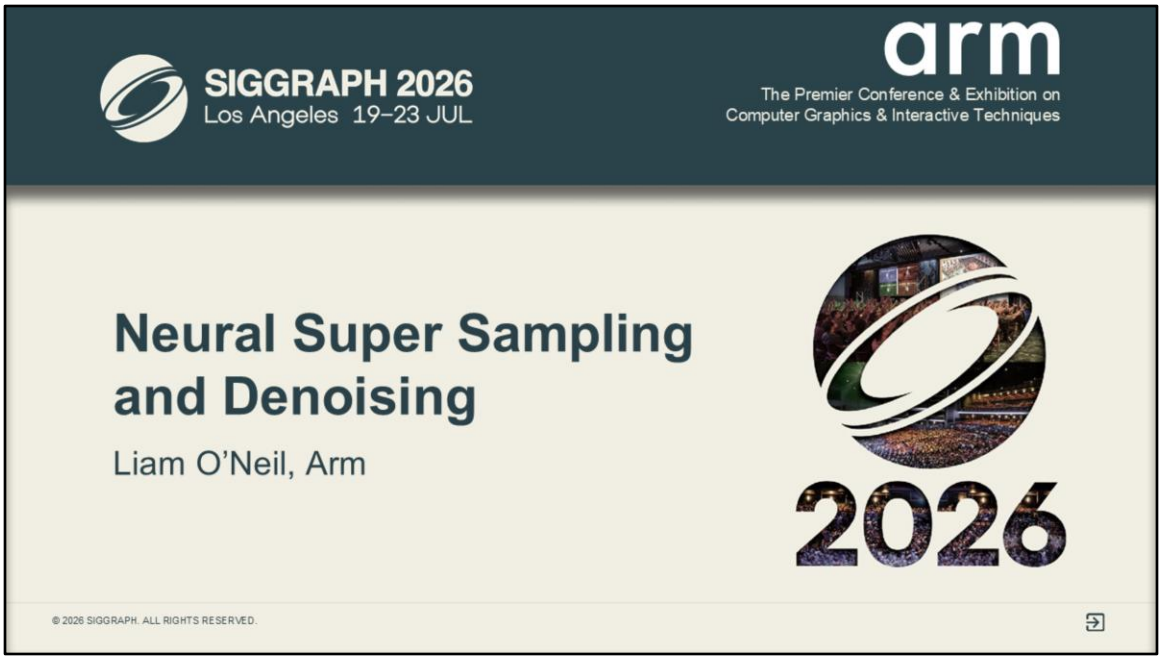
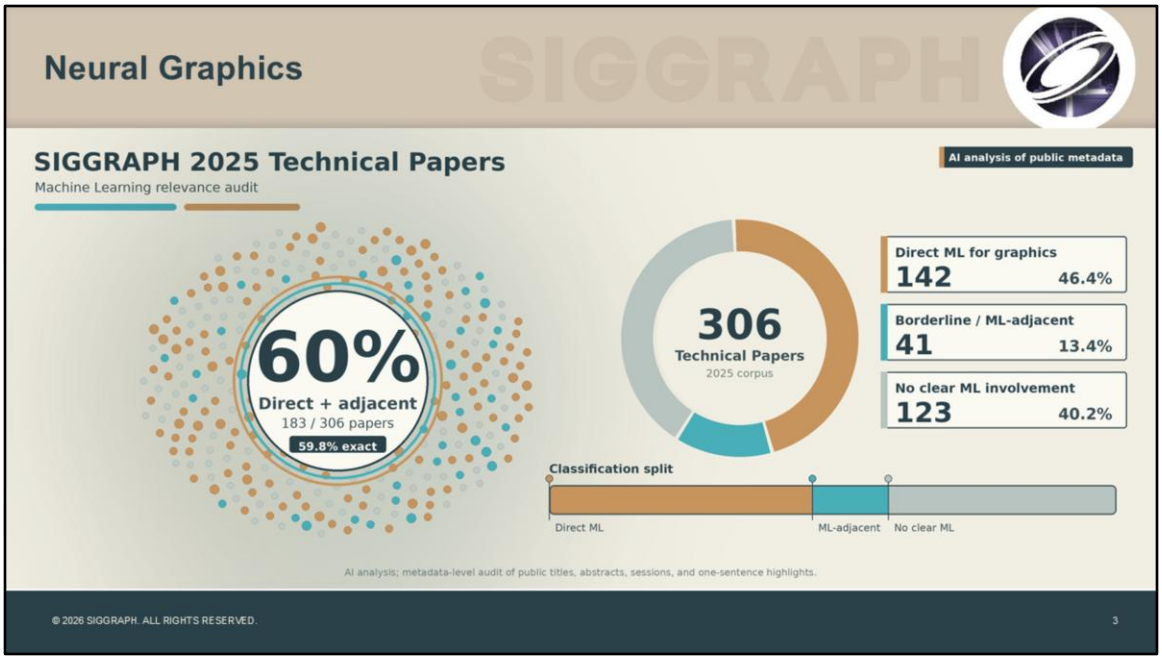


Hello everyone, it's great to be back at siggraph.

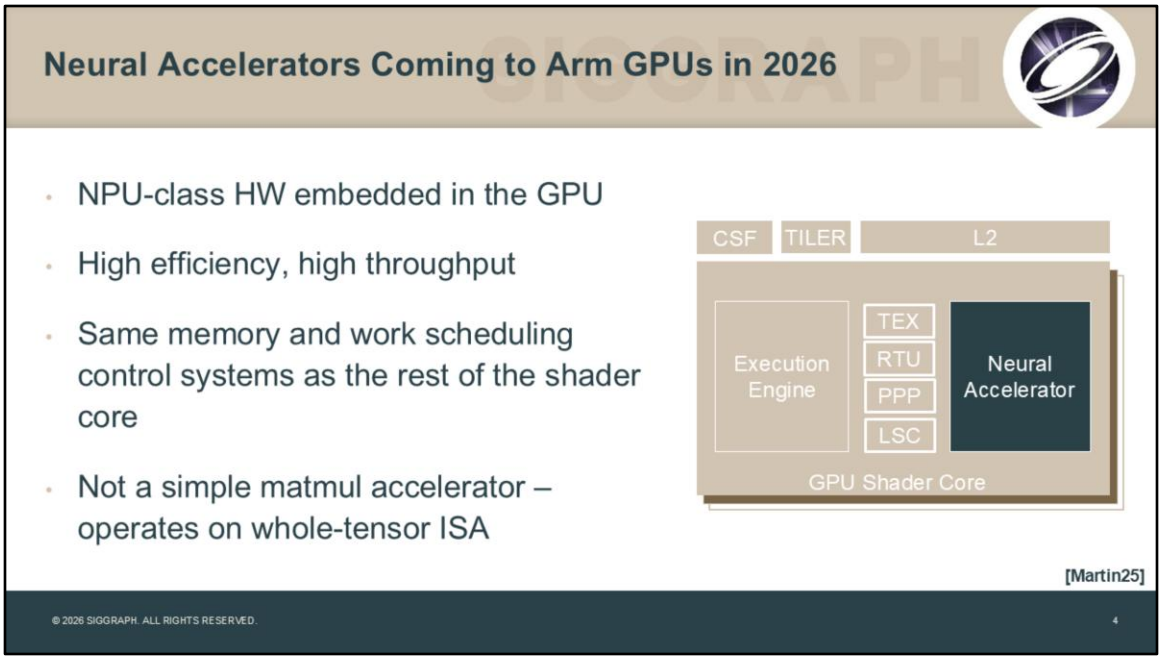
Today I'm here to share some of the work and learnings, from my team's effort to extend our neural super sampling solution to support denoising.

Before I get into the technical details, I wanted to give some context and motivation as to why we've been working on this.



So, to set the scene, naturally in 2026, I asked an LLM what percentage of technical papers last year were related to neural graphics, it told me 60%, maybe the LLM has vested interest, but I can believe this number.

Looking around the conference, again this year, neural approaches continue to be a dominant frontier of research in graphics.



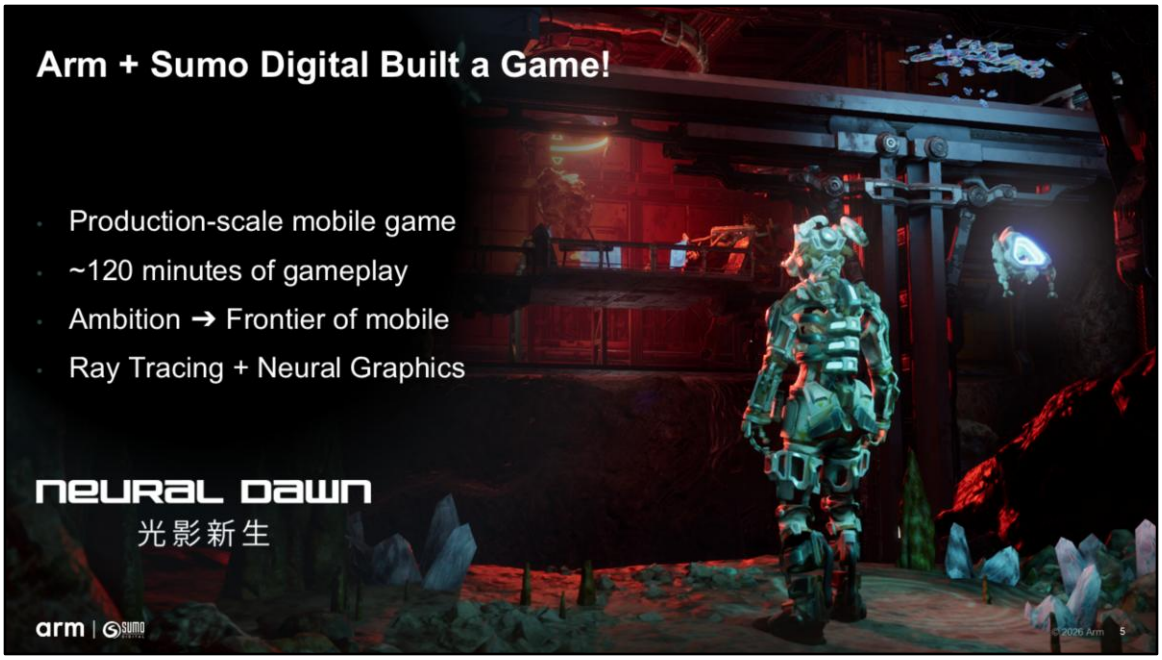
And in order to enable the transition from research to deployment, last year at SIGGRAPH, Arm announced neural accelerators inside of our GPU's, this will be in silicon products later this calendar year.

In a nut-shell, this adds:

- High efficiency, high throughput, npu-class accelerator
- In the familiar GPU form factor, using the same memory sub-system and work scheduling as the shader core

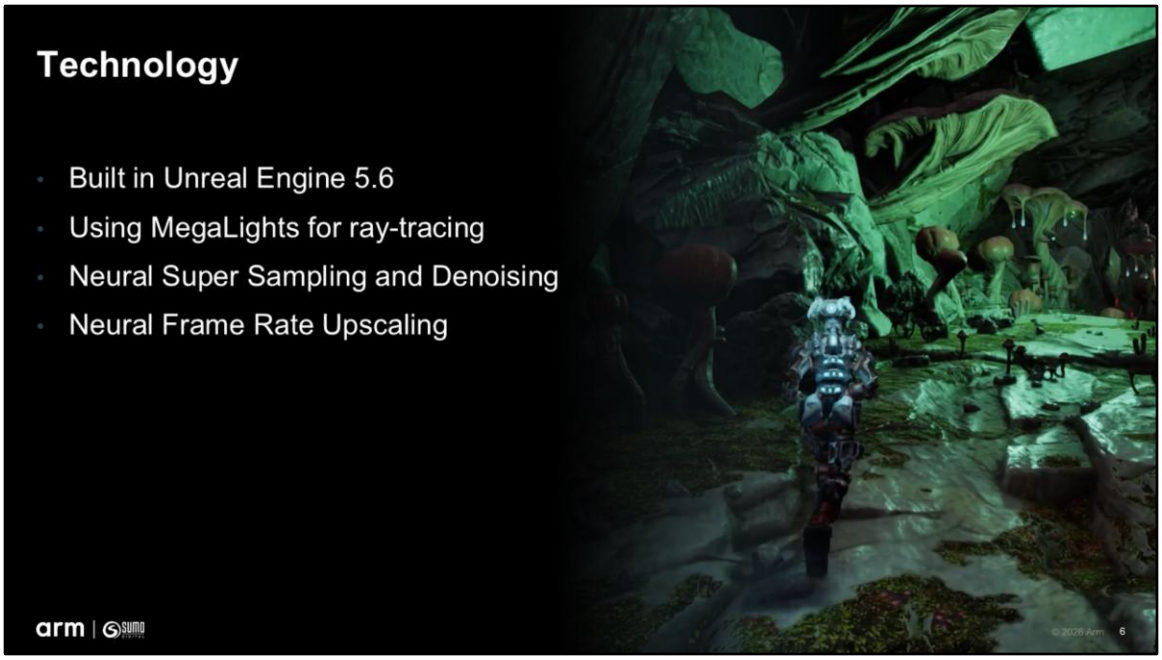
In practice, to use this we've added extensions to the Vulkan API, which give you a familiar developer experience, you have full control over synchronization, the ability to share resources, and dispatching work to this looks just like a compute dispatch

I promise, this is my only hardware vendor specific slide, but I just wanted to paint you a picture of the dog food.



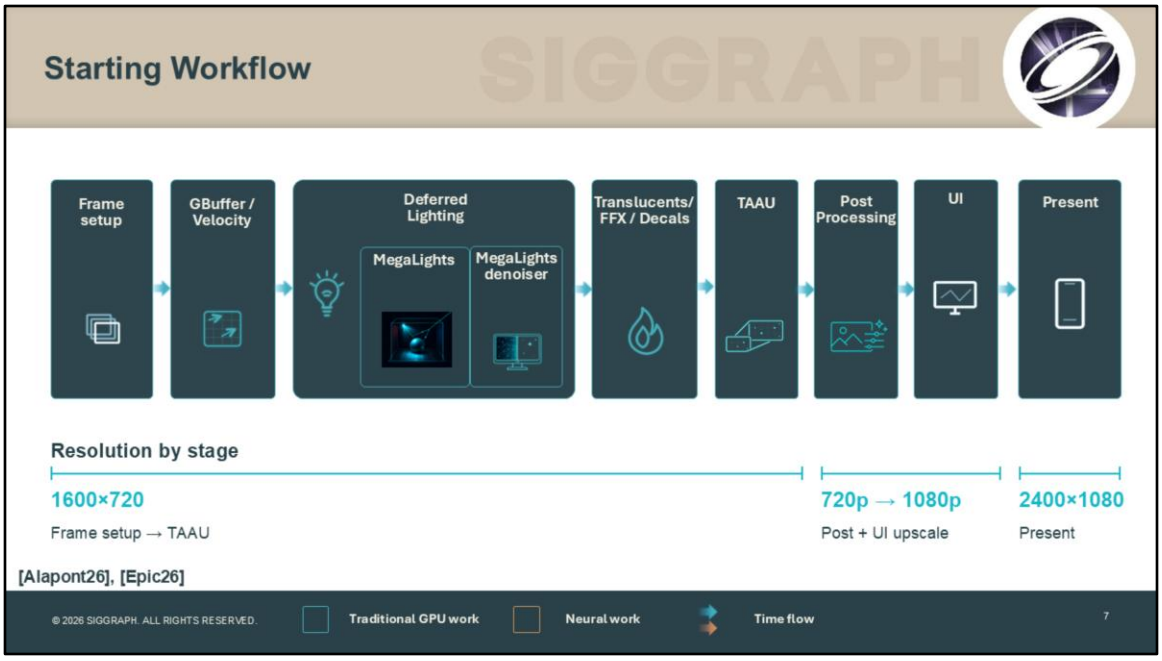
Because for the last year, we've been eating it!

Alongside our great partner Sumo Digital, We've built a production mobile game, it's around 2 hours gameplay, called neural dawn, and it's been created with the sole ambition of demonstrating the frontier of mobile.



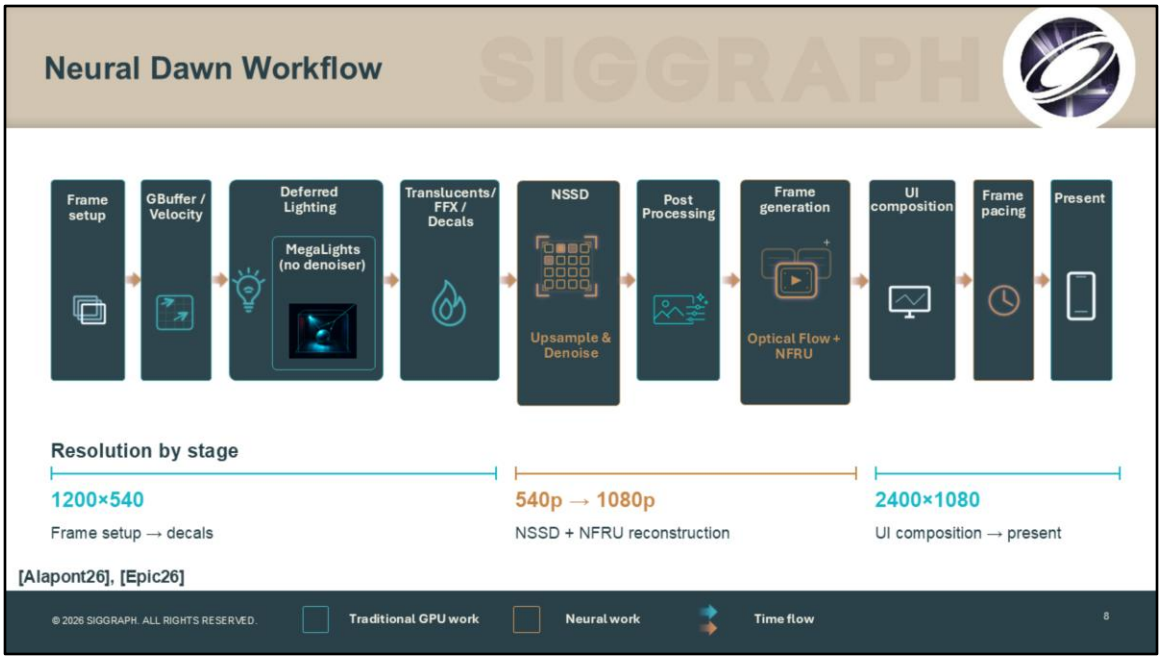
For a roofline view:

- It's built in Unreal Engine 5.6
- We're using megalights for 100% ray-traced direct lighting, using 100’s, and sometimes 1000’s of lights per scene
- And to enable the best possible performance and quality, we’re using neural techniques for spatial upscaling, denoising, and frame-generation

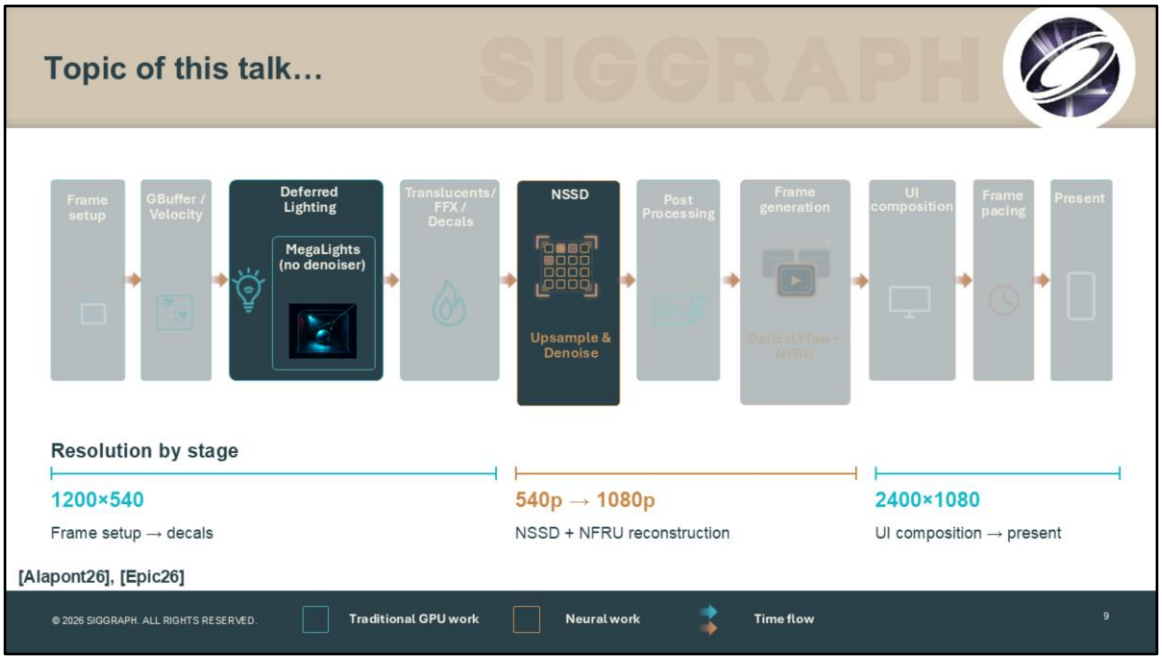


So, I'm not an Unreal expert, but this is a high-level pipeline overview of our starting workflow in the engine.

We're using a branch where MegaLights is still tagged as an "experimental" feature. It wasn't quite ready for mobile, and so didn’t meet our performance and quality aspirations.

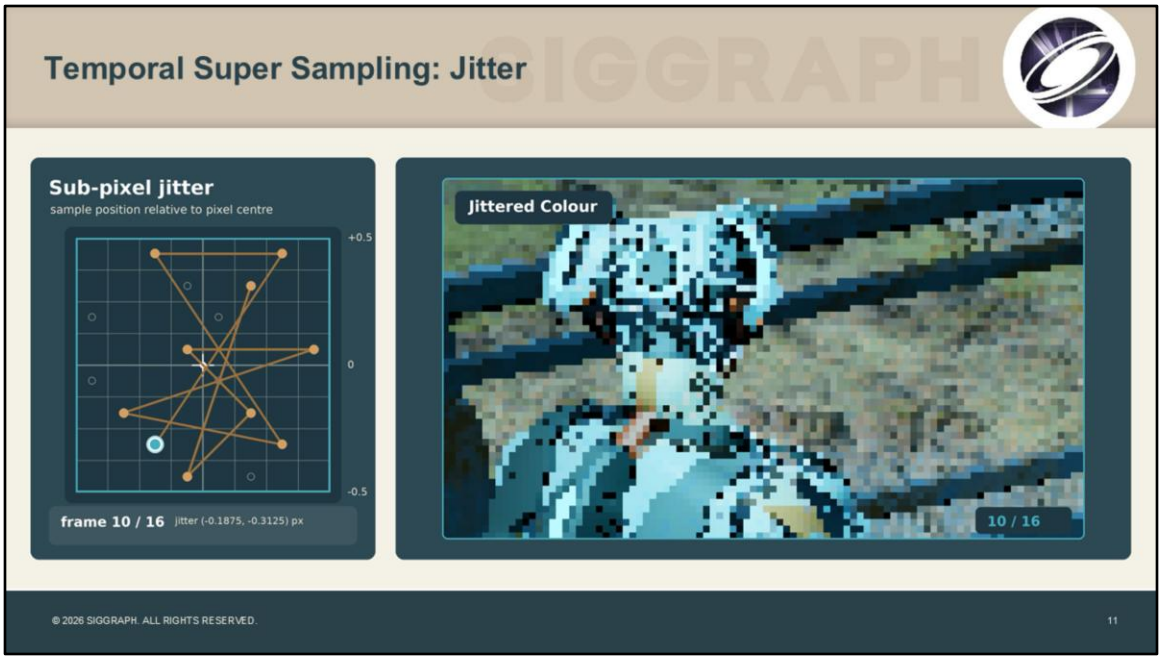


During the development of neural dawn, we've arrived at a pipeline that looks like this, where we've added our NSSD technology for upscaling and denoising, alongside our frame generation and pacing technologies.

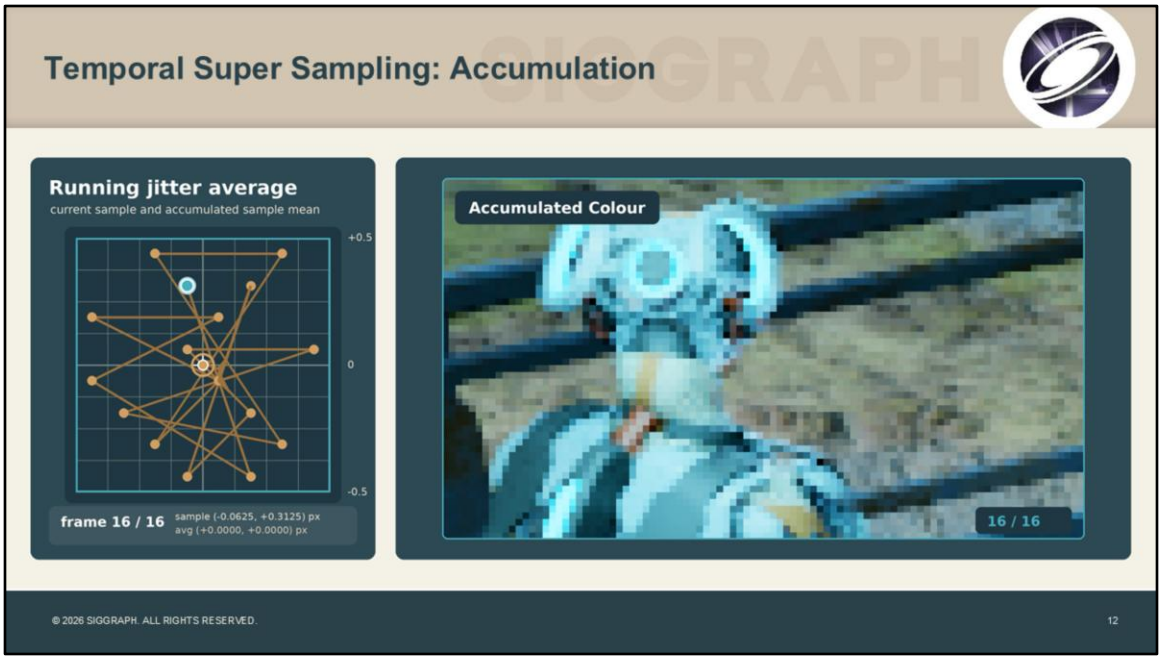


There's a lot to talk about, but my talk today, is specifically focussed on the denoising tech we've built, sharing some generally applicable insights around how we made it efficient for mobile, and some small details around how the team got this working nicely with MegaLights.

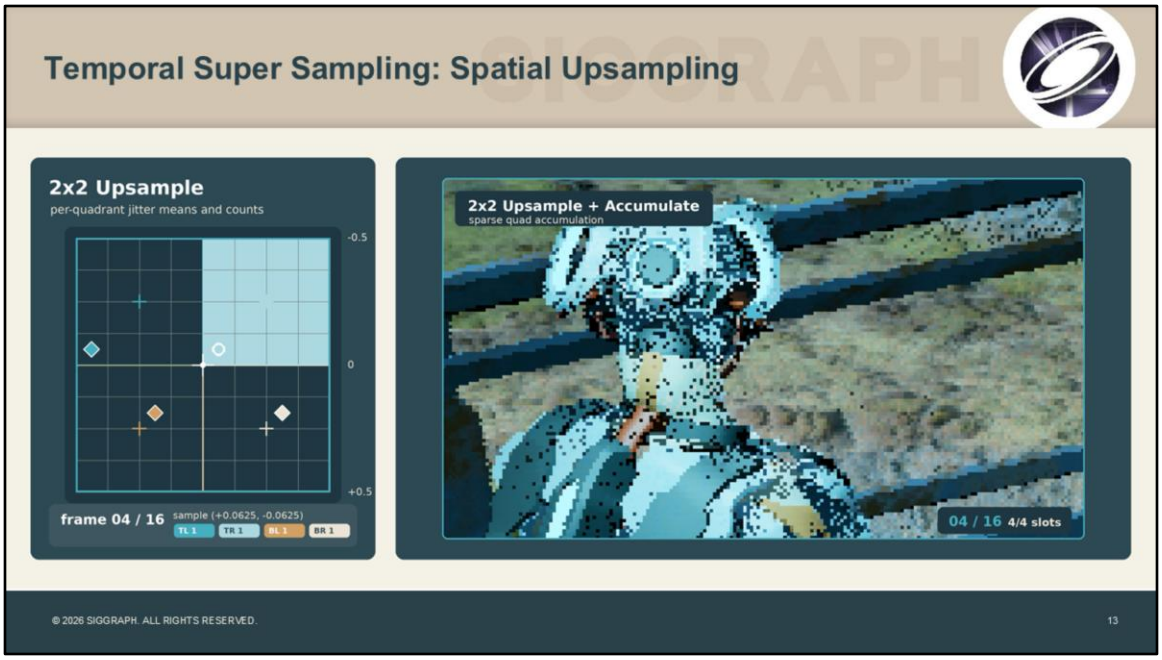
So, whilst we've presented in this forum before, and Daniel Craig covered it excellently in advances yesterday. I wanted to provide a high-level overview of how the super sampling technology works, as it's the foundation from which we have built off.



So, temporal super sampling starts with jitter, for each new frame we render, we apply a small sub-pixel offset to the camera matrices, enabling us to render from a novel sample position for each new frame.



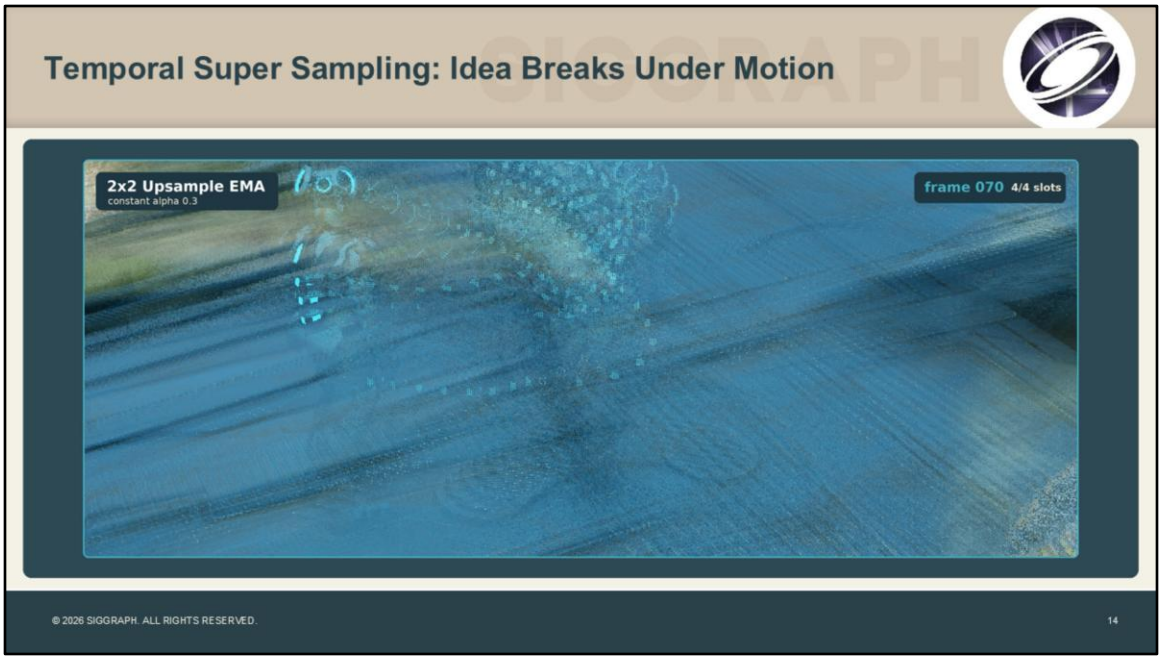
We can then accumulate each new sample, using an exponential moving average, which resolves aliasing over time, with the advantage of amortizing the performance cost of super sampling in the time dimension.



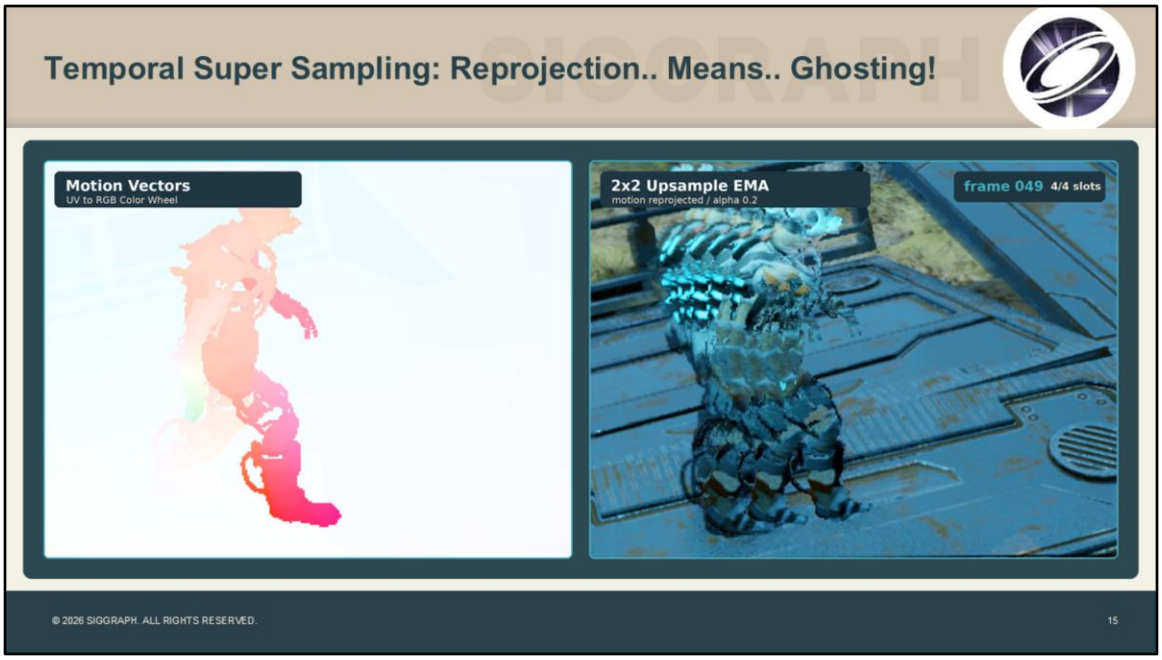
To incorporate spatial upsampling, we can accumulate new samples into a higher resolution frame buffer, to do this, we splat the low-resolution samples into their relevant pixel position in the high-resolution grid, based on the jitter vector.

So, for the example here with 2x2 upsampling, the low-resolution jitter offset falls in only one position in a high-res quad per frame. And we only accumulate a sample into the history for that position.

This still provides the same super sampling, only it now facilitates upscaling too, and we've amortized the cost even more aggressively in the time dimension, as each high-resolution pixel only gets a fresh sample, every 4 frames now.

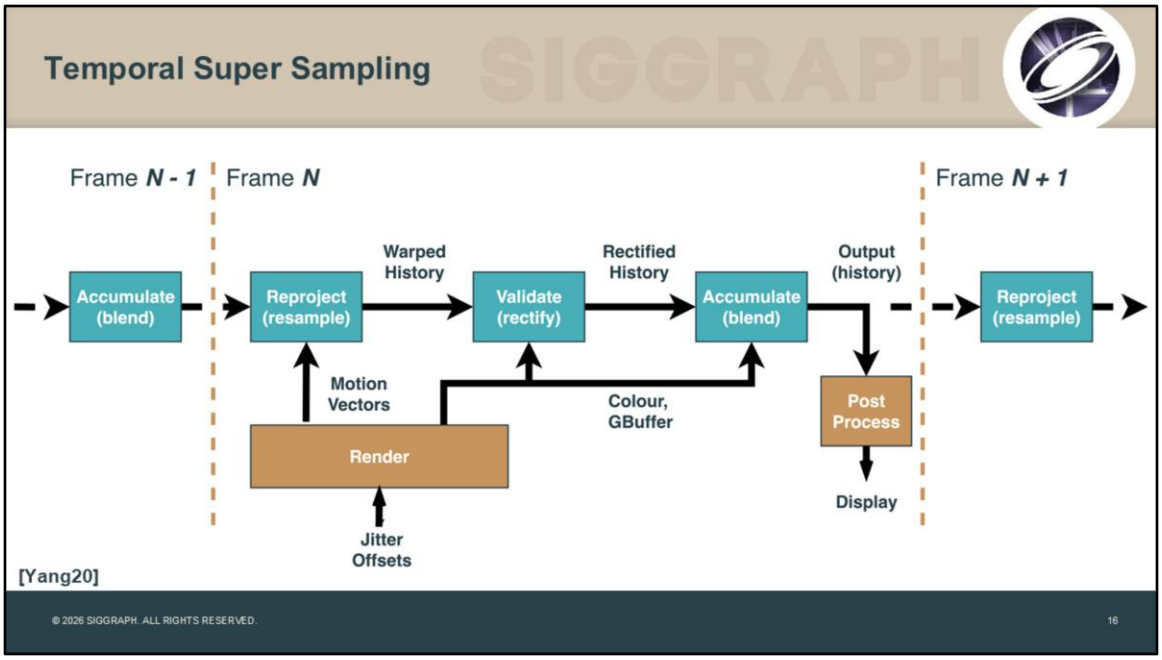


And if you're familiar with this approach, as you might expect, the idea completely breaks under motion.



So, the industry has converged around rendering motion vectors, which allow you to resample the history frame, to get maximum utility from past samples.

The only issue is, reprojection inherently creates disocclusions, and if you don't manage them appropriately, this will manifest as ghosting.

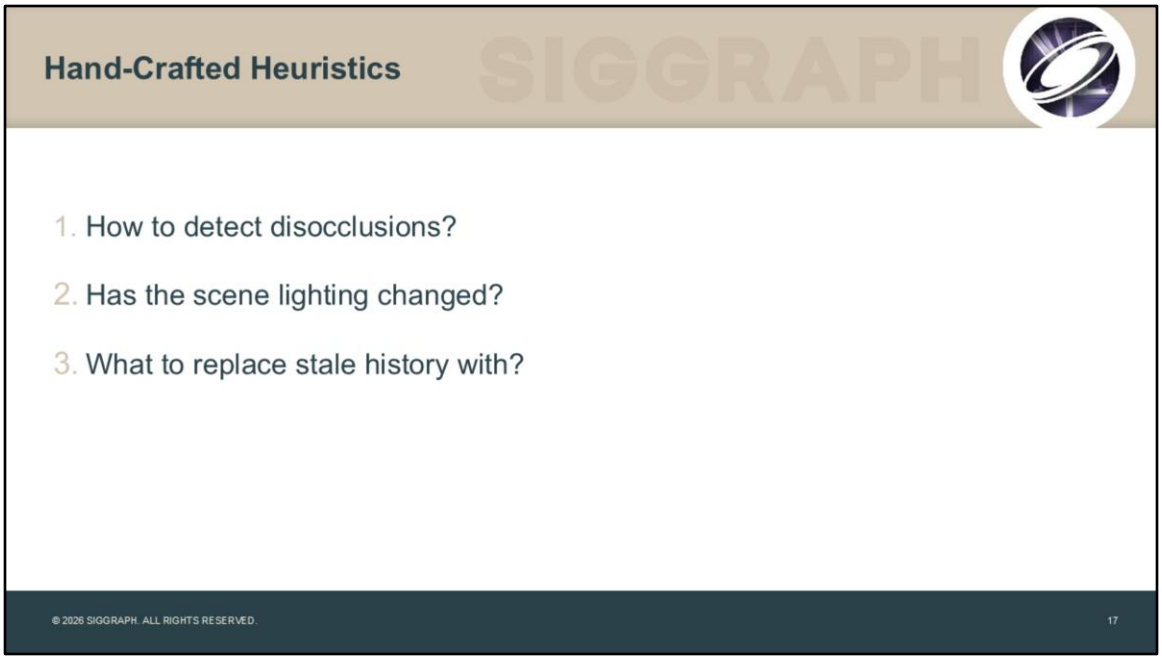


So, traditional Temporal Super Sampling solutions typically break the algorithm down into 3 high-level stages:

- Reprojection, using the rendered motion vectors, to warp the history frame into the current reference point
- Validation, or Rectification, which seeks to invalidate the history in locations where either disocclusions have occurred, or the lighting has changed.
- Finally, accumulation, where we integrate the fresh raw jittered sample, further resolving details and aliasing

In my experience, the challenge in this algorithm lives in the validation stage, where best results come from retaining historic samples for precisely as long as they're valid.

Resetting the history too frequently can manifest as flicker, or bias induced blur. And holding on to the history for longer than needed, can introduce lag into dynamic lighting updates, or ghosting.



Traditional approaches seek to solve these key questions in the validation stage:

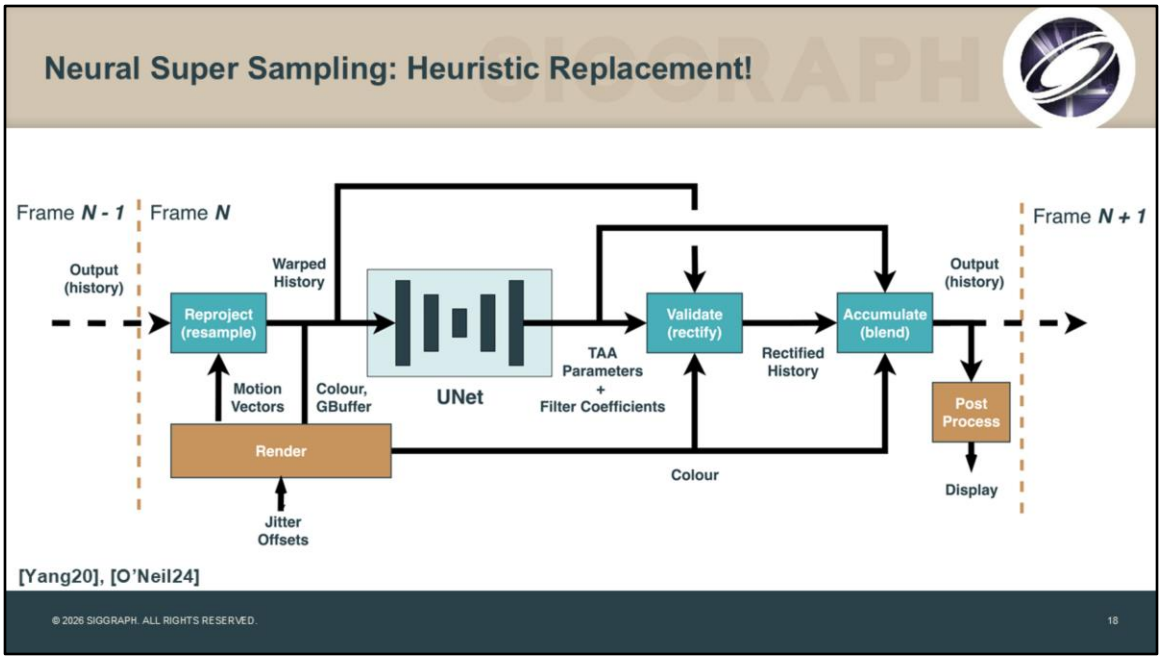
Has a disocclusion occurred? You can typically work this out via geometric test between current and previous world space position.

But geometric tests, don't infer an answer to whether the scene lighting has changed? You could resolve this via constantly clipping the history within an AABB of statistics computed from the currently rendered colour, such as past work in TAA.

But if you're upscaling, the frame you're computing those statistics from, will be low resolution and aliased, and a global clip to AABB can unnecessarily re-introduce those artefacts in the output.

Finally, there's the question of what to replace the history with, when it is truly invalid? This typically involves hand-crafted filters, which don't have global context of the feature they're upscaling.

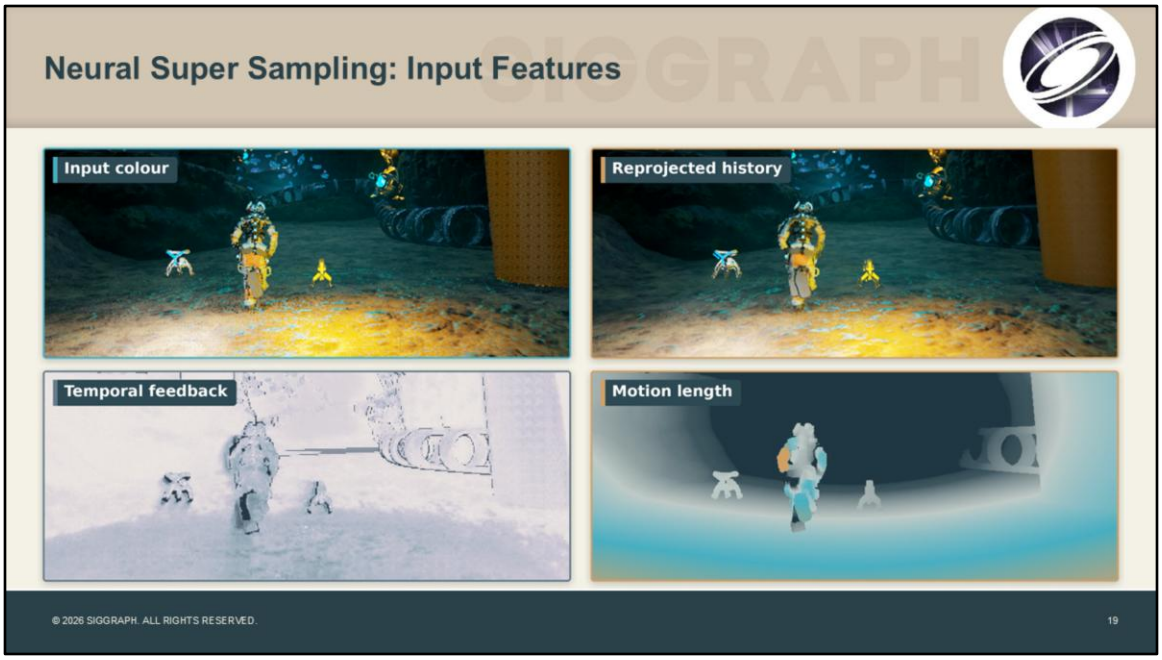
So, the problem is ill-posed, especially when upscaling too, there's many context-dependent heuristic decisions.



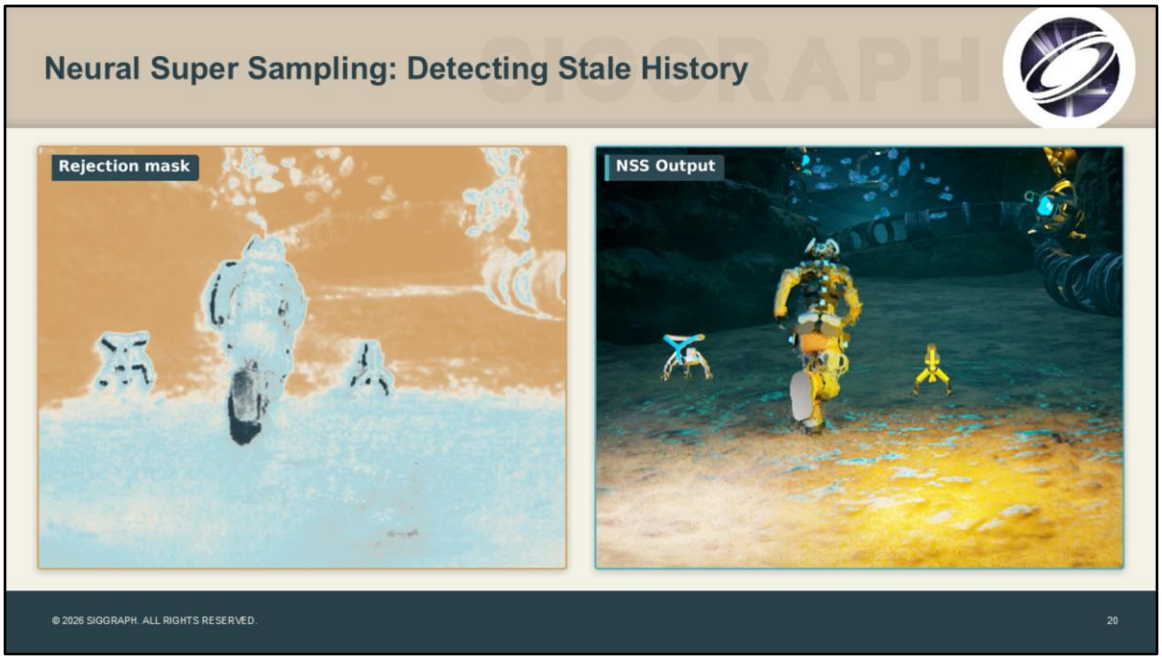
So, in our past work, we presented neural super sampling, framed as heuristic replacement.

This is not direct image prediction, rather it's a network that predicts coefficients that drive the typical screen space image processing passes, you see in regular hand-crafted Temporal Super Sampling.

Its sole purpose is to temporally reconstruct the high-resolution frame, making dynamic decisions on how to best do this, at a per-pixel granularity.



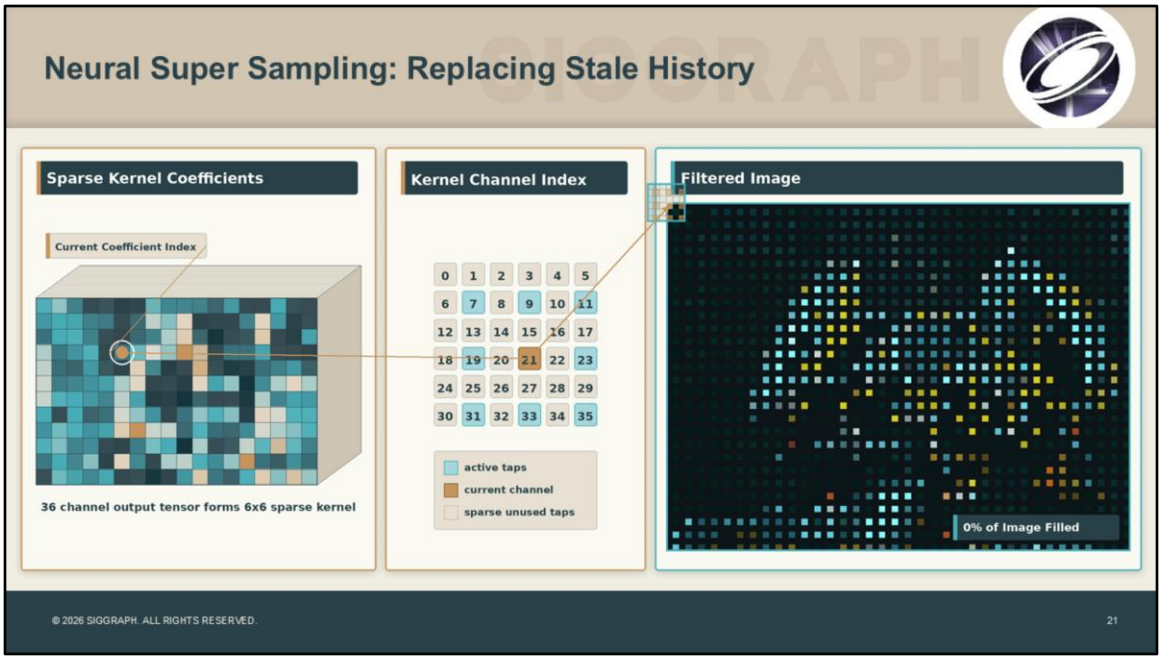
So, to give an example, we provide the network with the current rendered input colour, the reprojected history, some intermediate temporal features, which are reprojected too, and geometric cues, such as the motion length.



And in return, it provides us with a dynamic rejection mask, and we can use this to invalidate the history.

You can see in the figure, this completely replaces history in regions of true disocclusions, such as under the characters arm, when in motion, marked by the dark blue,

Or the network can decide to update the history more conservatively, such as where the moving drone is casting light onto the floor, represented with the lighter blue.



When we need to truly replace the history, we do this with learnt kernel coefficients.

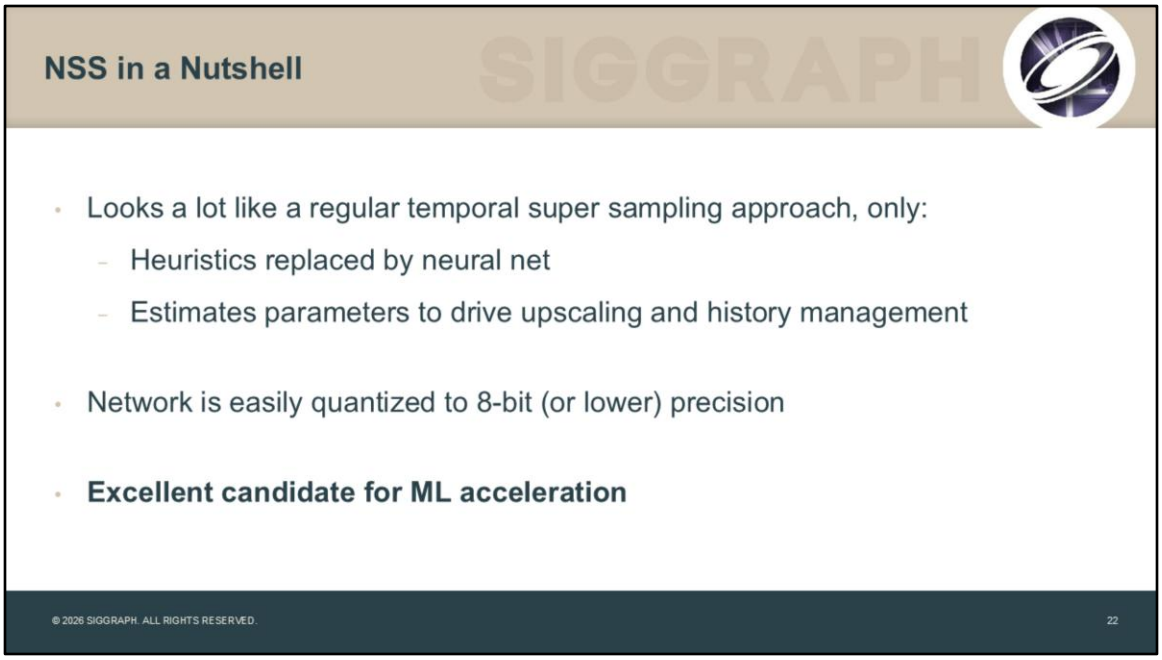
Moving from left to right of this figure, we first estimate a sparse volume of kernel coefficients at low resolution.
Based on the current pixel index, we sample this volume, pulling out the coefficients which are spatially aligned to the pixel we're filtering.

The image we filter, is formed by splatting the low-resolution samples to the high-resolution grid, based on the jitter vector. Similar to the sparse accumulation, I showed a few slides back.

In this example, we are performing 2x2 upsampling, so each quad has one valid sample splatted, its position in the quad dependent on the current jitter vector.
We then apply a 6x6 filter, but as the image is sparse, we can optimize the kernel to drop the empty taps, leaving a 3x3 filter in-place.

This approach allows us to service arbitrary non-integer scale factors and makes

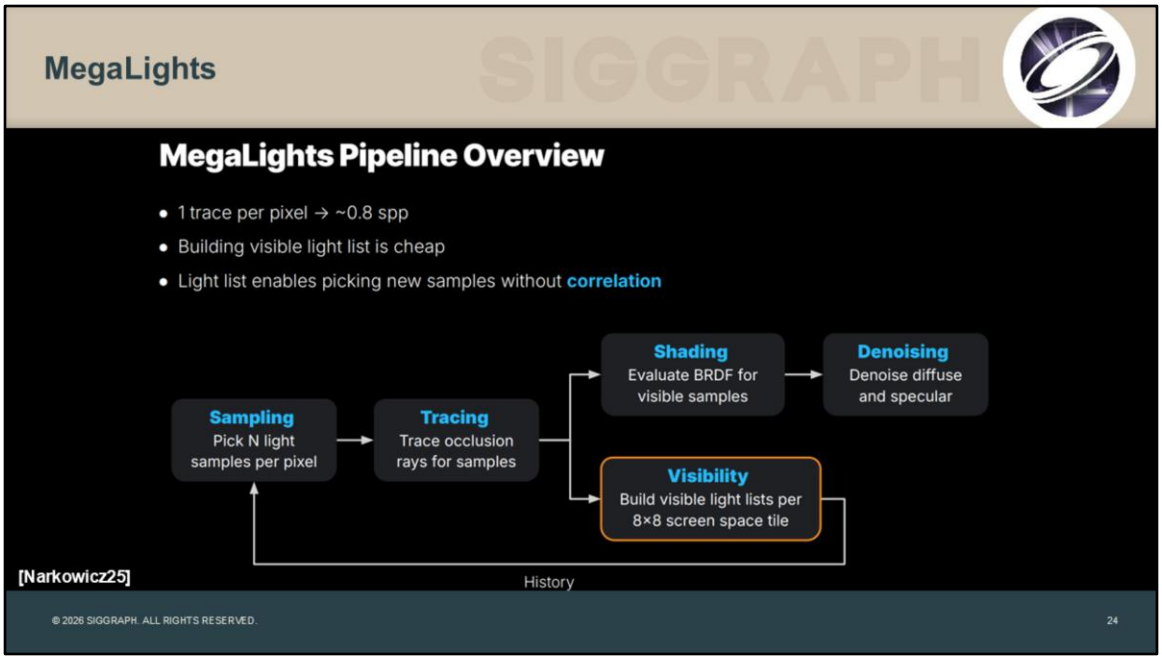
the estimated kernel coefficients implicitly jitter-aware.



So, in a nut-shell:
- We replace hand-crafted heuristics with a neural net
- This drives upscaling and careful management of the history buffer for temporal reconstruction

In practice, the coefficients are much less precision sensitive than if estimating HDR color directly, so the network is trivially quantized to 8-bit precision or lower, and this makes it an excellent candidate for mobile ML acceleration.

So, what did we learn and change when extending this to denoising?



Well, to start with, in neural dawn we are using MegaLights. Which means ray-tracing for direct illumination, and that means noise!

I'm not going to dive into MegaLights, I will instead sign post you to the excellent talk, this slide is taken from, which Krzysztof and Tiago presented in advances last year.

You will notice though, that MegaLights has a spatio-temporal denoiser, so why not use that?

For the readers benefit, here is a high-level summary of the differences between MegaLights and ReSTIR:

ReSTIR:

- First run candidate sampling: pick N light samples per pixel then select the best candidate.
- If selected candidate is not good enough, repeat good samples from

history and neighborhood.

- Candidate sampling and sample reuse add significant cost:
  - Candidate sampling needs to weight each light sample by the BRDF.
  - Need to check visibility for each reused sample. Requires 2-3 traces per spp, can't really trace more than 1 ray on mobile.
- Sample reuse introduces samples correlation. Bad for denoising.

MegaLights:

- Use two lists for each 8x8 pixels to keep track of visible and occluded lights in previous frames. Guide majority of samples towards previously visible lights:
  - Sampling occluded lights less often help reducing variance.
  - Allocate a fixed percentage of sample to previously hidden lights to discover new visible lights.
- Only requires one trace per spp.
- Lights are sampled randomly from a list. Denoiser friendly as it reduces sample correlation.

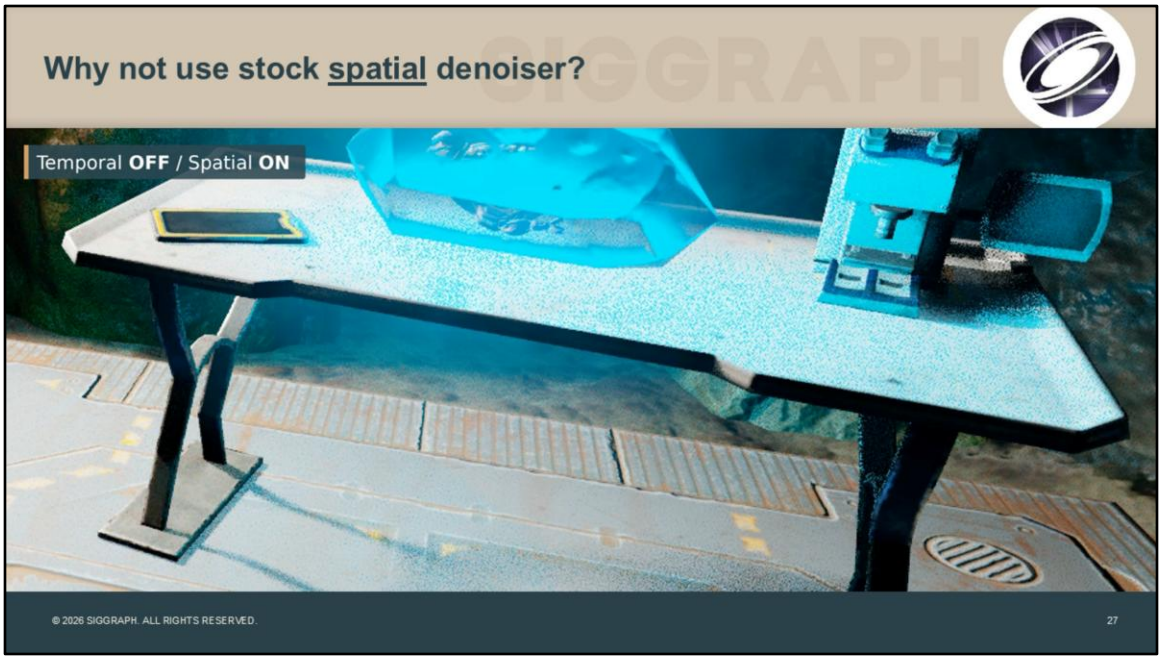
So, to start with the temporal denoiser.

Performing temporal accumulation at low-resolution before upscaling, averages away the jitter, this has 2 knock-on effects:

- Firstly, It breaks the jitter aware upscaling, I was showing previously, as samples are now averaged towards the pixel center, and can't be used to splat into the grid based on jitter, without the jitter, we introduce aliasing in the edges of shadows when upscaling

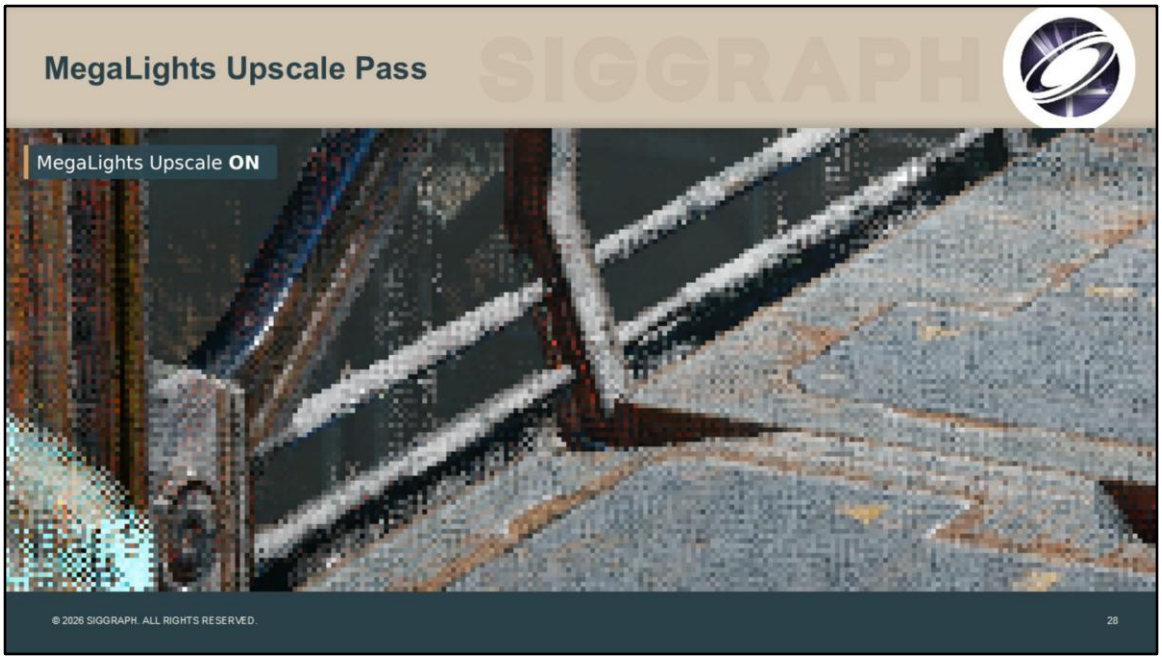
- Secondly, if you average the lighting towards pixel center, and not the albedo, it can cause some pixel misalignment between the jittered albedo and the centered lighting, this can show up as artefacts at geometric boundaries

In this example, which hopefully comes across, I'm attempting to demonstrate aliasing on the boundaries of shadows, caused when the temporal denoiser is switched on, I should note, this issue is accentuated by the low-resolution we cast the rays at on mobile.



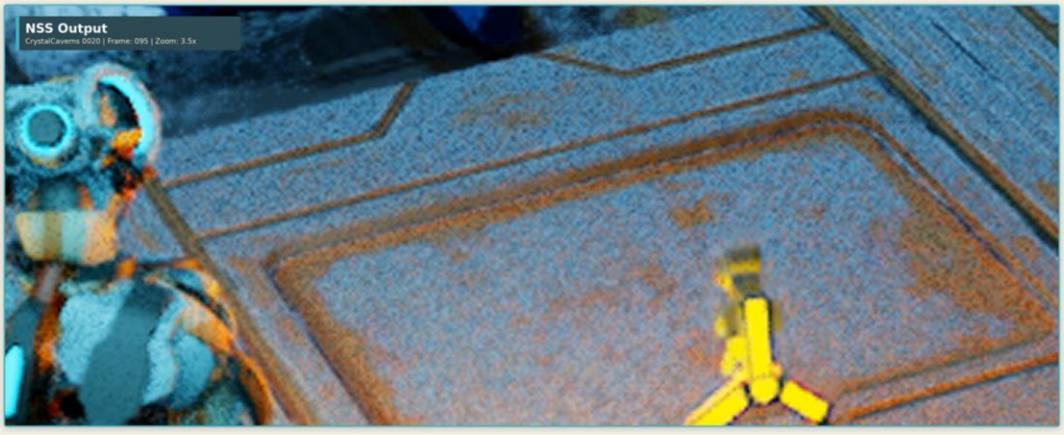
Since we need to disable the temporal denoising, the spatial denoiser alone is not expressive enough to remove the noise, and since we're going to apply a denoiser anyway, we disable this to save the compute.

I'll also note that we prefer applying filtering after accumulation, as otherwise it causes some bias.

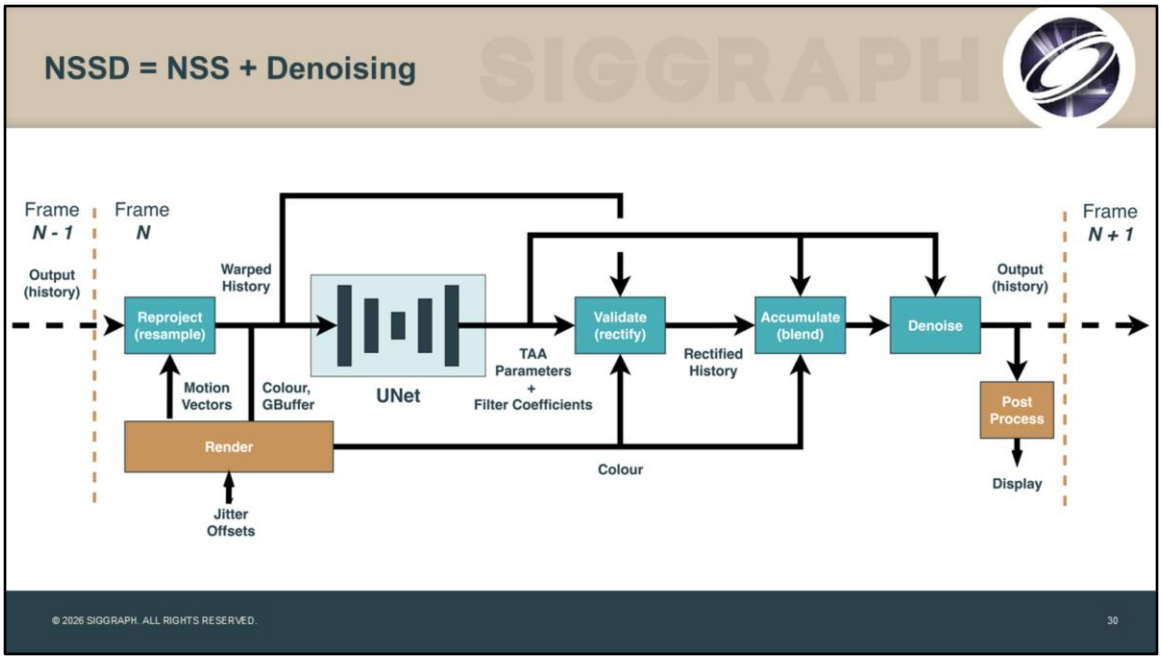


A quick note on a MegaLights specific feature, which is the upscale pass, we disable this also, as the stochastic upsampling it performs appears to cause some artefact patterns when used with our jitter-aware temporal accumulation.

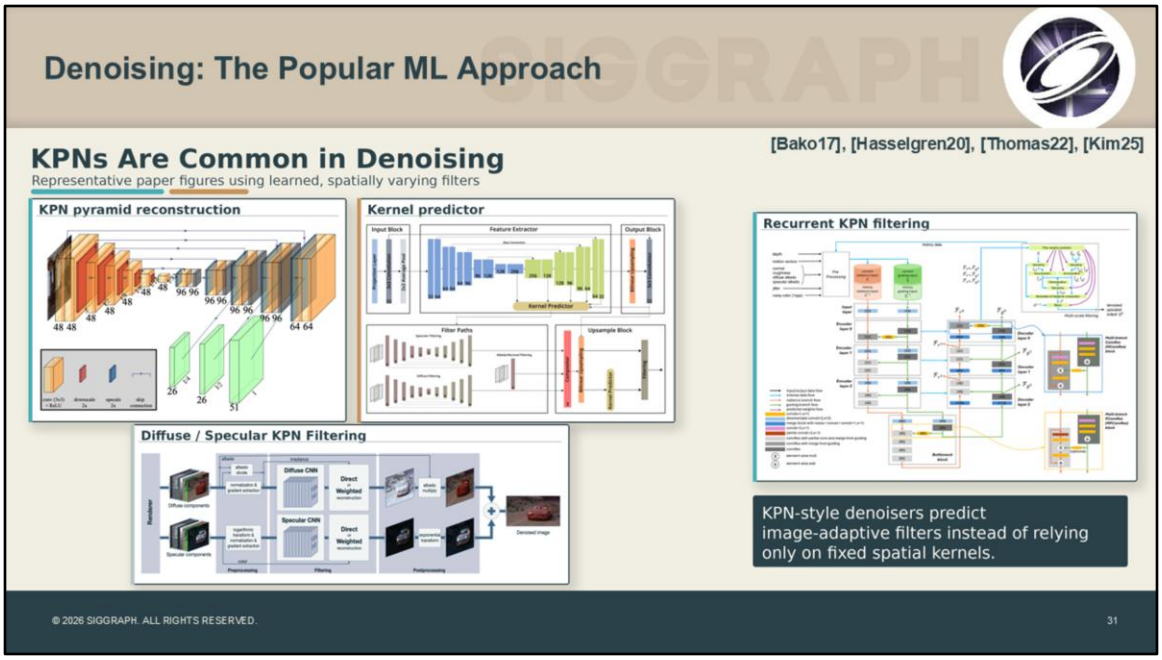
NSS Alone Struggles to Fully Denoise



So, this leaves us with an output like this, in which you can clearly see, the temporal accumulation and light weight filtering used in NSS alone, is not capable enough of cleanly removing all the noise.



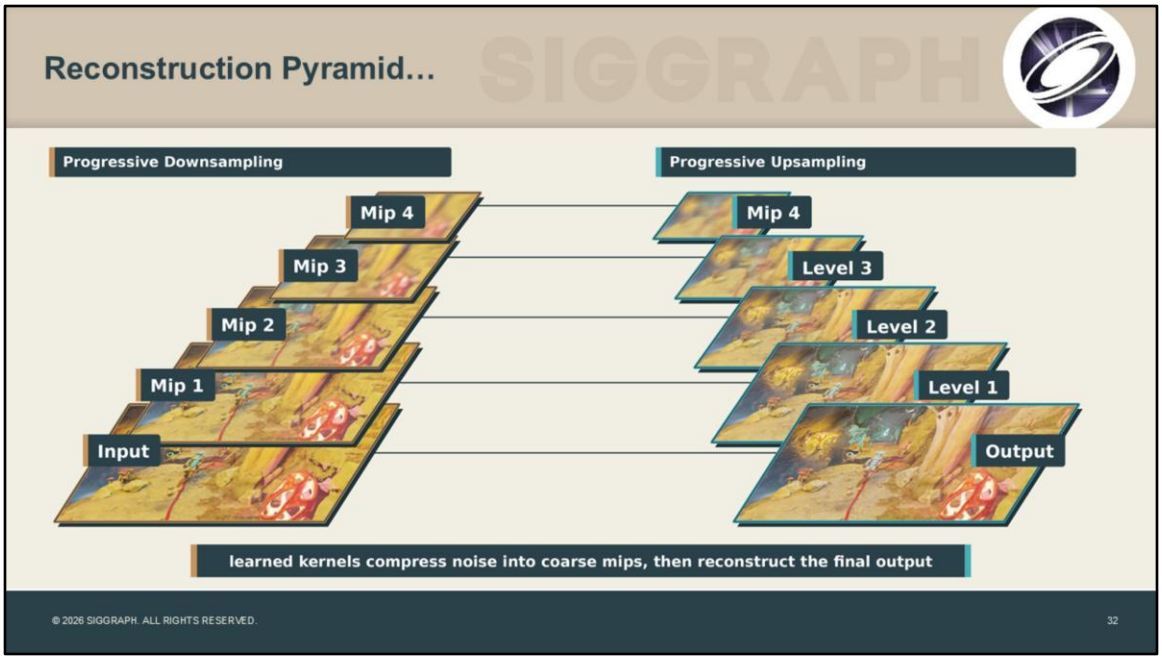
So, high-level our proposed solution looks like this, where we add a denoise stage post-temporal accumulation. And expand our network architecture to predict additional parameters to drive this stage too.



But what should this stage do?

As you might expect, there's a great body of literature on neural denoising approaches.

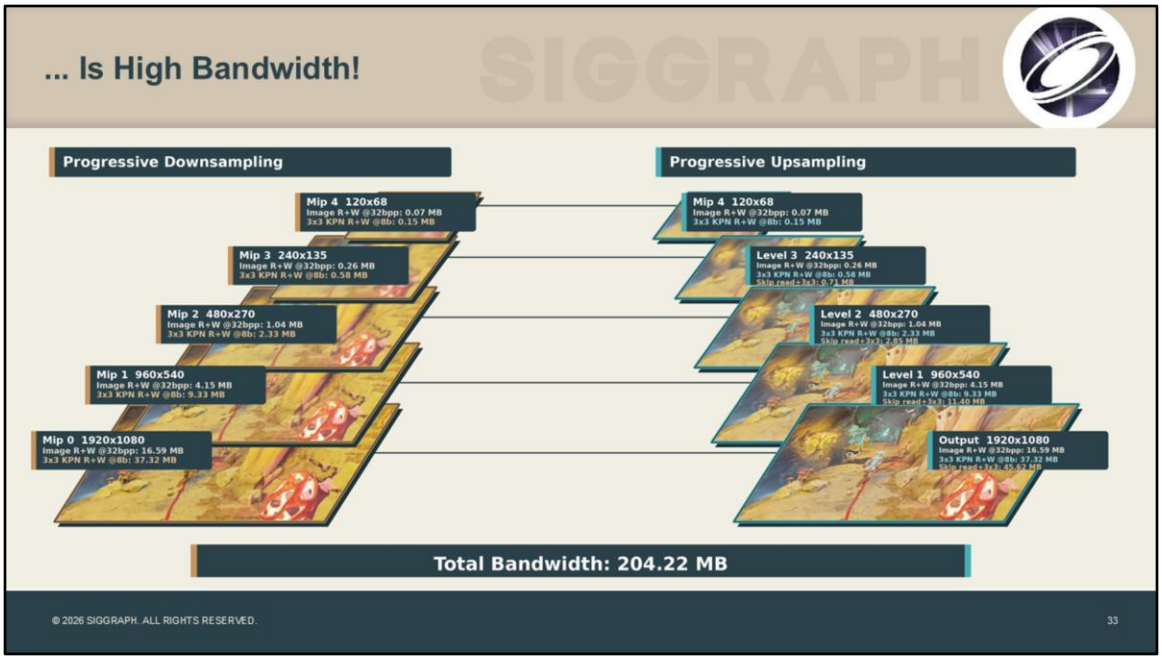
Large kernel prediction networks are a very popular approach, these take inspiration from spatially varying kernels, such as the joint-bilateral approaches used in SVGF, but are learnt, so more expressive, and don't need hand-tuning.



A typical kernel prediction denoiser perhaps look something like this, where you start with an input frame on the left, and progressively downsample into a mip-chain, using predicted kernels from a network.

Once the downsample pyramid is generated, it's common to perform coarse-to-fine reconstruction, upsampling each mip level with learnt coefficients, and re-integrating the high frequency detail from the mip above, via skip connections.

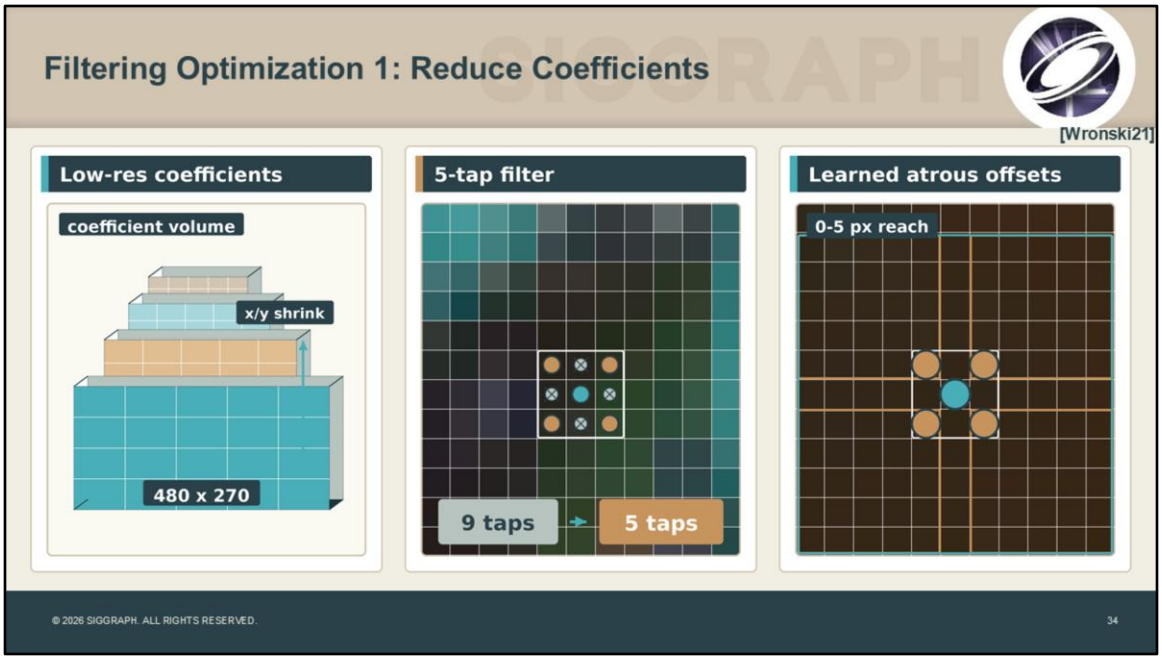
This approach makes sense, and is well adopted, but for a mobile developer, there’s a somewhat terrifying reality…



And that is memory bandwidth...

As Ralph mentioned, DRAM accesses directly correlate to power consumption, so high-bandwidth means faster battery drainage, warmer devices, and less game time.

So, whilst an approach like this worked well for us in theory, we seek to reduce bandwidth usage, with as little quality degradation as possible.



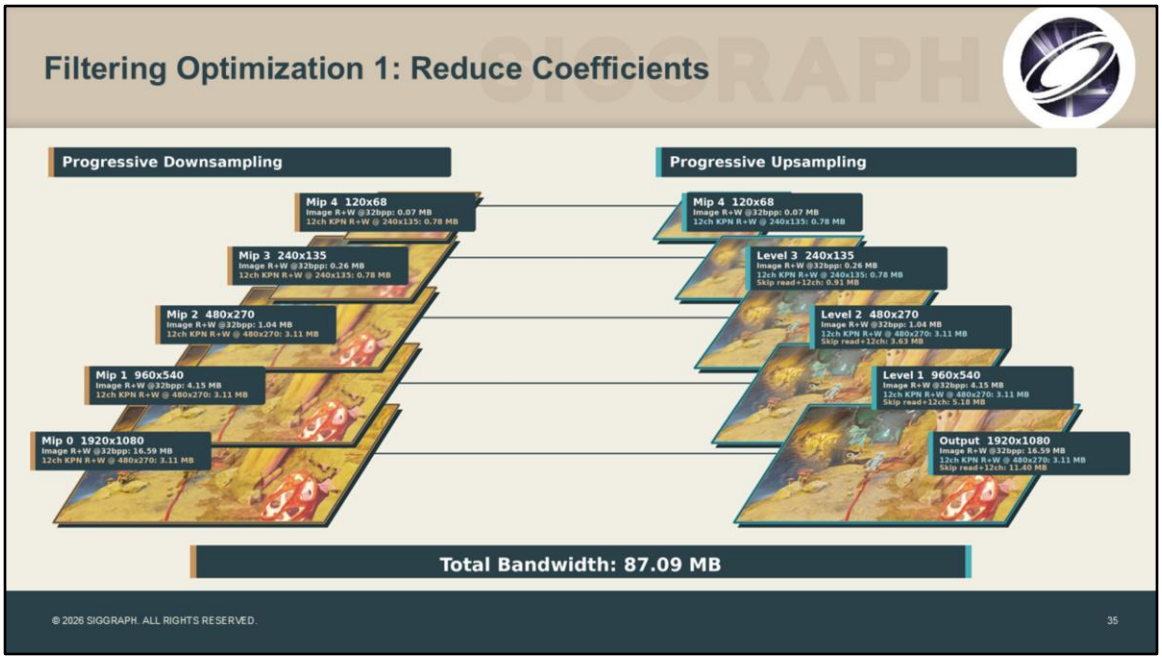
So, our first optimisation of this approach, aimed to reduce this bandwidth by directly reducing the size of the estimated kernels themselves.

We do this in two ways:

1. Starting on the left, quite simply, we reduce the spatial resolution they are estimated at, sharing coefficients across spatially local pixels
2. Secondly, we directly reduce the number of taps, dropping from 9 to 5, which reduces the number of coefficients we estimate, which is the channel dimension of the output tensor

To compensate for the reduction in taps, we estimate a learnt dilation offset, this allows for the network to learn to directly push taps away from their origin, similar to the atrous filters used in SVGF, this gives the network expressivity to move the tap positions on a per-pixel basis, making the most out of the reduced 5-taps it now has available.

In practice, for our use case, we've not found these changes to have much of a noticeable impact on reconstruction quality, in comparison to the starting solution which estimated fixed 9-tap filters at native resolution.

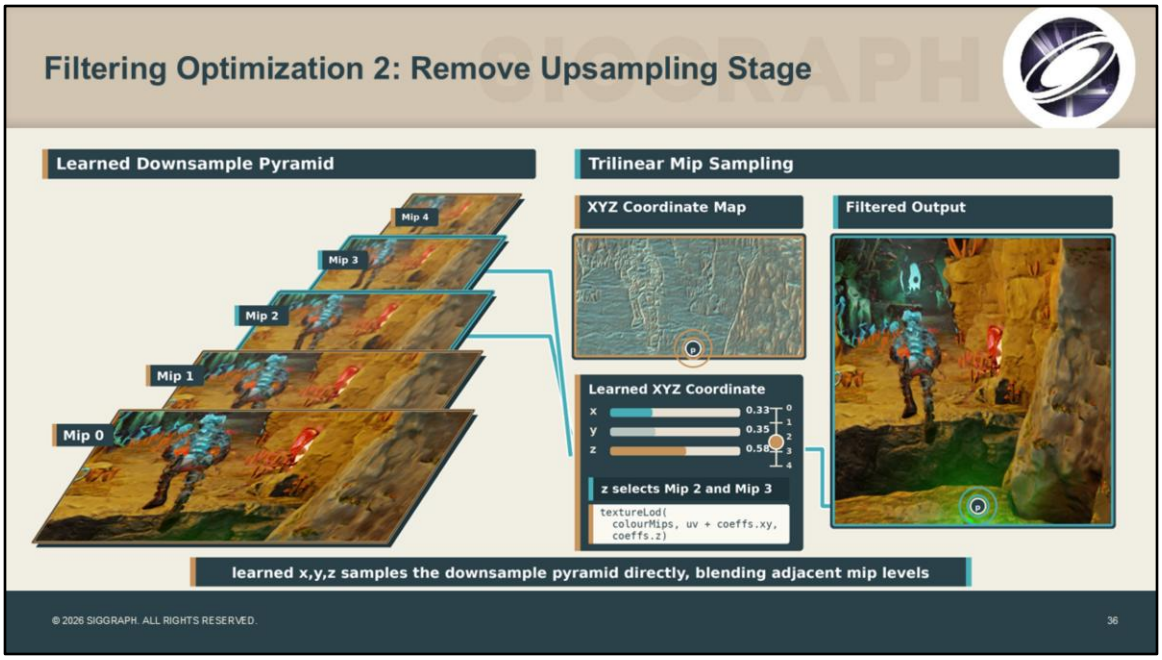


So, as you can see, this provides a good first step forward in bandwidth reduction, but we wanted to go further.

To do this, we exploited an observation we made whilst inspecting the upsample pyramid's contribution to the output frame. We noticed it often collapsed into sparse contributions.

We observed that in instances where a region of the frame is especially noisy, it naturally relies upon lower levels of the pyramid. Mostly ignoring the upper levels. Intuitively, this makes sense, as it is where the image is most heavily filtered and blurred.

And conversely, in the opposite scenario where regions of the frame have perhaps seen more temporal accumulation, so are not that noisy, the upsample pyramid predominantly draws on the higher mip-levels, where there's sharper higher frequency details.



To exploit this observation, we have now moved to an approach where we only generate the downsampled side of the pyramid, and instead of the progressive upsampling, we train the network to directly select exactly which mip-levels it would like to contribute to the final reconstructed output.

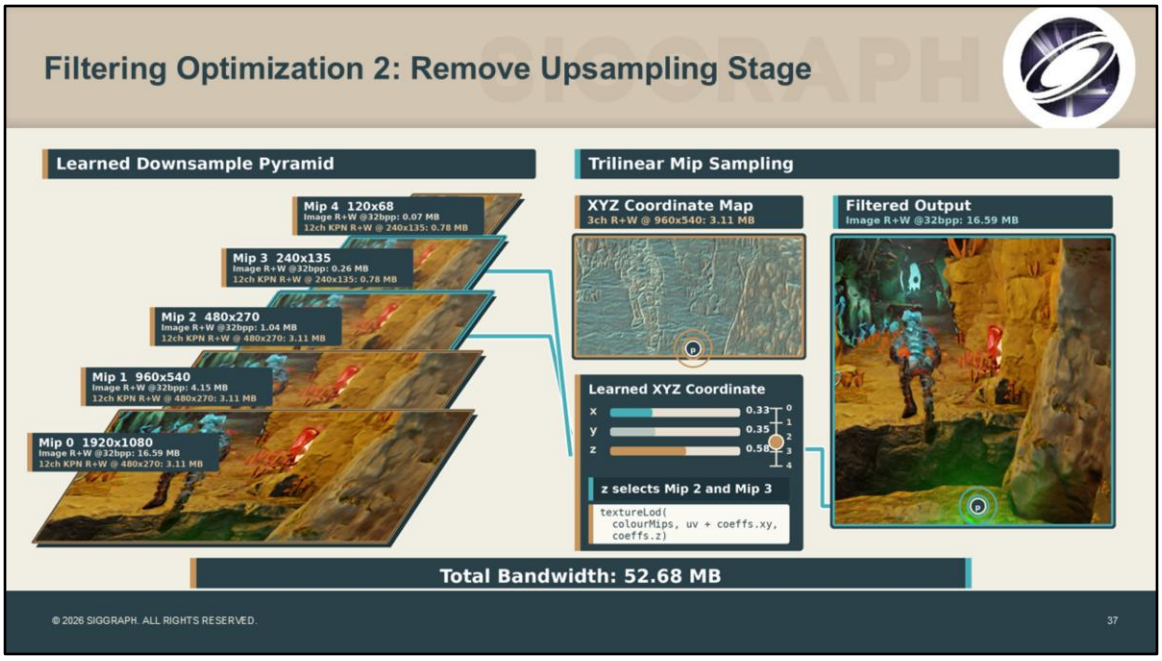
You can think of this as allowing the network to directly drive the trilinear filtering in a typical texture instruction.

In practice, once we have filtered and generated the downsample pyramid, our denoiser uses a single `texture` call per output pixel, where we rely on the hardware accelerated trilinear filtering to perform the denoising.

To do this, we estimate 3 coefficients:
- Z, which selects the mip level to sample from in the learnt downsample pyramid
- And then an XY offset, which we use to bias the uv sampling position, providing a bit of an edge stopping like function to the trilinear filtering

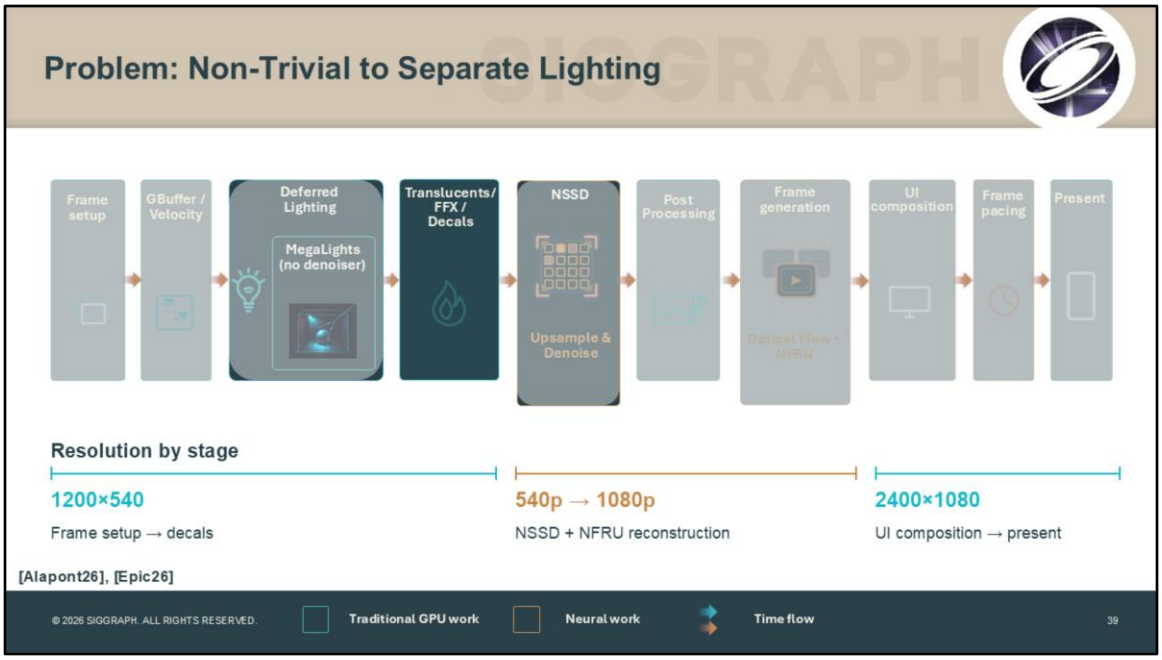
The filtering quality isn't exactly as sharp, but this approach saves precious bandwidth and compute. In practice, we find the trade-off worth it, as it only

looks marginally blurrier in regions where the history has been recently reset, which typically occurs in motion, when it's harder to notice.



The total bandwidth with this approach is much more manageable, and with a cache efficient implementation it looks better than the naïve 50MB.

Finally, on the filtering, we wanted to denoise our lighting separately, to avoid blur of texture details. Which is commonly done in denoising solutions.

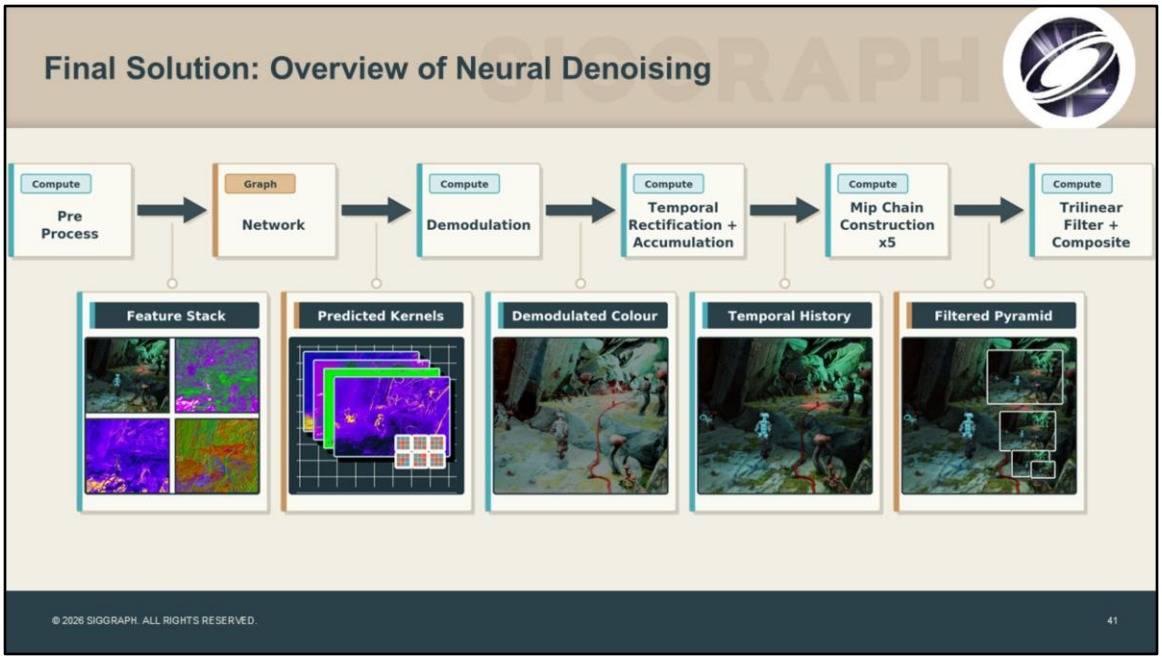


But in our pipeline, we still wanted the dispatches that happen after deferred lighting but before upscaling, like translucents.

And in practice, we found it non-trivial to handle this with separated lighting components.

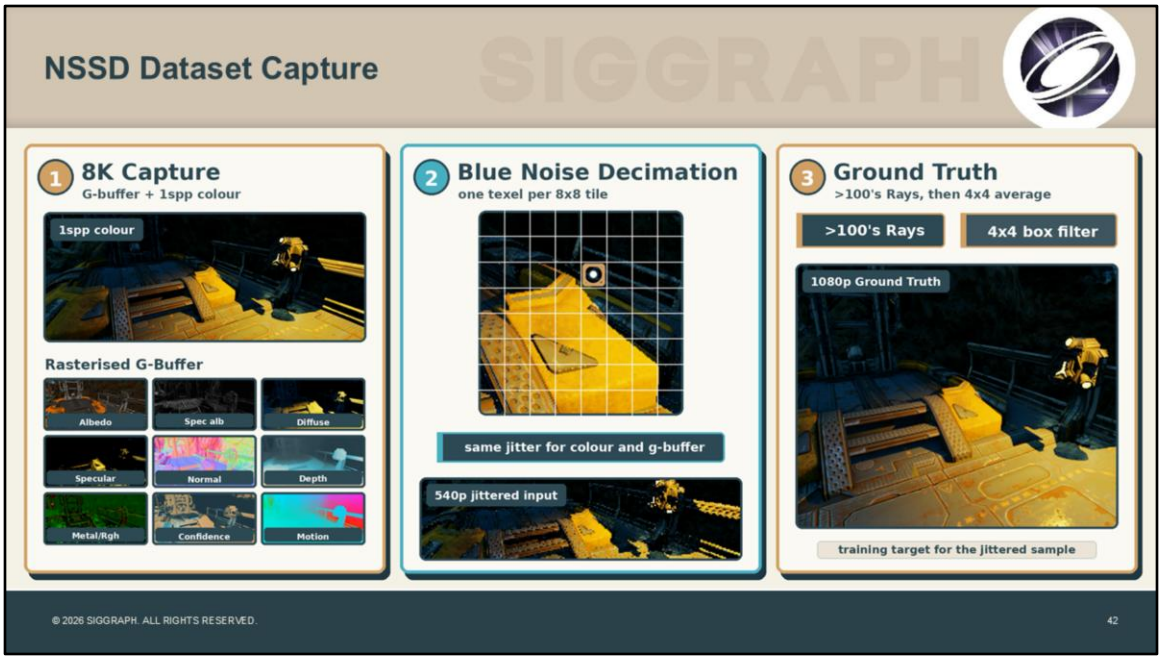
To work around this, we take inspiration from existing work, where we train the network to directly estimate a demodulation coefficient. To separate the lighting from the albedo after these passes.

Similar to prior work, we found this works well, the network seems to just learn what it needs to, and this helps to better preserve high frequency detail.



So, to paint the full implementation picture, moving left to right:

- We start with a pre-process dispatch, that computes and packs the input tensor
- Then we dispatch the Unet architecture as a graph, via our new vulkan extension
- The post-process then breaks down into:
    - Demodulation, where we separate the lighting from the network estimated base colour
    - We super sample both of these components separately
    - Then we compute the learnt downsample pyramid, on the lighting, that we want to denoise
    - Finally, we reconstruct the final output via the trilinear filtering and recomposite the colour



So, just to quickly touch on some details around how we train this network.

We start by generating a dataset, and we do this in a 3-phase approach:

- First, we capture the G-Buffer and 1spp lighting at 8K.
- Second, we simulate jittering by selecting one texel per 8x8 tile of the 8K g-buffer and 1spp color, this synthetically creates our jittered low-resolution assets.
- Finally, we cast 100's of more rays with MegaLights, followed by a 4x4 box filter, to create a nicely anti-aliased, noise free, ground truth.

Whilst we capture everything in the kitchen sink for our dataset, during ablation work to optimize the network, we have found the following inputs to be most beneficial for our use case.

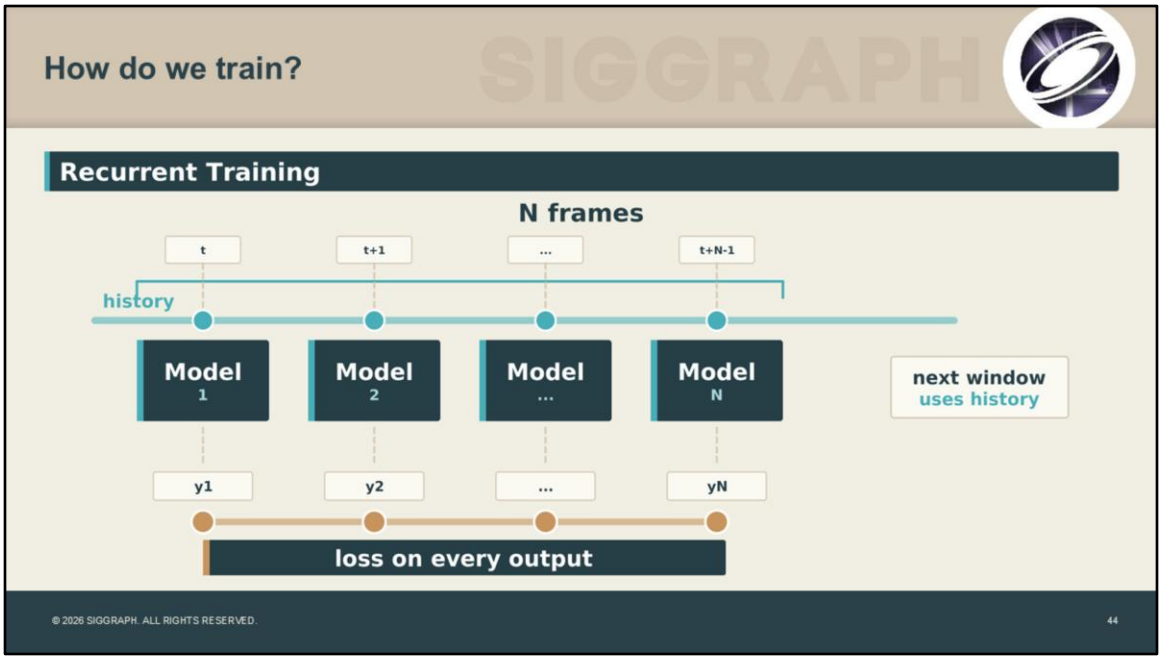
For the current frame, we input:

- current radiance
- the luma components of specular and diffuse albedo
- normals
- and a pre-computed disocclusion mask

From the previous frame, we input:

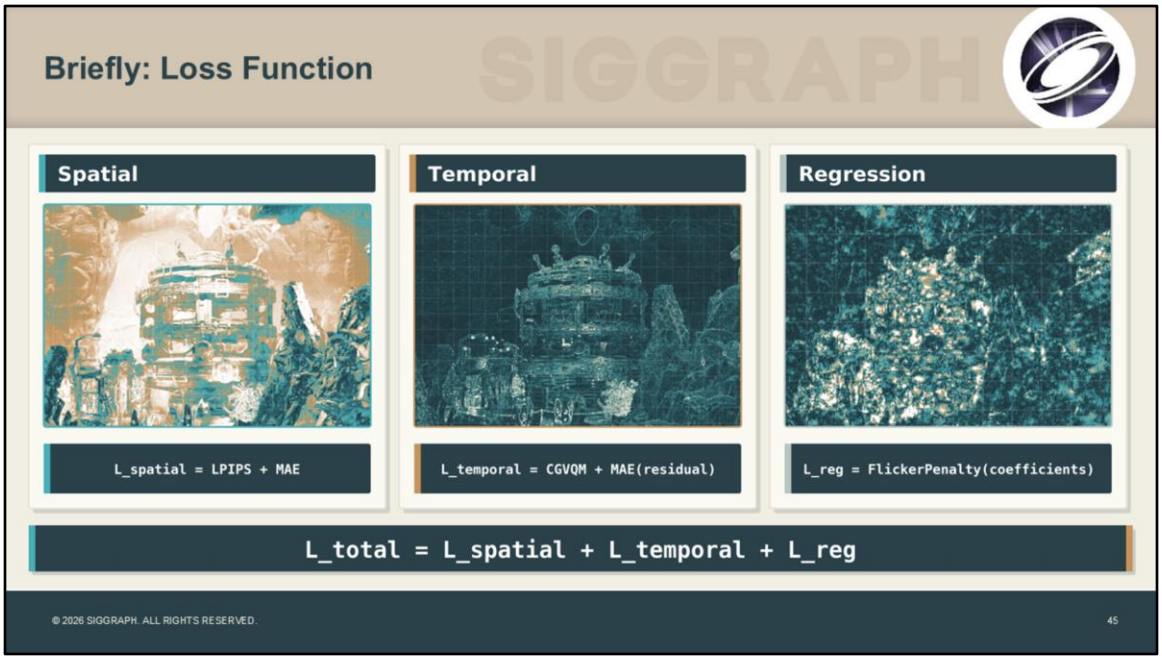
- the warped temporally accumulated history
- the previously denoised output
- temporal feedback features
- motion length

We found inputting the previously denoised output as well as the temporally accumulated color did help, as the former indicates to the network where it needs to denoise, and the latter indicates how it denoised previously, which improves temporal stability.



As for training, we follow standard practice, training the network in a typical recurrent loop.

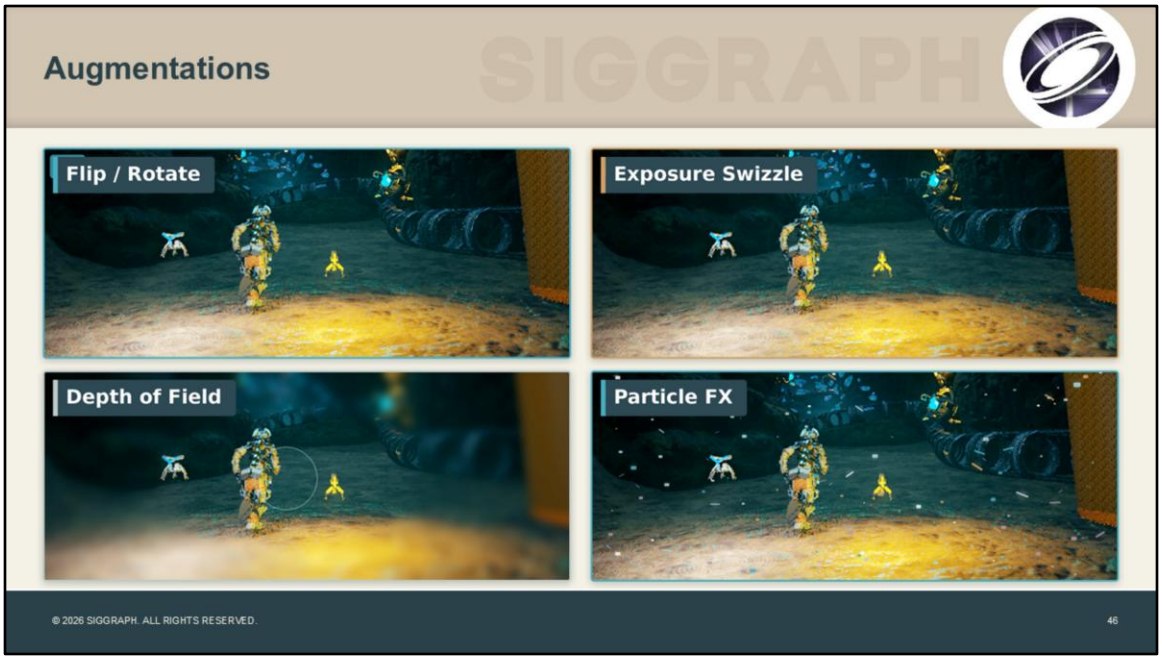
For denoising specifically, we find it's beneficial to maintain the history buffers across subsequent recurrent training steps, as it can take 10's of frames to converge on an accumulated result.



Loss functions really deserve a talk on their own, but loosely, ours breaks down into 3-high level components:

- Spatial
  - Where we compute the error directly on the output frames, encouraging the network to reconstruct pixels as close as possible to ground truth
  - We use some weighting between both typical L1 and perceptual quality metrics like LPIPs
- Temporal
  - Where we compute the error on the residual difference of the current and previous frames
  - Again, we use multiple terms, including perceptual metrics, like CGVQM
- Regression Penalties
  - We find that applying small regression penalties directly to the coefficients the network estimates, can help improve stability, for example penalizing flickering of the dynamic rejection mask

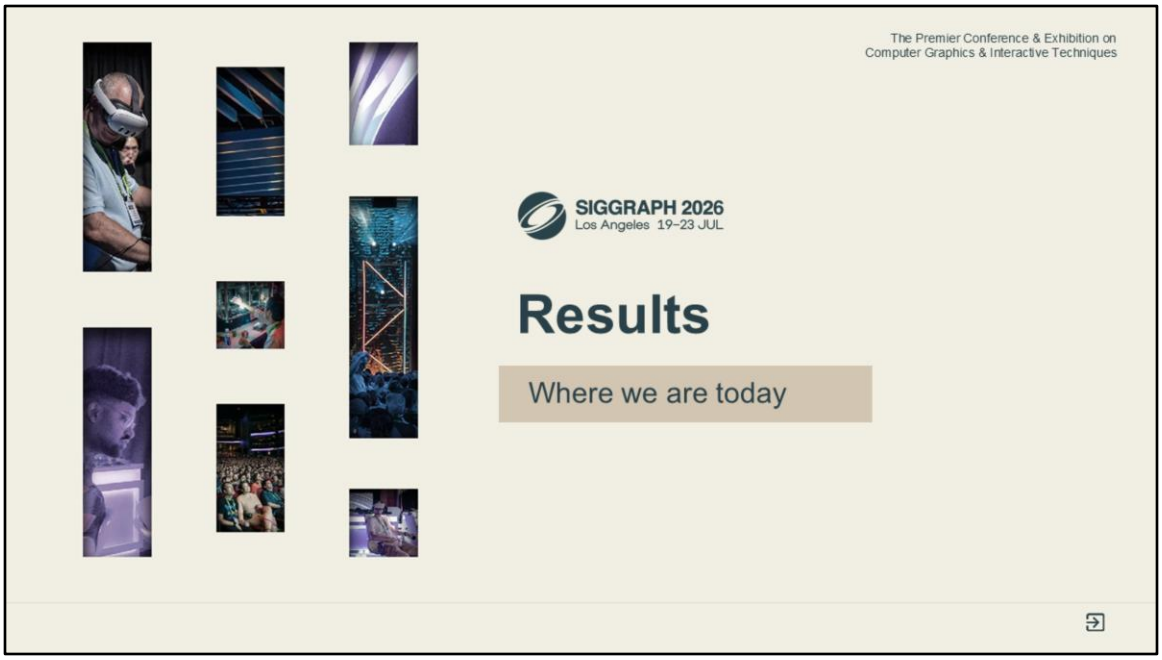
I’ve just plotted some error maps here, I find it useful to visualize the loss, when working on a specific artefact or defect, it’s an intuitive way to get an understanding of whether the loss is addressing the issue you're looking at or not.



As for training augmentations, we do all the typical crop, rotate, flip.

We also:

- randomly vary the image exposure
- randomly apply depth of field, which is used in some cinematics
- stochastically render temporally coherent particle effects in the recurrent training loop.
    - We found this helps improve decision making for content in the scene that is missing motion vectors



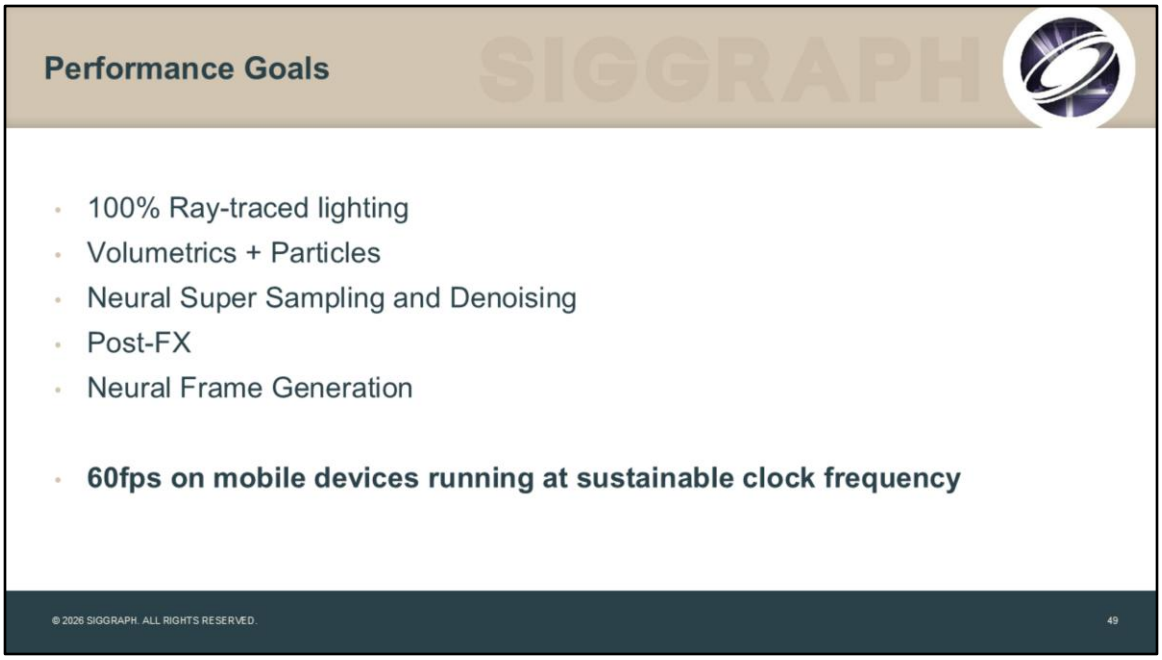
So, just quickly I wanted to show some of the current results

So, I wanted to give an example of the advantages we're seeing with this approach.

Here, we're using the stock MegaLights denoiser + TAAU, which is the default on mobile, on the right. And the output from NSSD on the left, both with the same number of traced rays.

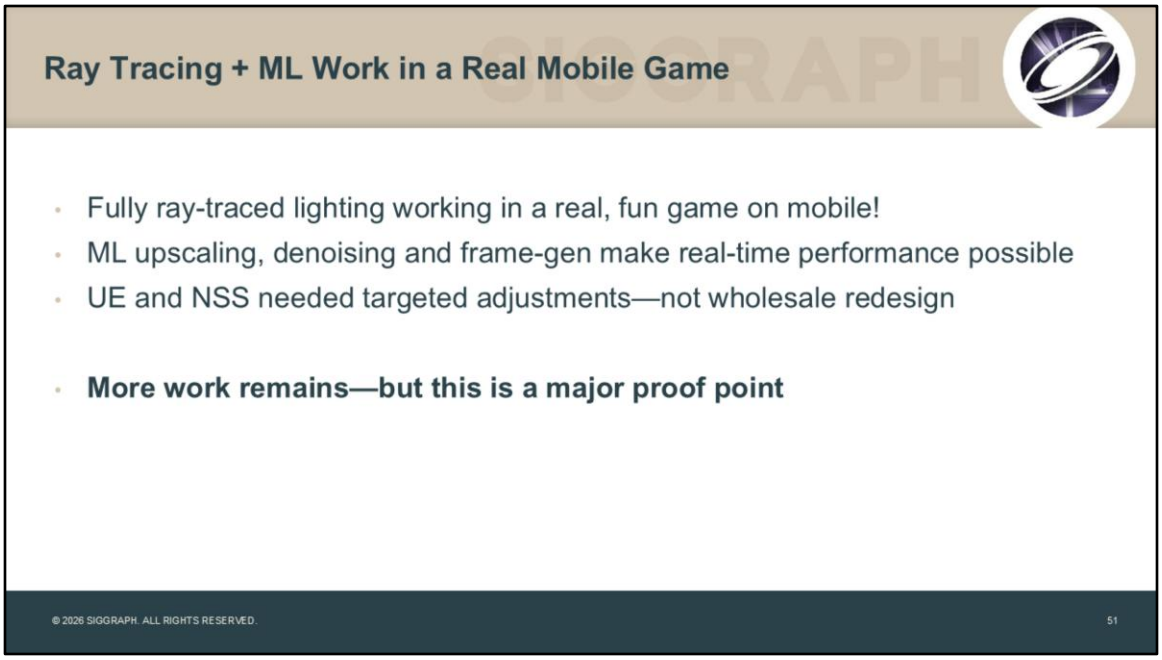
This isn't supposed to be some competitive objective comparison, both sides are still work in progress, our goal here to produce the best quality denoiser and upscaler we could within the frame budget of the game, pinned to last years branch of the technology.

We're balancing rays cast, resolution and denoising cost as holistic objective.



For similar work in progress reasons, I'm not in a position today to talk about performance, I can say the goal is to hit a stable 60-fps, on a thermally constrained clock frequency, after frame generation.

So just to close out and mention what's next



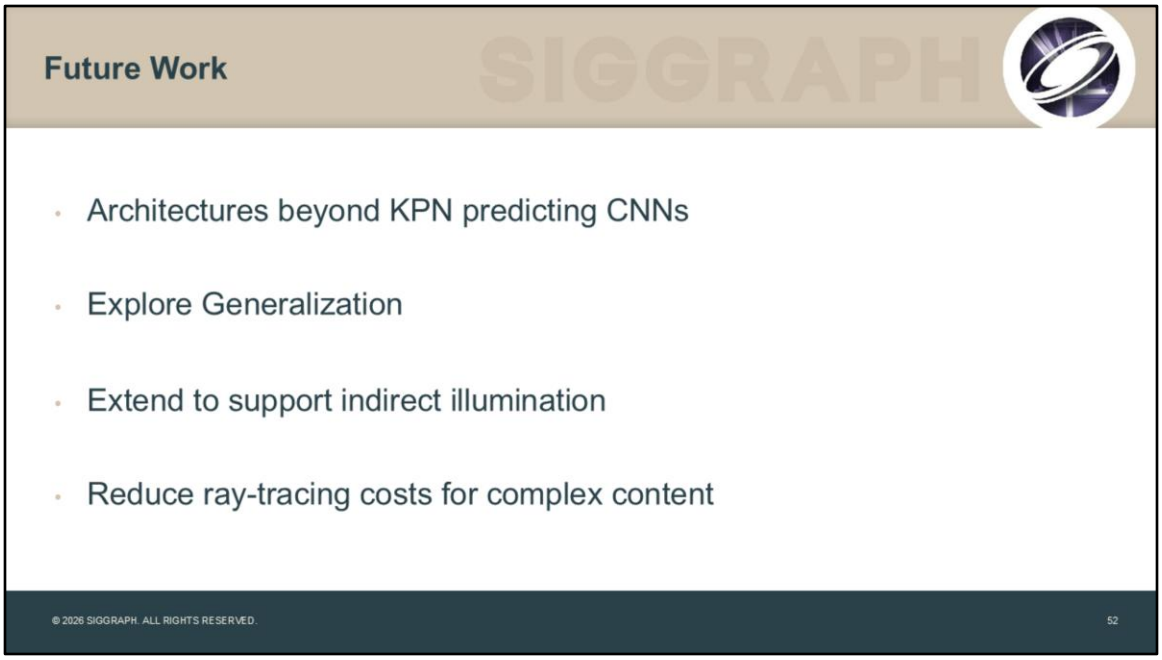
This project has been really fun to work on, I hope you can see that.

Ray-traced lighting, ML upscaling, denoising, and frame-gen, in an actual game we can play on a mobile at 60-fps is something previously only possible on desktop.

And this is really born out of all the latest advances in the ray-tracing tech from Epic, supported by similar advancements in ML and the capacity to accelerate it.

This has all come together to make a title like this possible for the first time.

There's more work to do, but this serves as a great proof point.



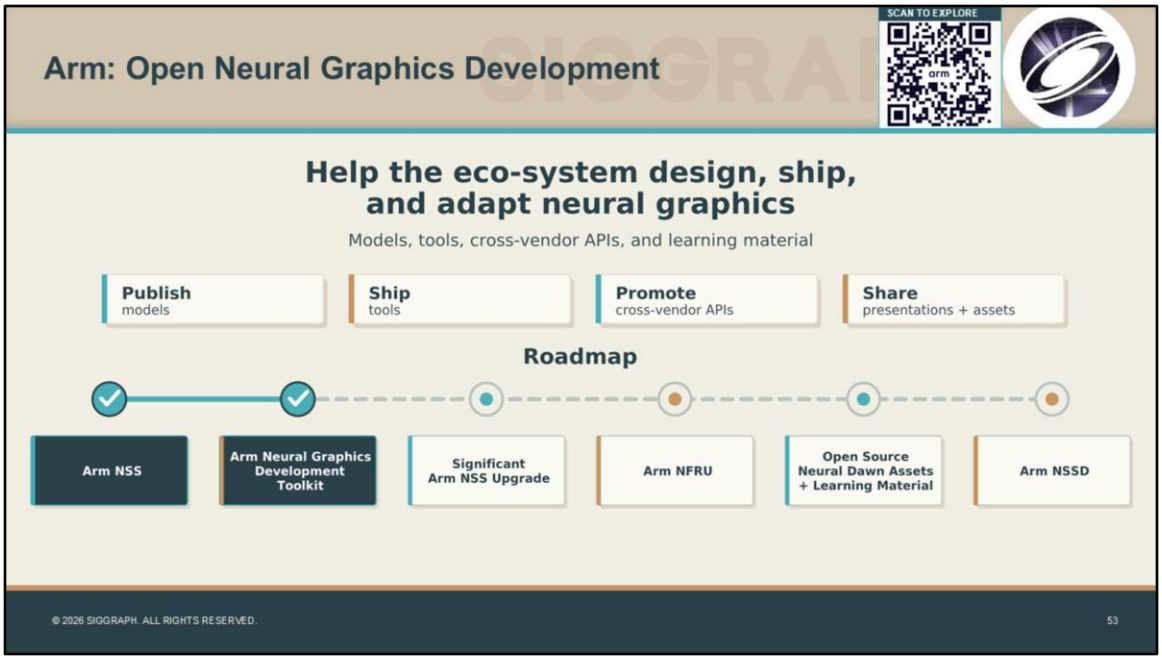
So, on future directions:

- We're interested in exploring architectures beyond KPN predicting CNNs, they're working well now, but we're interested in how we could push the ML further, scaling to heavier network architectures, as the amount of available ML compute scales to meet the demands of adjacent fields.

- We also need to do some ablation work on generalization, since we've been so focused on Neural Dawn using MegaLights, we've not yet had an opportunity to test how the denoiser works when moved to other ray-tracing algorithms, with different noise profiles, such as ReSTIR.

- Of course, a natural extension of that would be to expand the solution for denoising indirect lighting too. That's on the bucket list.

- And lastly, we've had to build the title around avoiding certain things which are more expensive for ray-tracing, such as alpha masked geometry, further technical advancements in this direction, would be of benefit to this work too.



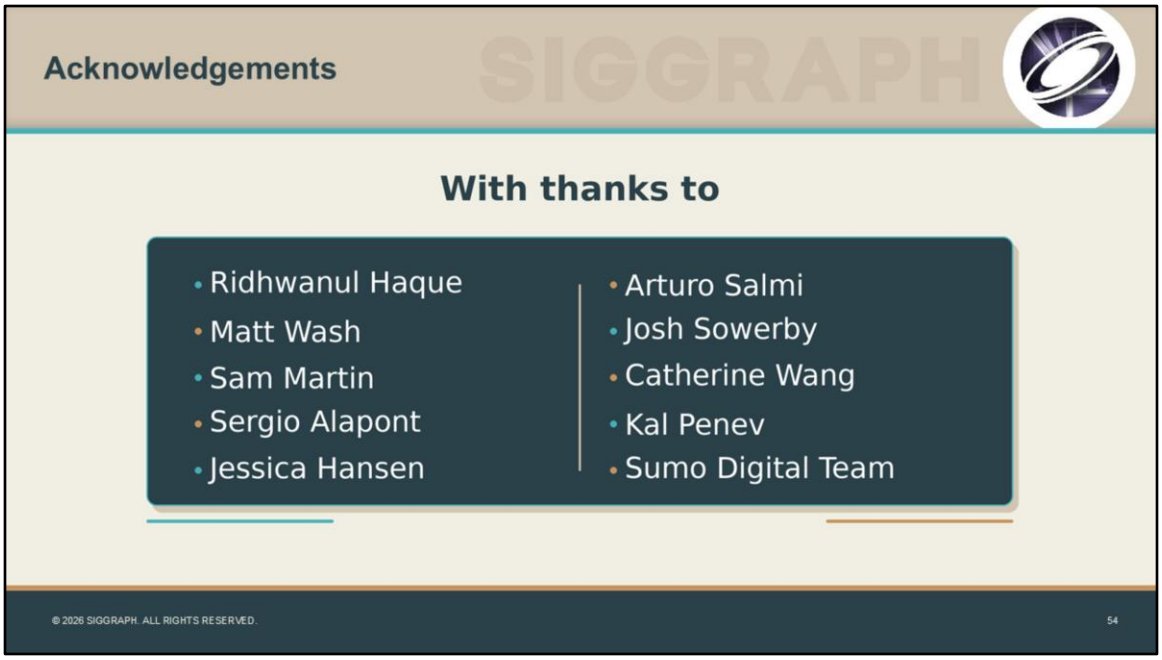
So, just one final note from me, on open neural graphics development, Arm wants to enable the graphics community to do this work too.

We've already published our first version of NSS to hugging face, this includes the Vulkan implementation, training code, and our model weights.

We’re shipping tools to make it easier for developers to build on this too, such as the unreal engine plugin we've developed to capture our training data.

We'll be publishing an upgrade of NSS soon, then our frame-gen technology, and we’re planning to open source the assets from neural dawn, with learning material for training with it.

After some of that ablation work, I mentioned previously, we’ll then look to publish this denoiser.



And of course, a project like this doesn’t happen without all the brilliant people who worked on it.

Special thanks to Ridhwanul and Arturo, who’ve worked tirelessly on what's been presented today.

And with that, thank you very much for listening, I'll stick around to to take questions after the next talk!

Probably will fall short on time to show this, but this is the neural dawn trailer.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

SIGGRAPH 2026
Los Angeles 19–23 JUL

References

With Links

References: Bibliography

REFERENCE	TITLE	SOURCE
[Martin25]	How GPUs Must Adapt in the Age of Machine Learning and AI	YouTube video
[Arm26]	Arm delivers a step-change in mobile gaming with Neural Dawn	Arm developer hub
[Alapont26]	Neural graphics in practice	Arm news room
[Epic26]	"MegaLights in Unreal Engine," Unreal Engine 5.8 Documentation, Epic Developer Community.	Epic docs
[O'Neil24]	MOBILE NEURAL SUPER SAMPLING	Arm community PDF
[Yang20]	A Survey of Temporal Antialiasing Techniques	Eurographics DL
[Narkowicz25]	MEGALIGHTS: STOCHASTIC DIRECT LIGHTING IN UNREAL ENGINE 5	Advances in RT
[Kim25]	Neural Supersampling and Denoising for Real-time Path Tracing	GPUOpen
[Thomas22]	Temporally Stable Real-Time Joint Neural Denoising and Supersampling	ACM DL
[Hasselgren20]	Neural Temporal Adaptive Sampling and Denoising	NVIDIA Research PDF
[Bako17]	Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings	Disney Research
[Wronski21]	Procedural Kernel Networks	arXiv

Licenses: Content

- Neural Dawn | 光影新生
  - Arm Ltd.,

ಧನ್ಯವಾದಗಳು
Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank you
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు