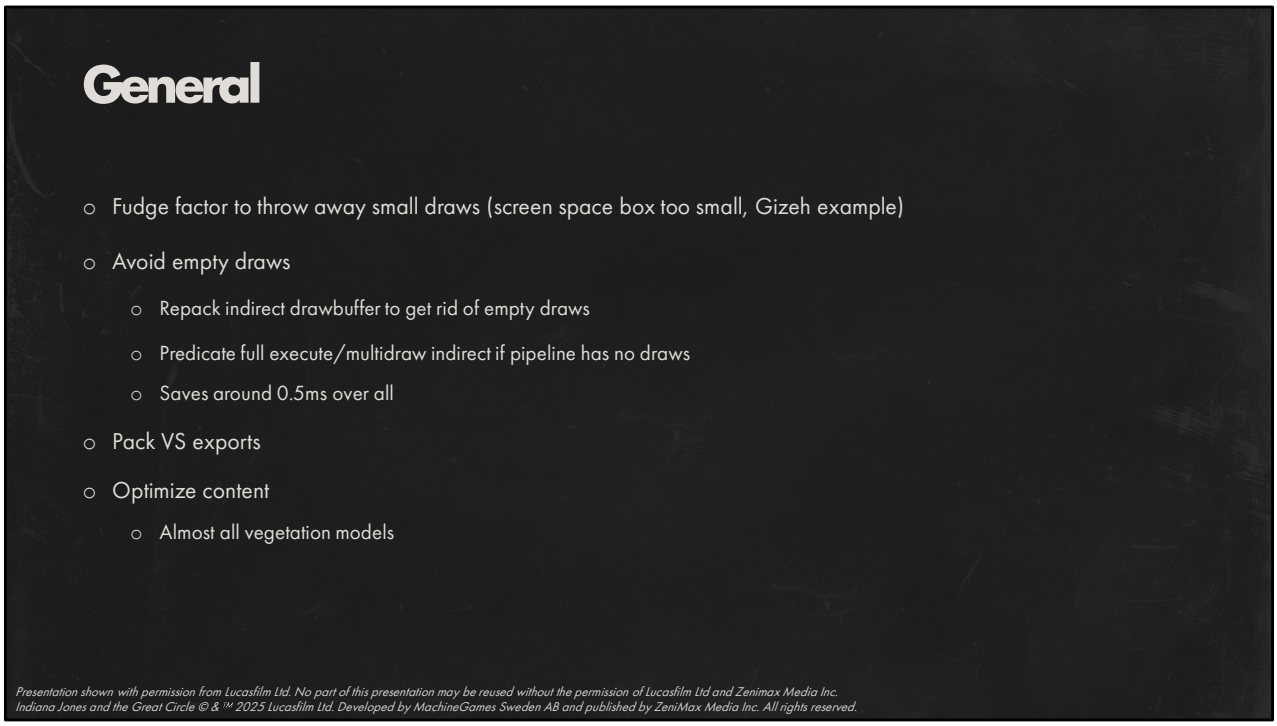
We reuse skinning data between passes whenever possible. A prepass skinning compute shader is ran, very few things take the vertex shader skinning path.

And as a general thing: Packing and compressing data helps a lot, reducing bandwidth and allowing more data or instances to be used. Things like tightly packing structs, ensuring texture formats and vertex formats are compressed.

Since we use GPU driven draws for the most parts, the order in which pipelines are drawn matters a lot. We can save some time by ensuring alpha compared pipelines are always last, giving a nice perf boost in depth and shadow passes.
For alpha tested pipelines we also coarsely sort draws to try to get rid of some overdraw. Here we don't do a global sort but instead sort in buckets of 1024 draws within one pipeline as this can be done inside one group.
This is usually enough as we rarely go over 1024 draws for a single alpha tested pipeline.

As I mentioned earlier the occlusion buffer is generated from visible occluders last

frame. As these occluders mainly are bigger statics this can sometimes not been enough.
For cut scenes we force some bigger dynamics to also be occluders to avoid having to draw what's behind them.
And sometimes we end up placing extra manual blockers (boxes or planes) to help with issues where the regular statics do not occlude enough or generate an occlusion buffer with holes.

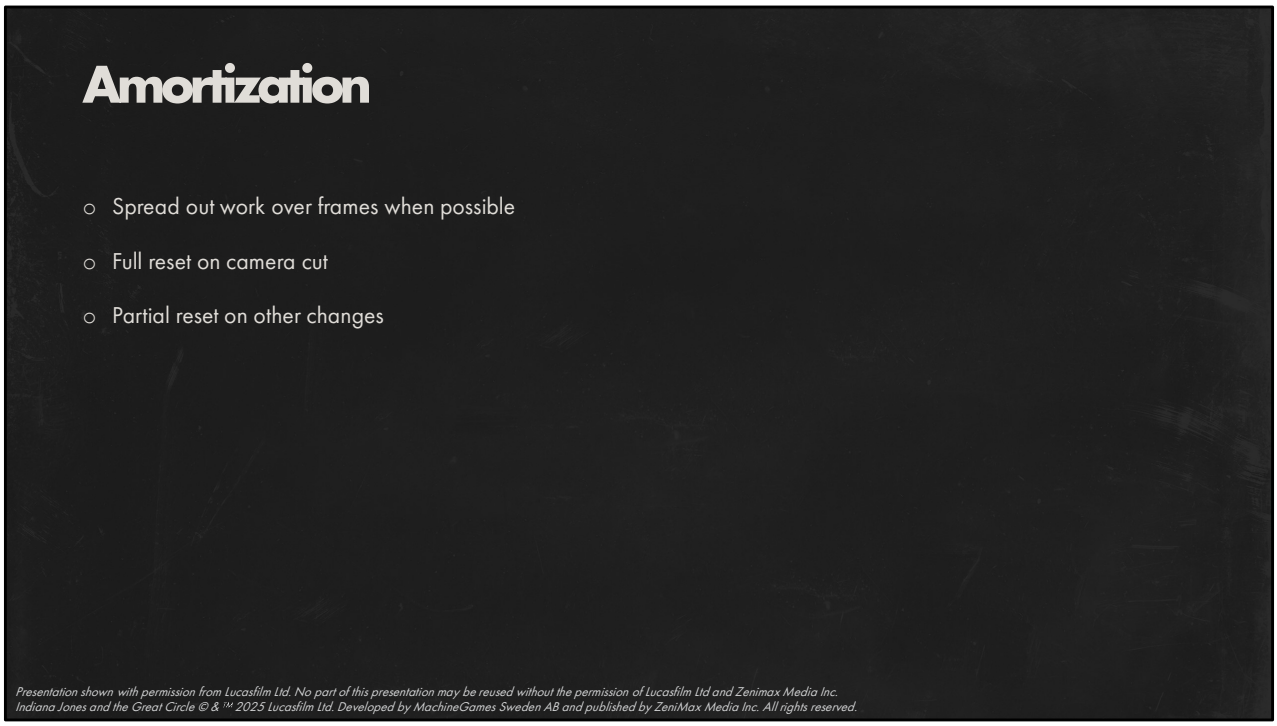


For the Gizeh scene we had an issue where we generated too many draws as the whole level was loaded at the same time. We added a small draw threshold which could be tweaked manually to discard draws that had screen size under a certain threshold, and while not perfect it was sufficient to get into budget without sacrificing to much visual quality.

To avoid having the gpu spinning up empty draws we have a draw compaction phase after GPU triangle culling that removes any draws with zero indices. This combined with a predication step to skip any pipeline that has zero draws generated was quite a boost. Somewhere around 0.5ms in scenes with a lot of pipelines but very few at time showing on screen.

Vertex program exports had some holes in it (exported full float4 when only certain values where used). Swapping to exporting floats made compiler do a better job of packing into fewer float4

We also did a huge pass on vegetation models very late in the project to simplify and get triangle counts down. Almost all of the 500 models where remade and optimized during the last two months.



As I said earlier, we try to spread out work over frames as much as possible

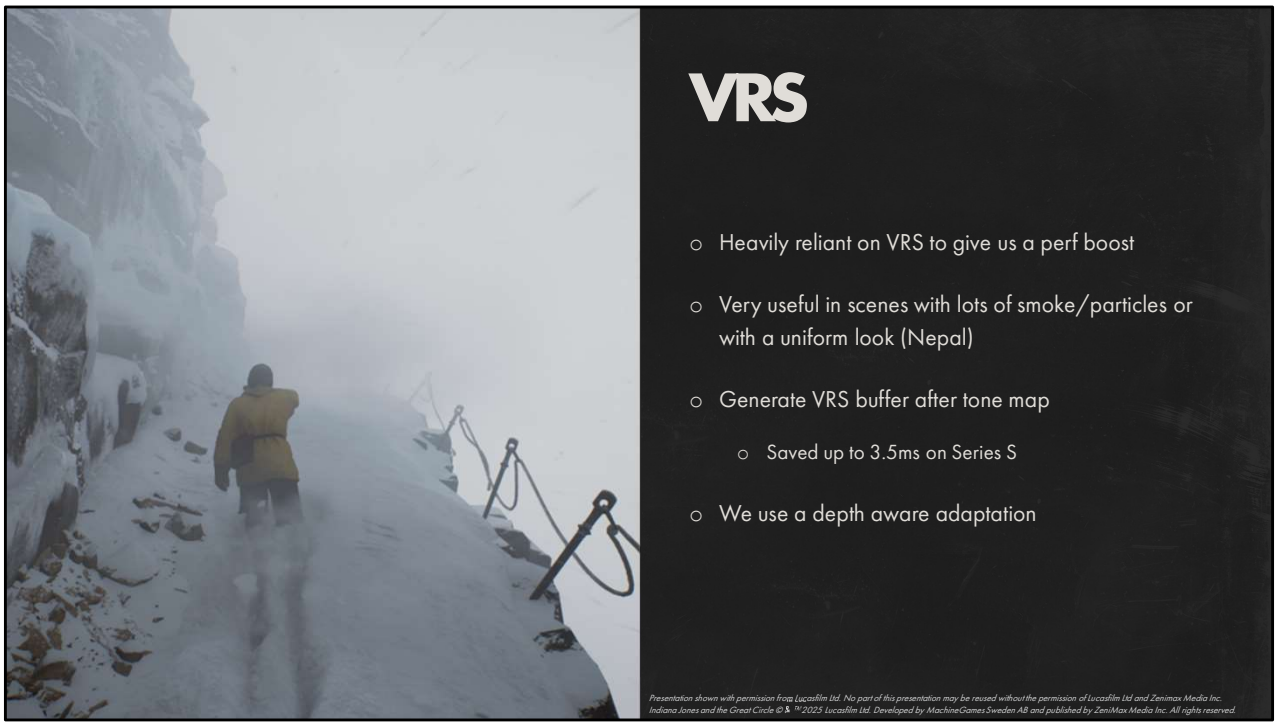
When we do a camera cut, in a cut scene for example, we do a full reset of temporal buffers and then present the last frame a few extra times to allow temporal and amortized systems to catch up.

Other things such as environmental changes can also impact temporal caches. These are cleared on demand but was a steady source of bugs when transitioning from area to area with different environmental settings.

We target full resolution but drop down in resolution if we see that 60Hz cannot be hit. We aim for this to be an exception though.

The lowest we can go is 44% pixel count, which translates to the resolutions shown here.

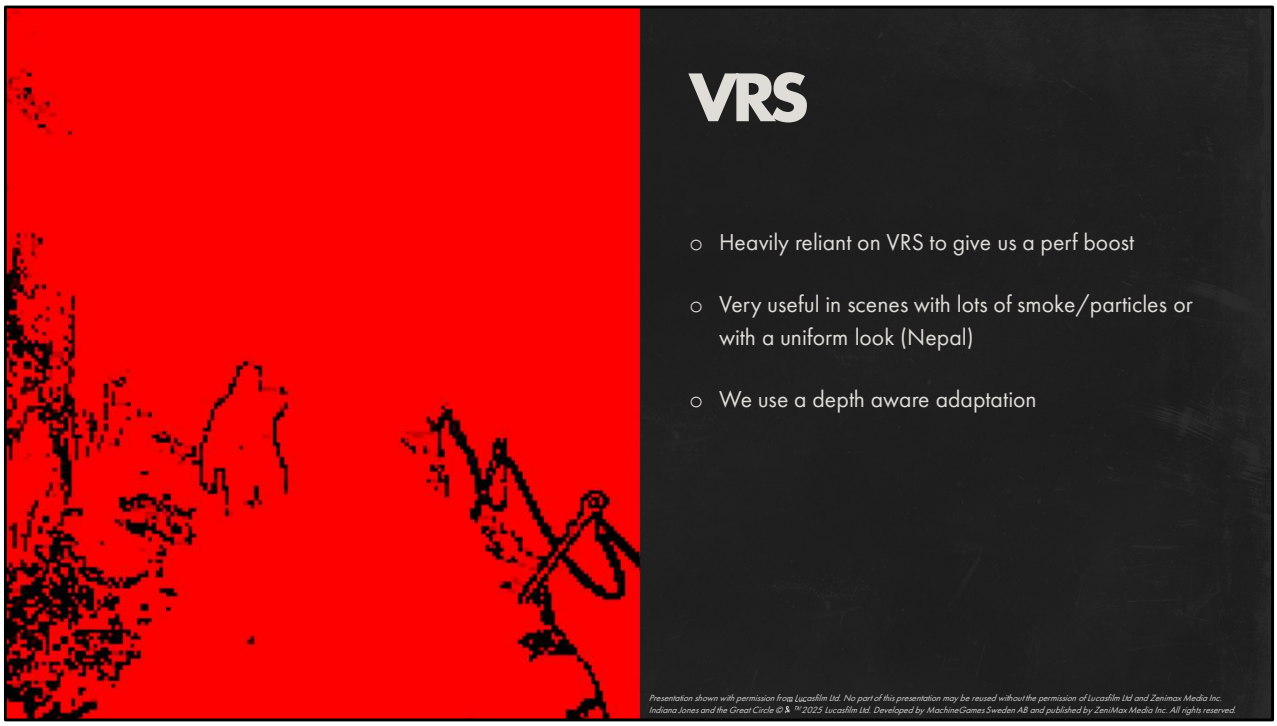
Another thing done quite late actually was that we tied anisotropic filtering to res scale. 100% resolution scale translates to 4 aniso by default, then we slowly increase aniso up to 8 if we keep hitting 60Hz. If scaling drop below 100% we decrease aniso to 2.
This gave us a bit of a perf boost when needed but also gave a quite nice boost in quality when possible. Big thanks to Martin Fuller at Microsoft ATG for implementing this.



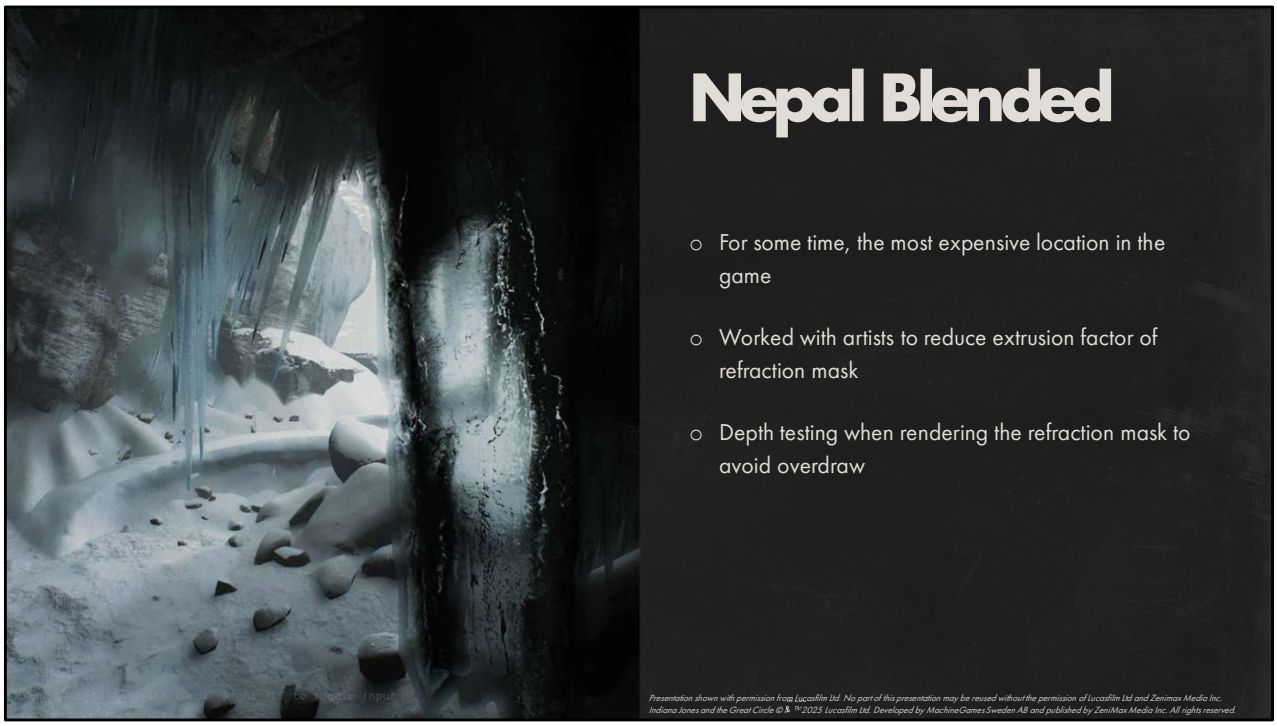
For platforms that supports it we also rely heavily on VRS to get a boost of perf in opaque and blended passes. This is specifically useful in scenes with lots of smoke or with flatter look. Nepal is a good example of this.

Initially the shading rate buffer was calculated before post processing. Late in the project we moved this to happen after tone mapping with quite good result. Overall this move could give up to 3.5ms back in Nepal, with about 1.5ms of that being from the blended pass alone.

Also, again with help from Martin at ATG, we use a depth aware implementation to get good quality along depth discontinuities.



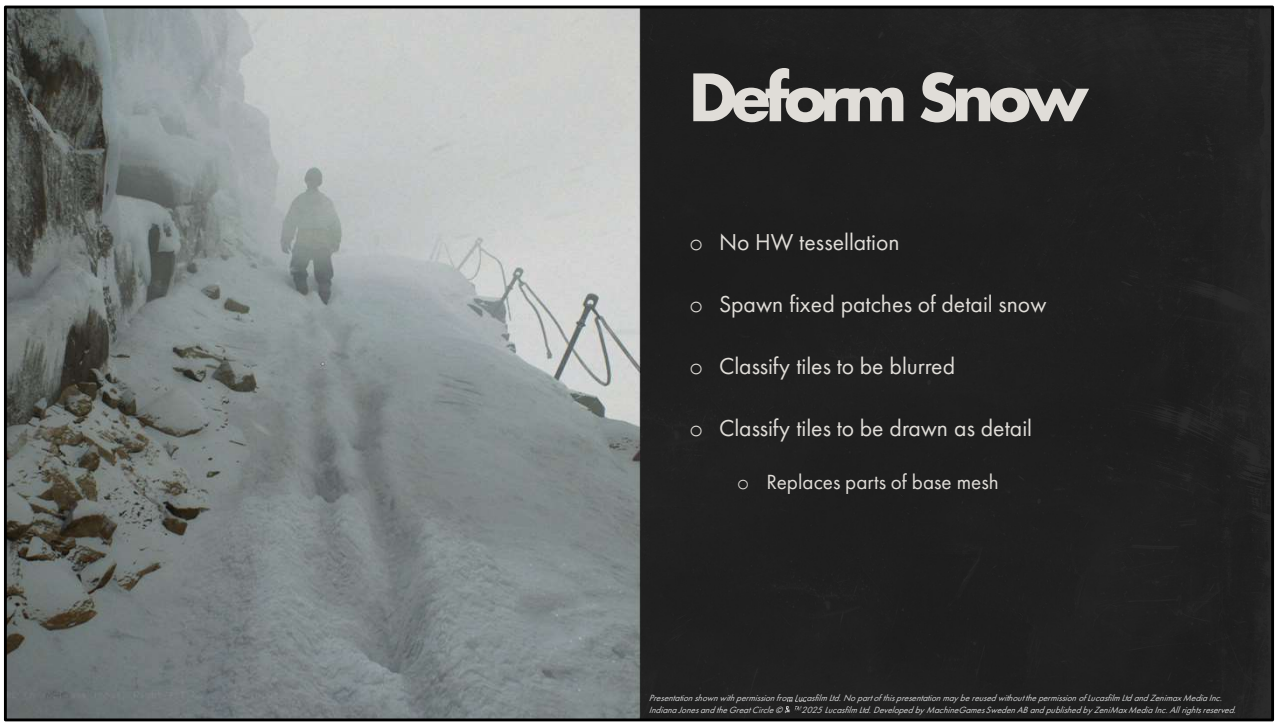
In hindsight we probably have this a bit too aggressively tuned to lower the shading rate.
It can also be a self reinforcing. Where a lower shading rate makes the final image a bit blurrier which makes us select a lower shading rate and so on.



The Nepal scenes rely heavily on blended passes and refraction to get the intended look.
For quite some time this were the most expensive parts of the game, some bit below 30Hz in places.

The icicle and ice block shaders were originally made for very sparse use where very few objects would add a bit of refraction to the scene.
Quite the opposite happened and the setup with the original intent did not scale at all.
We ended up taking a big pass with artists here, swapping things to use cheaper shaders where possible, reducing overdraw.

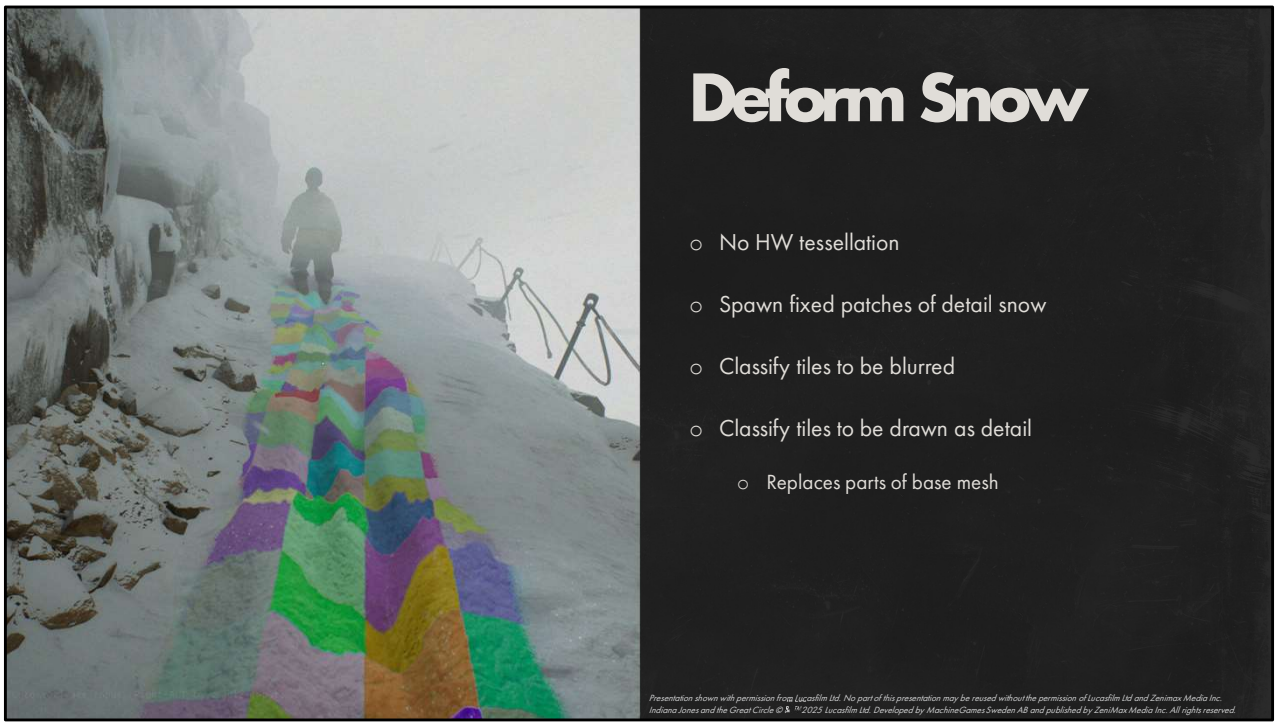
We also did a big pass on the refraction mask buffer we render before blended passes, which draws a normal-extruded version of the mesh to mark what can be seen from refraction.
This pass suffered from bad overdraw so adding a depth buffer for this pass combined with artists tweaking the extrusion ranges helped a lot.



Staying in Nepal: Our snow deformation do not rely on hardware tessellation, instead we have a fixed base mesh of deformable snow and we replace parts of it with a deformed mesh when needed.

We use a camera centered top-down projection of deformers drawing deformation depth into 2k x 2k render target, and most of the area isn't deformed in the majority of cases.

Group shared memory and waveops are used to downsample, such that 1 pixel in downsampled target equals filter kernel size.
After this we do a tile classification of what tiles needs to be filtered
For each pixel in downsampled target, we write atomically to a tile ID buffer with one extra border neighborhood for account for blur filter kernel size.
This saved ~1ms on Series S on average, compared to a full blur.

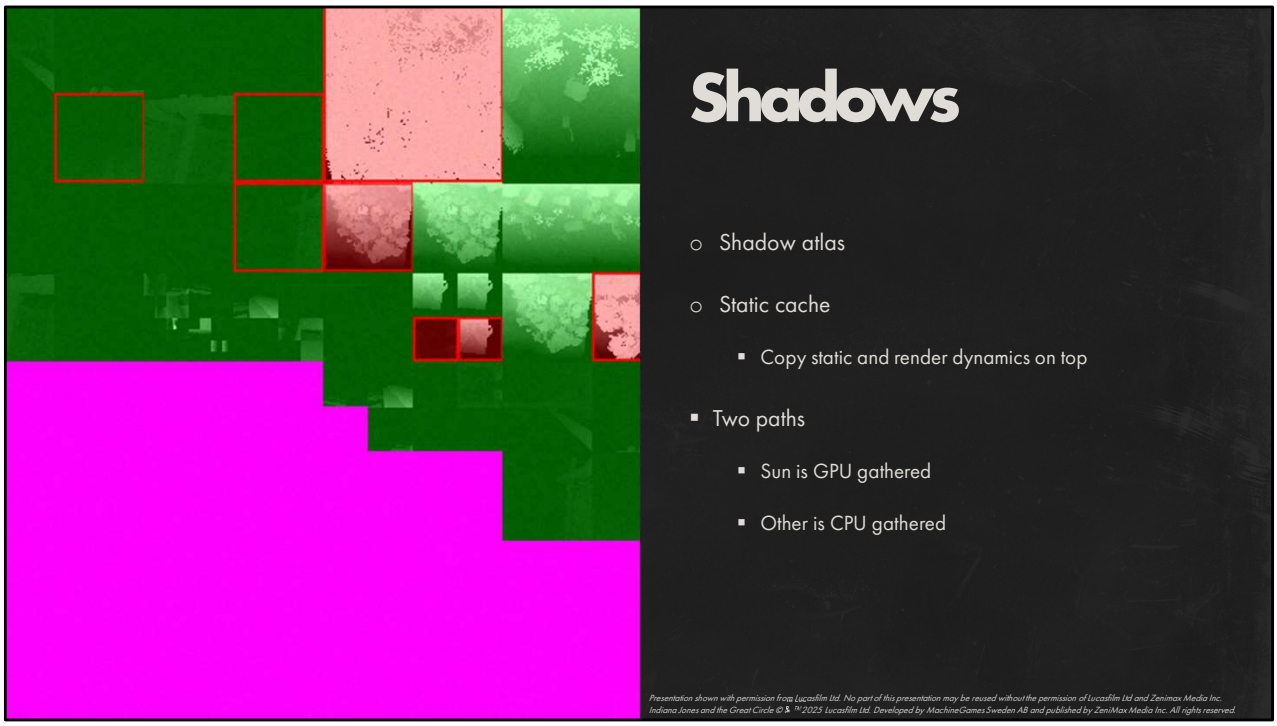


Each detail patch maps to 16x16 pixels in the full deformation target. One vertex per pixel.

We use downsample setup similar the filtering to determine how many detail patches to build indirect draw args for.

The base mesh is discarded in detail patch areas and a dither is applied between detail meshes and base mesh to hide intersection. Neither of these meshes are included in acceleration structures.

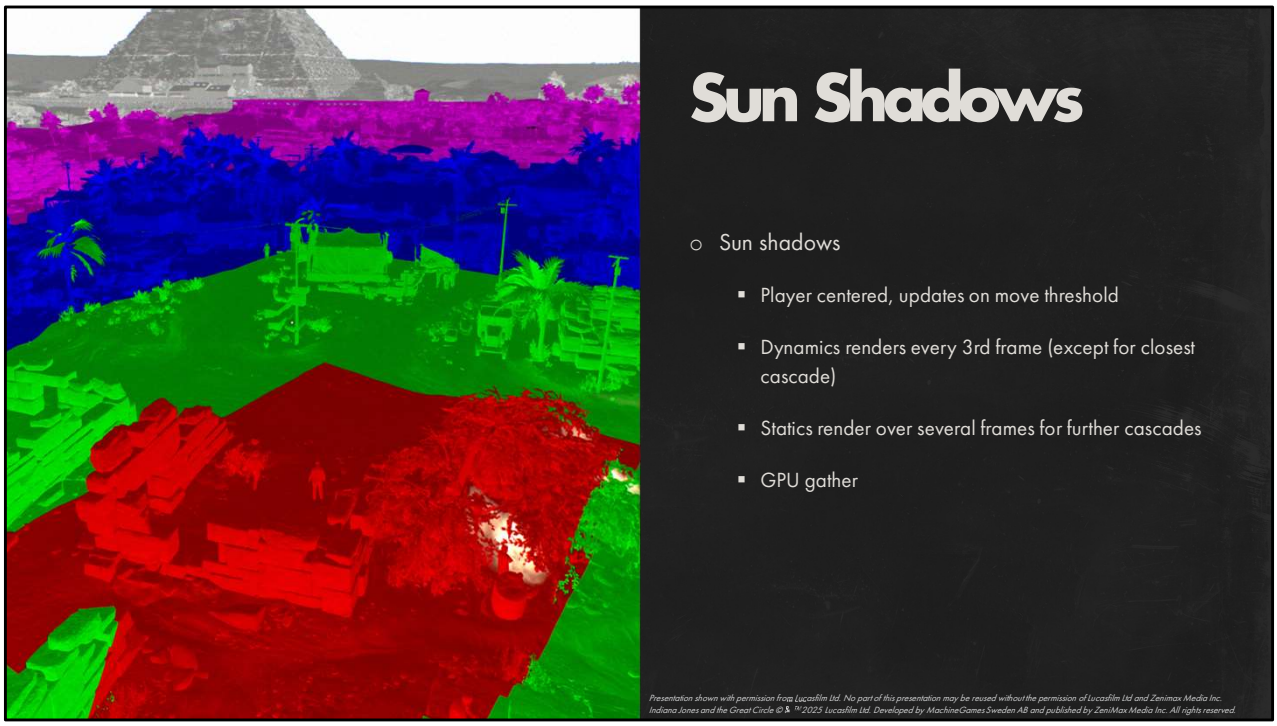
The split between detail mesh and base mesh was made because it allowed the use of static low poly base meshes with vertex painting to cover up seams.



We use a shadow atlas in which we allocate tiles of different size depending on what quality a light needs. For sun this is a static allocation but for other lights we prioritize lights so that we can fit the most important ones in.

As most lights are not moving, they allocate a “static cache” tile as well. This is an optimization we do where we render static geometry into a tile and then we can copy that to a composite tile each frame and render dynamics on top of it. If needed.

We have two major paths for shadow map rendering: GPU for sun shadows and CPU for all other light types. The reason for this is simply the GPU path landed quite late, and we did not have time to get that working for all light types.



We use 4 sun cascades which are all player centric and updates once the player has moved a certain distance threshold.
This means that the player can turn around and move a small distance before we need to redraw the static part of the shadow map.

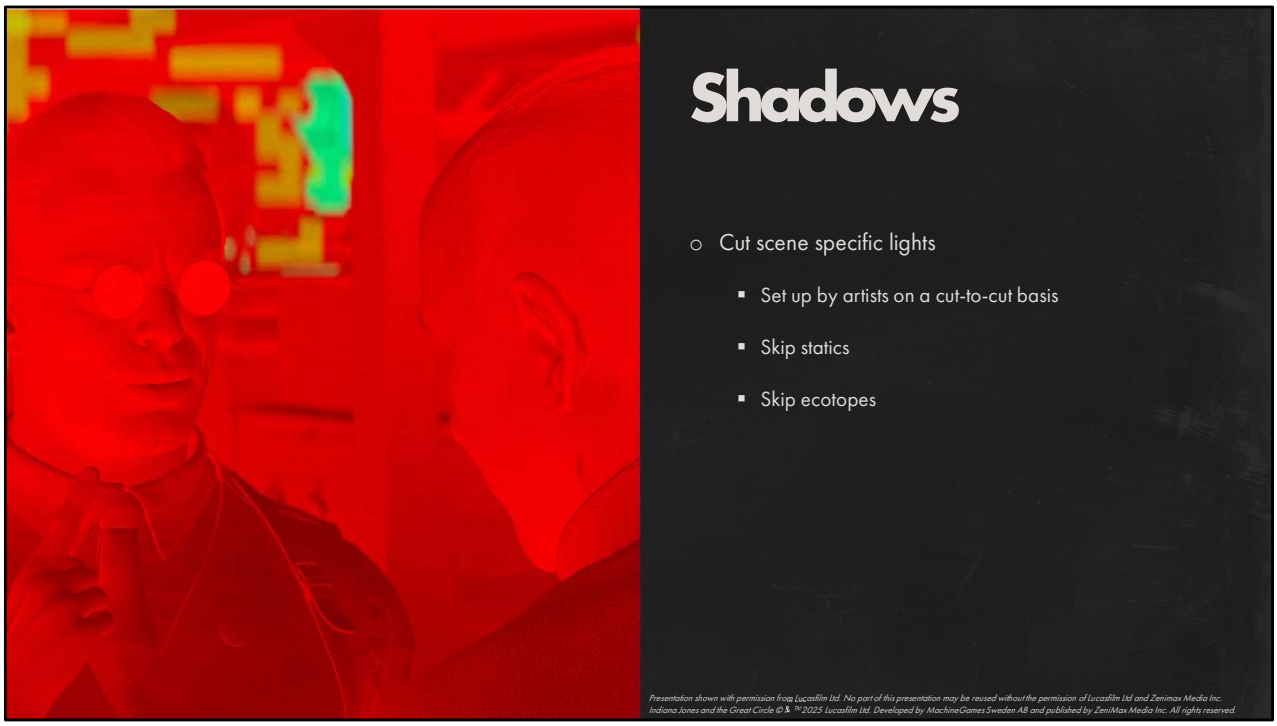
Closest cascade redraws dynamics every frame, and further cascades alternates to render every third frame.
Dynamic vegetation is rendered as static for far away cascades to avoid a lot of draws.

The Static cache is amortized over several frames for the further cascades, spreading out the load of updating over up to 30 frames.

What goes into the cascades follows a fully GPU driven path (very similar to opaque rendering) but with some shadow specific steps.
Once a static cache slot is done, we generate a downscaled occlusion buffer for the cascade and use this to cull dynamics.
Dynamics are also discarded if they can't cast shadows into the view frustum.

Another big difference is that shadows do not use GPU triangle culling, as we did not

have time to integrate that.
In the end we ended up with a lot of small triangles so if we want to keep this system, we probably need to spend some time on that or something equivalent.

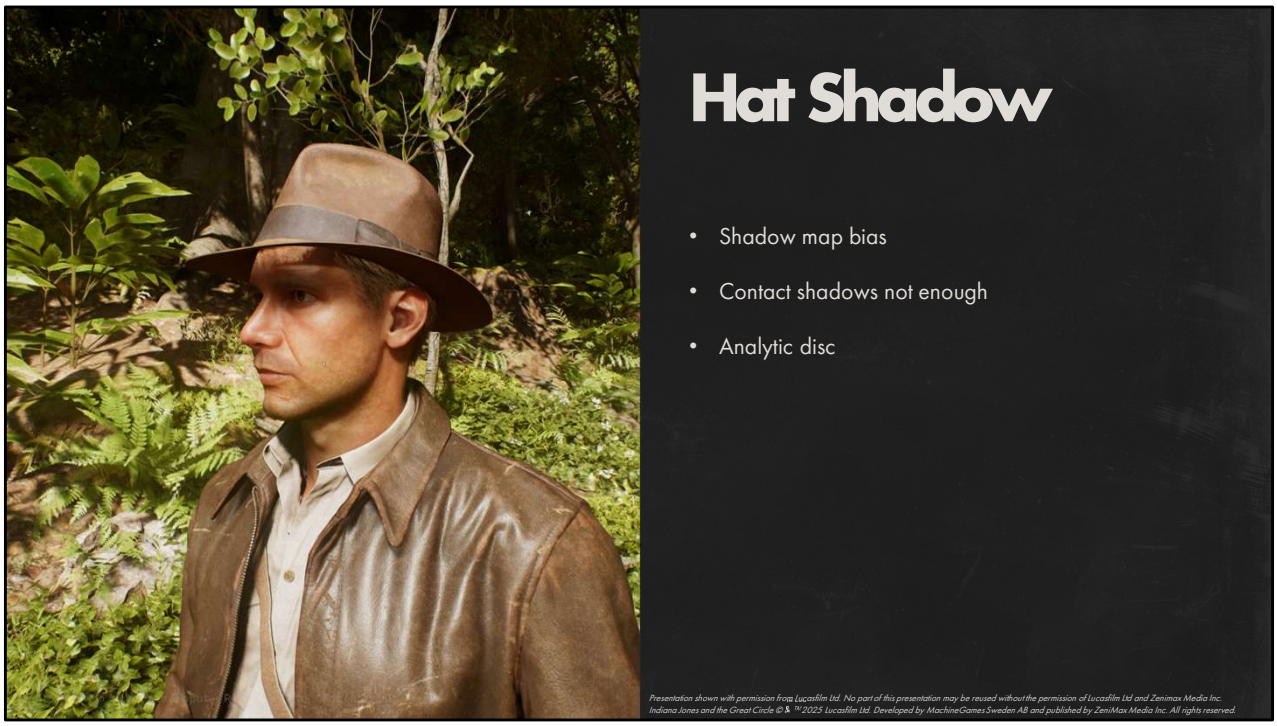


For cut scenes we end up with a lot of lights close to the camera, and additionally these lights often have high quality requirements, but they often only affect just character faces.

Here is an image from our light count debugger, showing that almost everything in this cut scene is hit by quite a few lights.

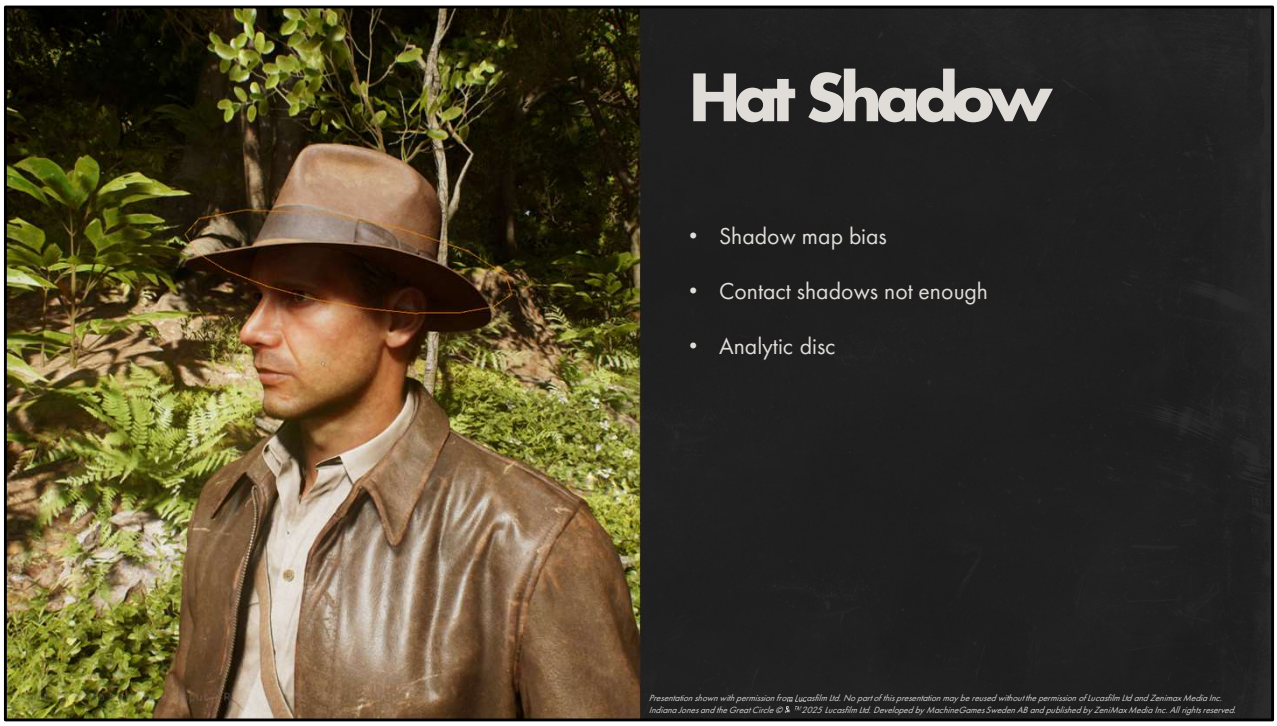
We ended up allowing these lights to skip parts of the world to gather shadow casters from, controlled by artists on a per light/scene basis.

Indys hat is a very important part of the visuals, but we had a lot of issues with our shadow maps having too much bias to make this look good.
Often there was visible light showing up under the brim of the hat close to the forehead.

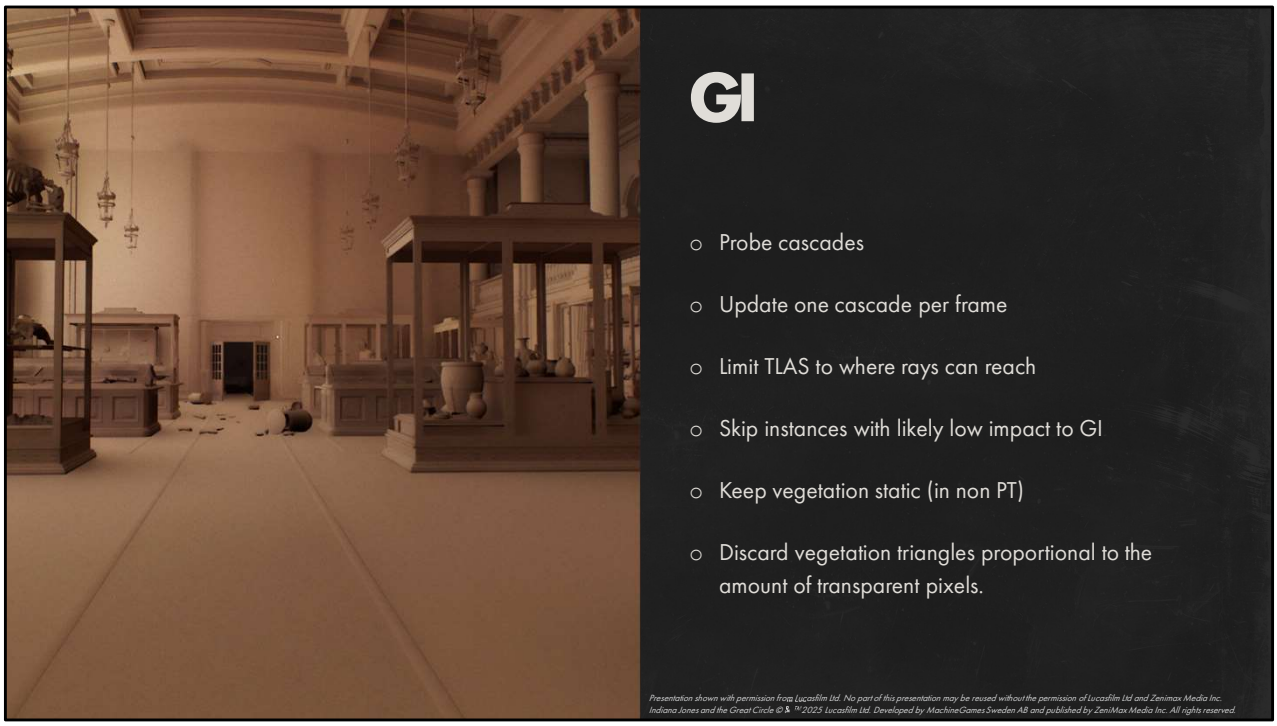


Screen space depth buffer tracing, also known as contact shadows, solves part of this issue but in a lot of cut scenes this was not enough.

We ended up placing an oriented disc on a joint on the hat so that it would pick up the transform from animations.
Shadow receivers on the same character setup would then get this data so it could do an analytic ray disc intersection to get a precise artist-controlled shadow.
We do something very similar with hair as well, where we place clipping spheres to discard hair strands so they won't clip through head wear.



Here is a debug view of where the disc is placed.
It’s not perfectly on the brim of the hat, but rather slightly above to avoid light leakage.



Our GI is based on a set of ray traced probe cascades, and similar to shadows we update one cascade every frame. The difference being that we don't fully update probes, we blend in new contributions to let the probe converge over time.

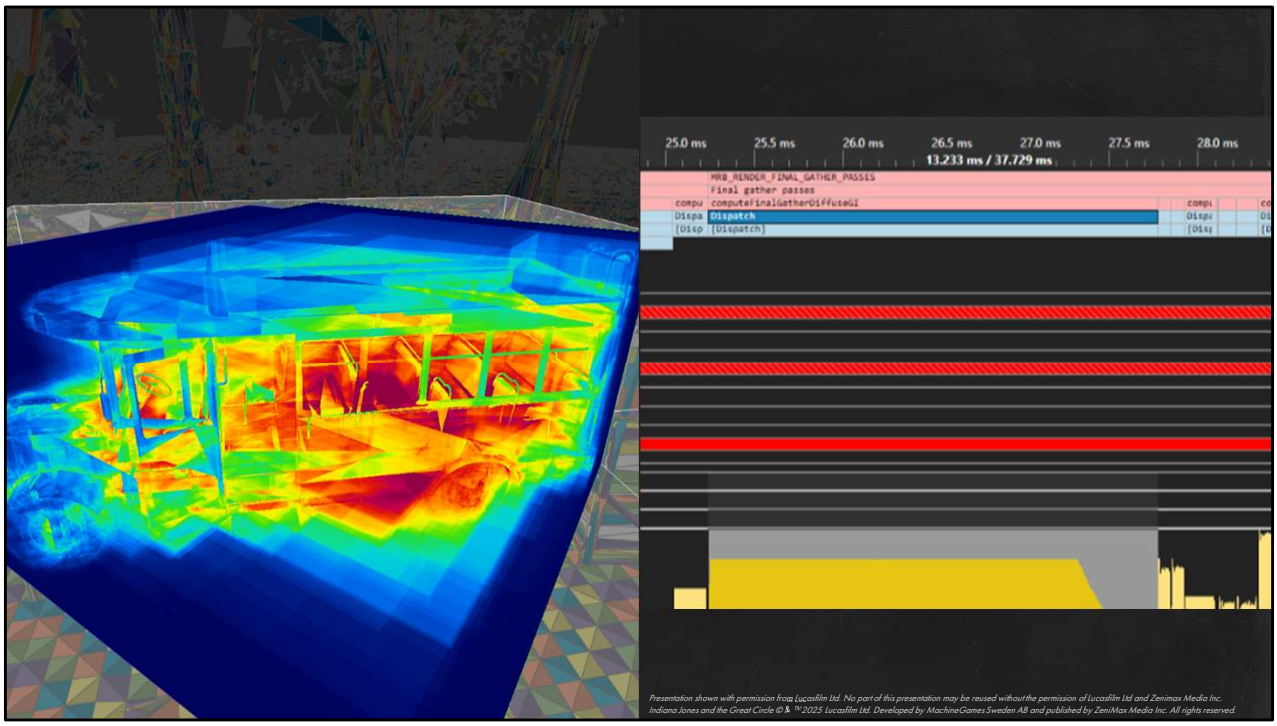
We calculate how far rays can reach and limit our top level acceleration structure to only that area. Also we skip any instance we deem likely to have a low impact on GI. Here we also amortize updates of dynamics, so not everything that animates is updated every frame.

For vegetation we have a bit of a simplified representation. We skip wind animation and do not support alpha compared vegetation for ray tracing.
Instead, we discard triangles from alpha compared meshes proportional to the number of transparent pixels in their opacity map.

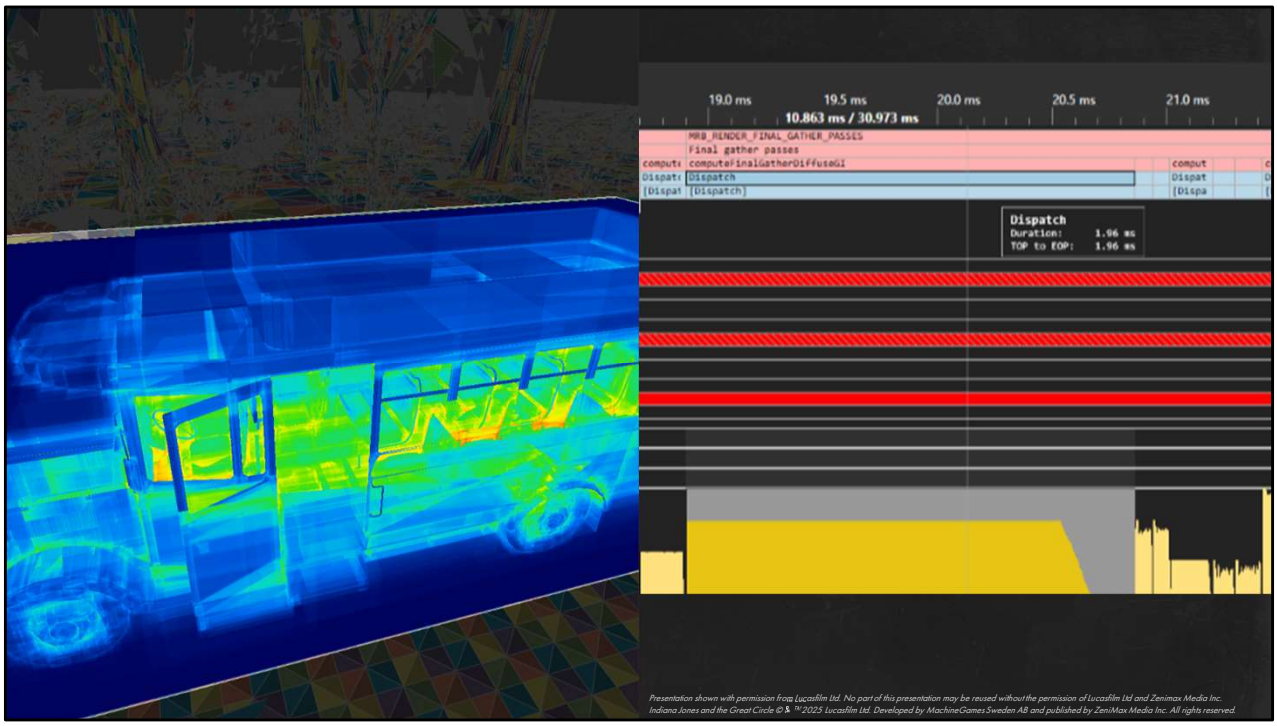
Bottom level structures are built offline where supported. This allows us to free up time to update more dynamic acceleration structures instead.

Late in the project we worked a lot on reducing VGPR pressure for hit shaders. For platforms that supports it we set the pipeline up to be able to spill registers over a certain threshold.
As long as the most used hit shaders where under this threshold it gave a small net perf boost.

One thing that we found very important is to periodically look at debug views to see ray traversal hotspots

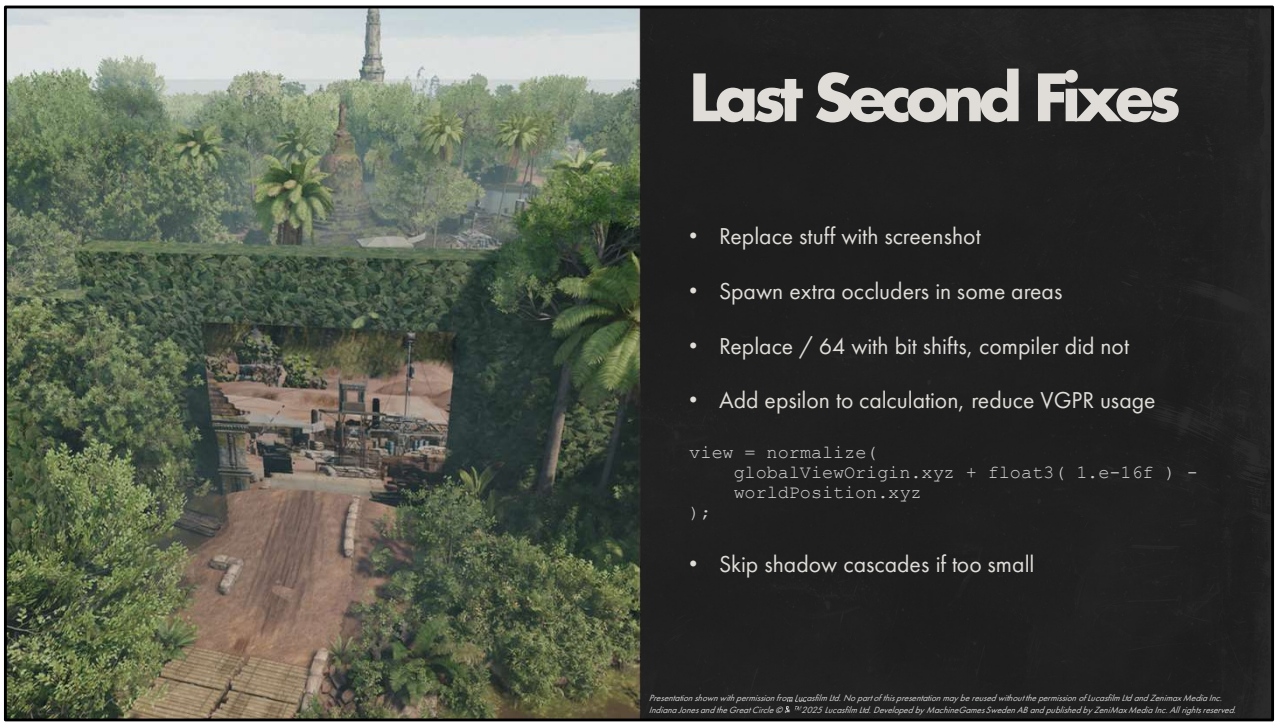


Here is a view from a bus from the intro cut scene to Sukhothai intro.
This bus was built axis aligned but was set up as part of the cut scene so it was moved around using a joint and ended up getting built as a dynamic acceleration structure at a 45 degree angle.



Art reworked this setup so that we could get a proper aligned local bound which roughly saved 0.5ms

And what's a shipping period without all the last second fixes to actually ship?



Here is a few of the things that was done:

Replace view through a tunnel with a screenshot and occluder plane to effectively discard all things behind it.

In some areas such as the treehouse extra large occluders where spawned to cut away unnecessary things that did not contribute to final frame quality

And some stuff that perhaps are more like compiler workarounds:

Division by 64 was not automatically replaced by shader compilers with bitshifts everywhere. Manually replacing that with bit shifts removed 60+ instructions in places

Adding an epsilon to an expression used in many places forced the compiler to rematerialize it, saving VGPRs and allowing us to go under 60 VGPRs in places. At the cost of shader instructions.
Sometimes this can be a tough balance to struck. It might be better to do repeat calculations to reduce VGPR pressure and get better occupancy.

And last, some levels with an overcast sky had some really strange shadow cascade setups with degenerate or clamped cascade sizes. We added code to skip any cascade too small to do anything, avoiding having to have the whole world render to just a few pixels.

That’s all for me. I will try to answer questions in the Q&A and hopefully more people from the team will be able to join in the discord channel to answer what I cannot.