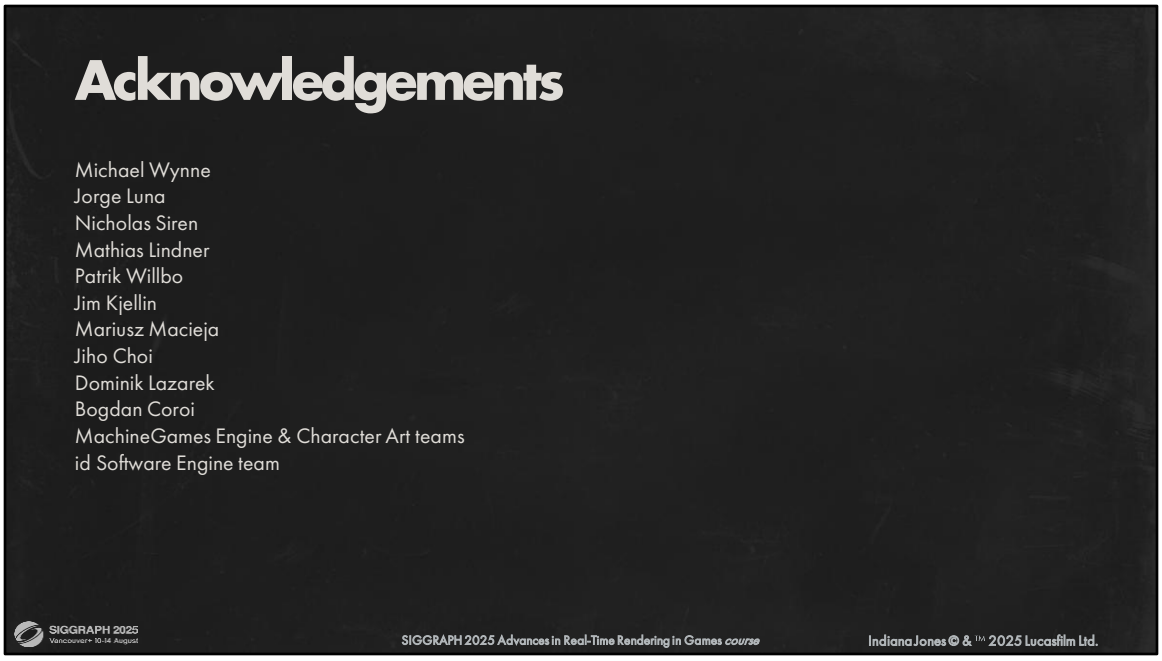
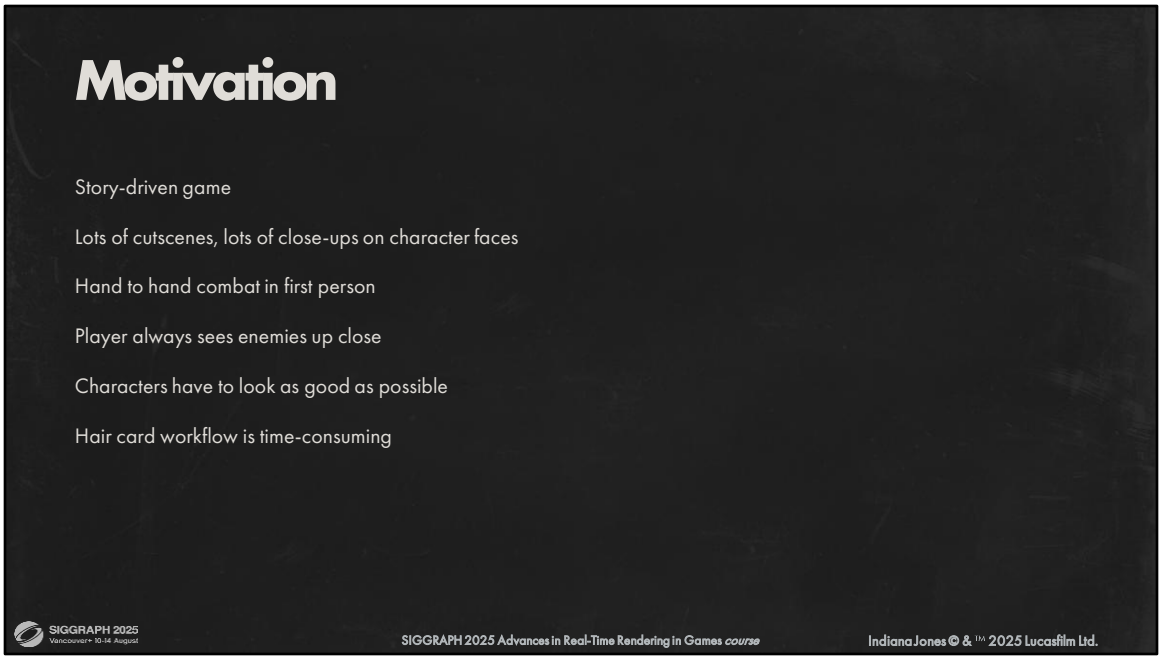

Hello, I'm Sergei Kulikov from Machine Games and this talk is about strand-based hair in Indiana jones and the Great Circle

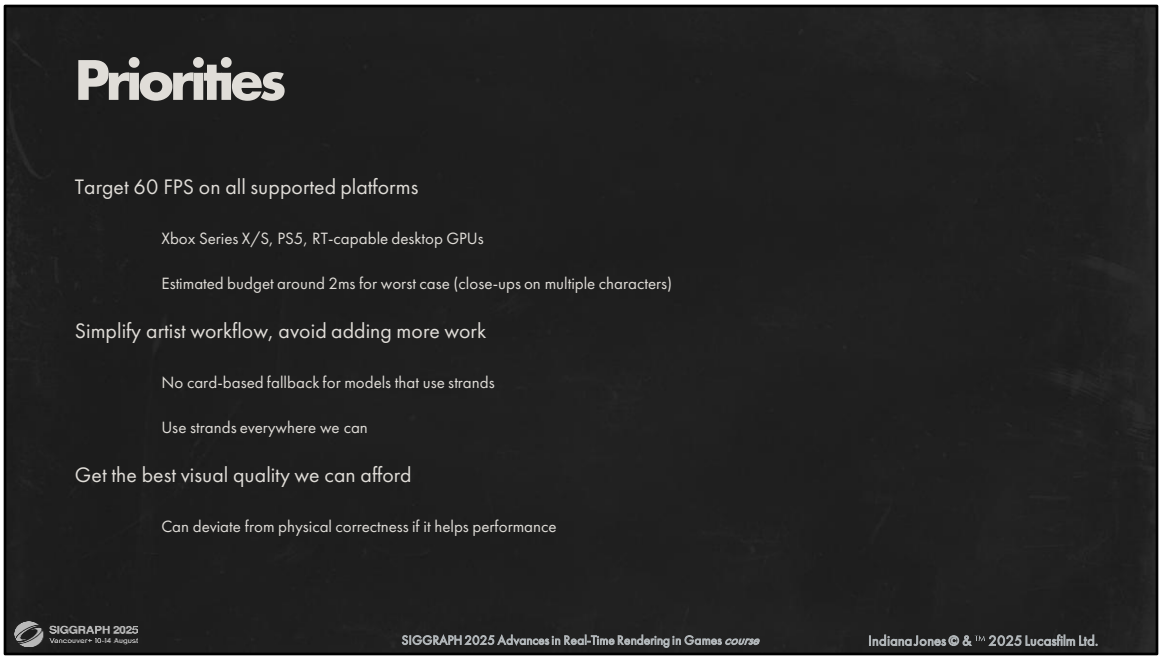

Here is our agenda for the next 40 minutes:
- First, we will look at what goals and requirements we had when designing strand hair. I will give a brief overview of the system we ended up with, and how its components fit together.
- Next we will dive deep into 3 large parts of our hair system:
Rasterization, Shading and Composition.
We will cover algorithms and performance optimizations we did for those systems to be able to run at 60 fps.
Unfortunately, we don't have enough time to cover hair simulation and hair path tracing today, so we will focus on the parts that I mentioned.



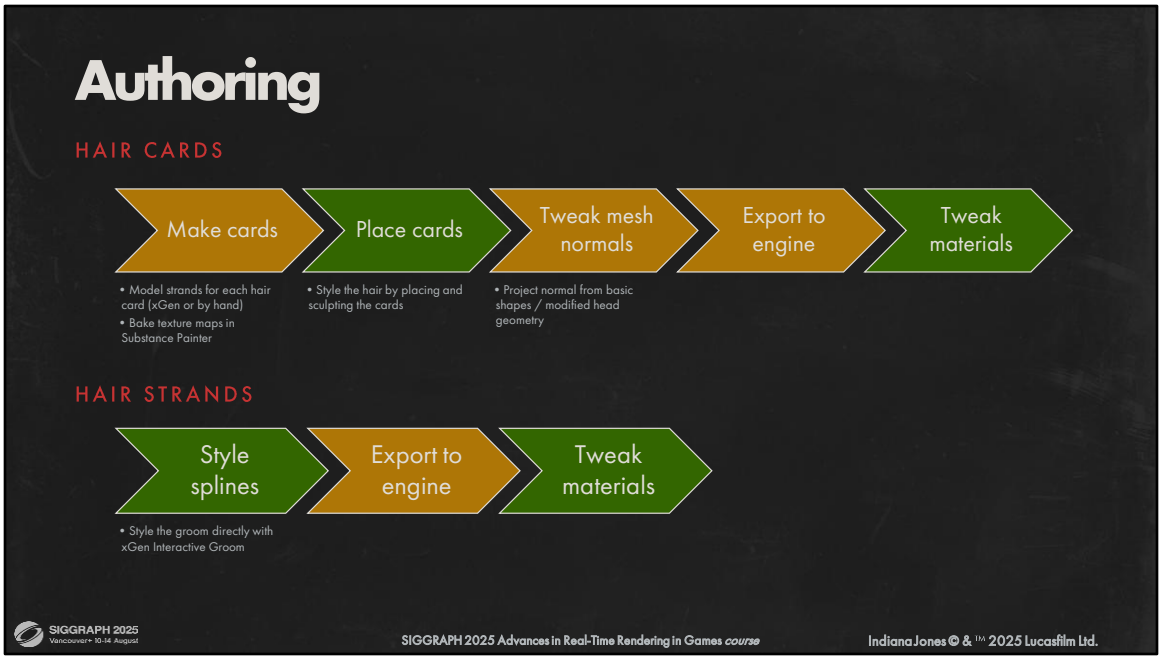

Before we begin, I want to stress that strand hair system was a large collaborative effort throughout the project development, and I want to say huge “Thank you!” to everyone
who contributed to prototyping, developing and shipping strand hair, our and idSoftware engine team for the beautiful engine that served as a foundation of our work, and of course to our wonderful character art team for their invaluable input and the great content they have created with the engine.
I also want to specifically thank Michael Wynne for setting us on the strand hair path and making the first working implementation of the system, and Jorge Luna for his great work on hair simulation.



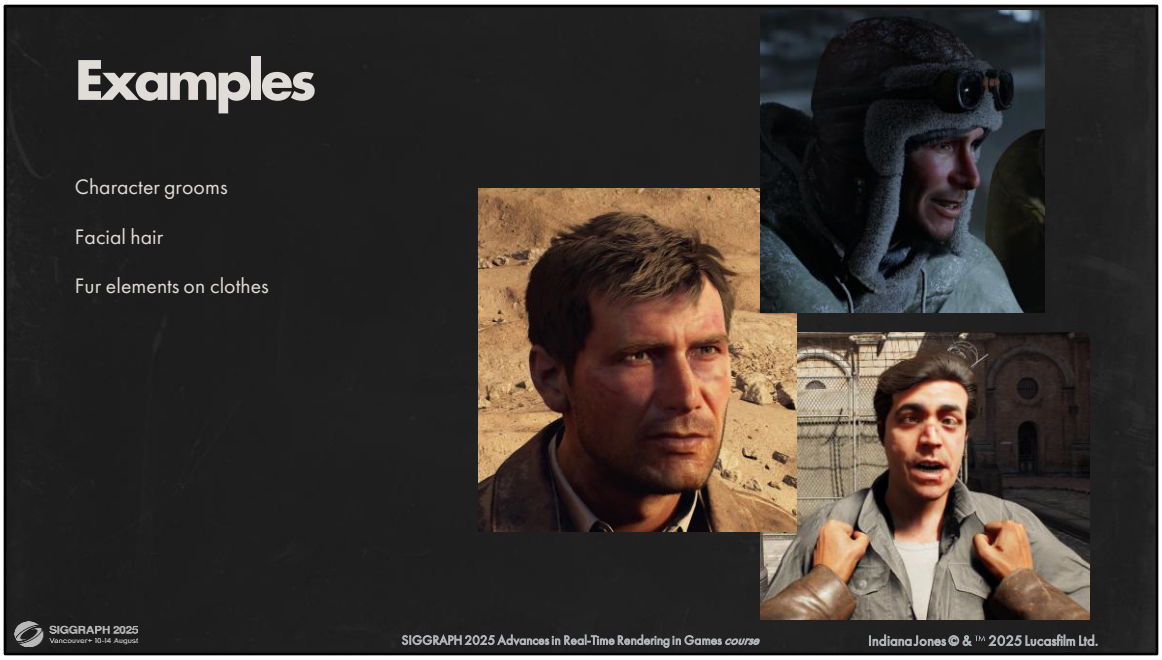

Ok, lets begin. Why did we decide to focus on strand hair for the game? Indiana Jones and the Great Circle is a first person game that puts you in the shoes of Indiana Jones and sets you on an epic cinematic adventure around the globe.
Its a story-driven game, with a heavy focus on the characters. We have lots of cutscenes, around 4 hours of them in total.
Also, first person perspective means that the player gets to see the faces of enemies very close to the camera. And with such a game our characters have to look as good as possible.
Another critical point for us was simplifying hair creation workflow for our artists. The game has many characters and we want to free our artists to do creative work.



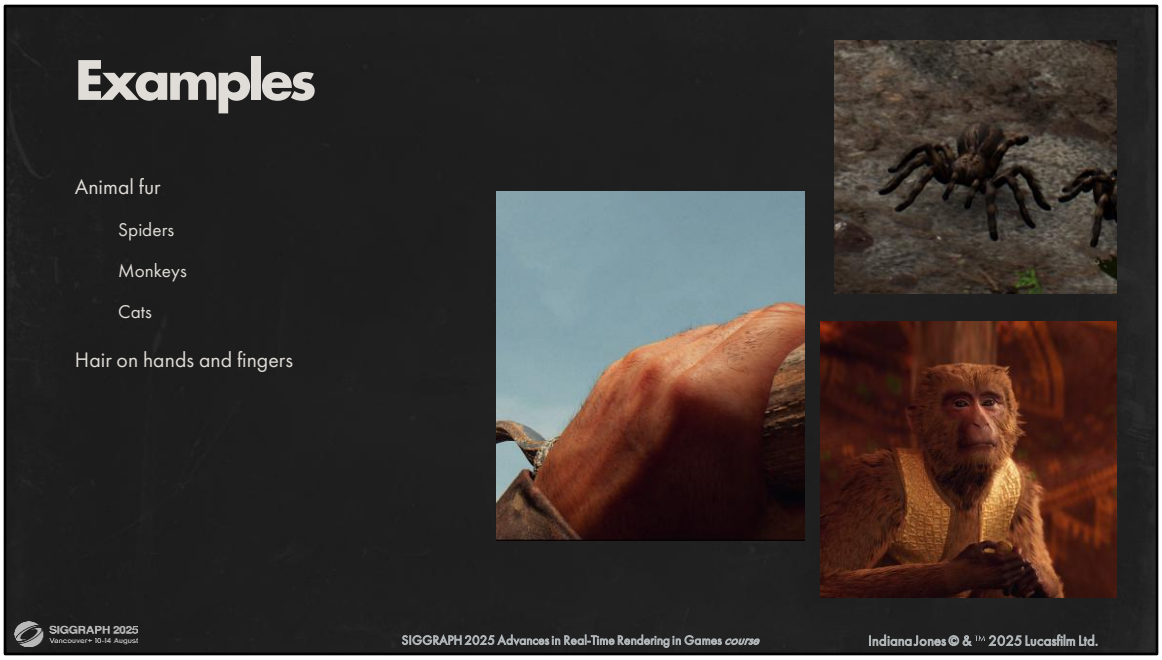

We also have several requirements:
First, we always had a strict 60 fps target for responsive and smooth gameplay, that was a clear goal and all systems we develop have to run at 60 fps for all our target platforms.
The GPU budget for the entire system was estimated as 2 milliseconds for simulation plus rendering on min spec at worst-case scenario, which is around a dozen characters on screen with some of them close to the camera. Of course in simpler scenarios we aim for hair to be faster than that.
We want to avoid putting more work on the artists. We do not want to create 2 version of each hair asset, so the system has to be fast and robust to be used everywhere. In the end we only had a couple of assets that still use hair cards, mostly animals.
And assuming those 2 requirements are met, we want to get the best visual quality we can afford. We want physical correctness, but if we can get a lot of performance by deviating from it, we will do that.



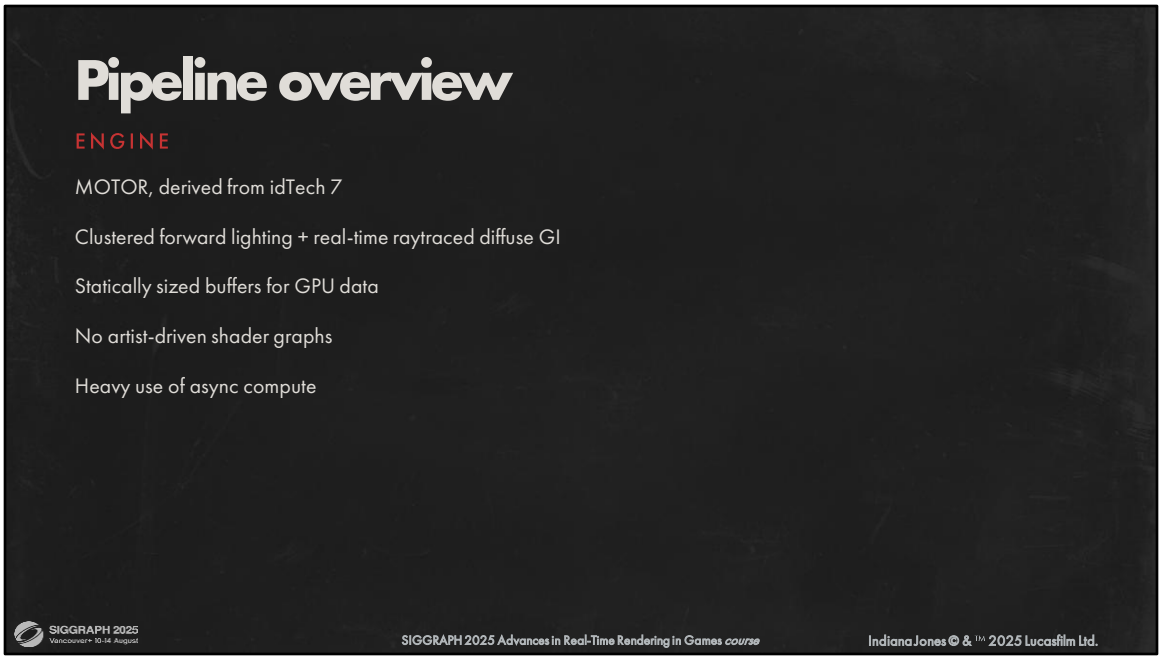

I want to expand a bit on how hair creation workflow looked for us.
When using hair card workflow we have to make the card atlas first from strands and then achieve the style we want by placing and sculpting the cards. After that we need to tweak normals, export everything to the engine and tweak materials there. When iterating you often need to repeat a few steps. And most of them are very technical and take a lot of time.
But when we use strands directly, we can get rid of 2 steps in the pipeline. Now we don’t have to do anything with mesh normals and don’t need to create cards. All that we need to do is to style the hair splines directly, we use xGen Interactive Groom but you can also use Houdini or xGen, and tweak hair materials after export. Each iteration is significantly faster. The differences between workflows are large enough so we can't make one version automatically from the other. That’s why we don't want to do 2 versions for our hair.



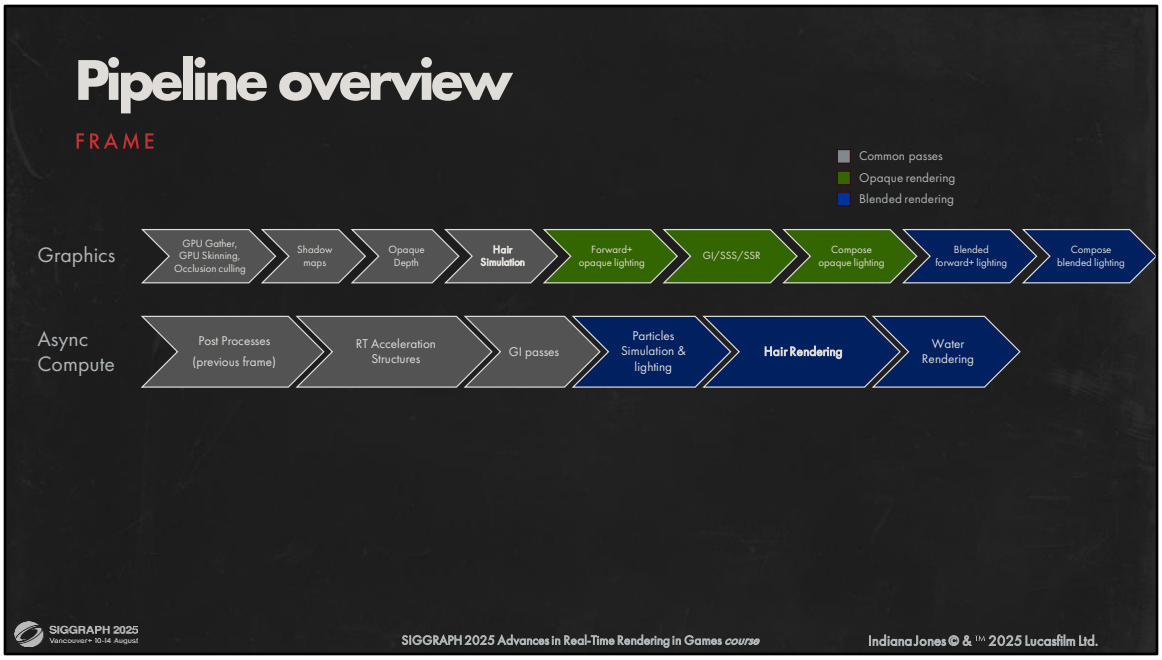

Here is a few examples of what we used the strands for. First, all character grooms and facial hair are done with strands. I think we have 2 corpses in the entire game with hair cards.
Some fur elements on clothes are also done with strands, for example, this hat and collar on Indy's winter jacket.




We also tried to use strands for animal fur. It wasn't a complete success, but we ended up using them for some animals, like these spiders and this monkey. Dogs you can see in the game still use hair cards. There were some issues preventing us from using strands on all animals, we will explore it a bit later.
Hair on hands and fingers were kind of last-minute additions to the game, but we were quite happy with how it worked out.



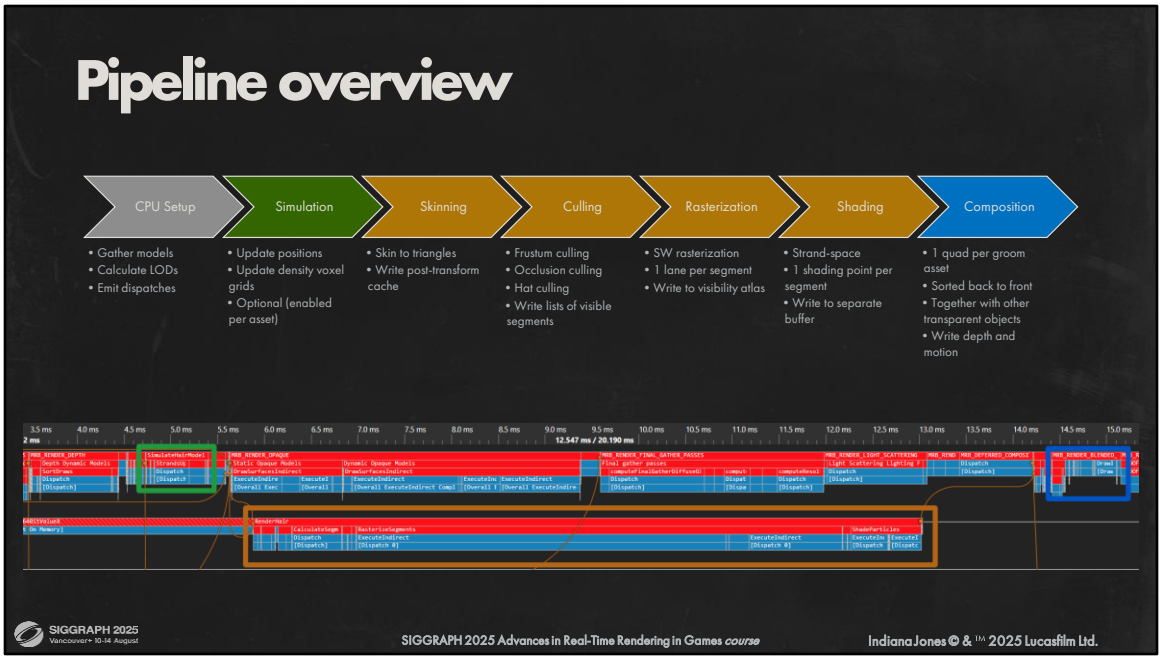

Before diving into hair pipeline, let me give you a bit of context about the engine. Our engine, MOTOR, is based on idTech 7 and shares a great deal of rendering code with it. It uses clustered forward rendering for direct lighting calculations, but some opaque passes are deferred, like diffuse GI. We allocate most of our GPU memory upfront and can't break these limits mid-gameplay. All our shaders are written by graphics engineers for maximum control over the code and maximum performance. The engine also heavily utilizes async compute, most of our frame is covered by some async work.




Here is a general outline of a single rendered frame. Width of each segment does not reflect the actual GPU timings, but rather its place in the frame with regards to work in the neighbor queue. For purposes of this talk we need to know that we have 3 large portions of the frame:

- Common passes, such as GPU scene gather, shadows, raytracing BVH construction and so on
- Lighting passes for opaque pixels
- Blended geometry rendering

We try to utilize async compute by working on blended geometry as early as possible and have some of it finished while the main queue draws opaque pixels.




Now lets look at our hair rendering pipeline. We have a small CPU setup part that just gathers all hair models in view, calculates appropriate LODs for them and emits dispatches for GPU.

On the GPU side, we split the pipeline into 3 parts. The first is simulation. Artists can select it per hair object. We do it on the main graphics queue, because we want hair simulation to be done before we start opaque lighting passes.

Second part is actually rendering the hair into a layered visibility buffer and doing vertex shading. We do that on async compute queue to overlap calculations with our forward opaque lighting passes and diffuse GI.
It allows us to get a more uniform load on the gpu. Hair workloads are very predictable and uniform, so they are filling the gaps in GPU utilization nicely.

And the last part is to compose rendered hair on screen. We treat the hair as any other transparent geometry in the game and render it back to front in the same pass with other geometry.

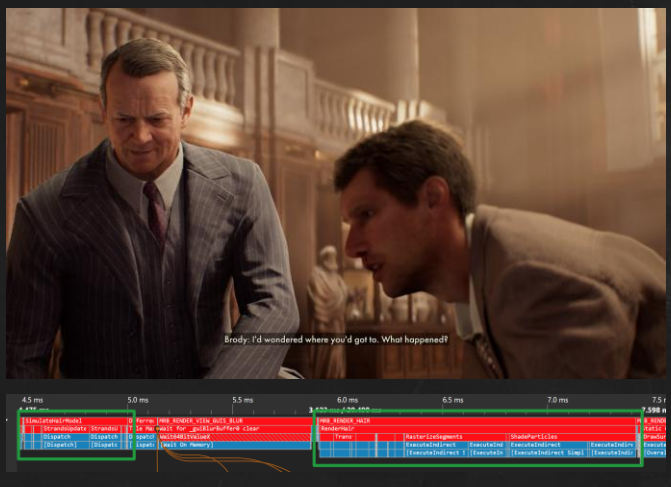
In the lower part of the slide you can see how it fits together with the rest of the

frame. Don't be scared by the duration of async compute passes, the actual workload is much smaller, its just fills the gaps in the graphics queue.

# Approximate timings

Async compute disabled

| Pass | Xbox Series S, 1080p | Xbox Series X, 1800p | PS5 Base, 1800p | PS5 Pro, 2160p |
|---|---|---|---|---|
| Simulation | 0.49ms | 0.36ms | 0.36ms | 0.29ms |
| Skinning | 0.17ms | 0.14ms | 0.14ms | 0.14ms |
| Culling | 0.10ms | 0.09ms | 0.08ms | 0.07ms |
| Rasterization | 0.61ms | 0.55ms | 0.55ms | 0.45ms |
| Shading | 0.62ms | 0.40ms | 0.39ms | 0.26ms |
| Composition | 0.40ms | 0.40ms | 0.42ms | 0.43ms |
| **Total** | **2.39ms** | **1.94ms** | **1.95ms** | **1.64ms** |




Async compute hides 0.5ms on average




If we disable async compute, for the average shot we will get the numbers shown on screen. You can see we exceed our budget on series S, but when measuring with async compute enabled we get back around half a millisecond which fit into our predictions.
On other hardware timings for passes are a bit different with respect to each other, but they sum up to similar numbers.

# Starting point

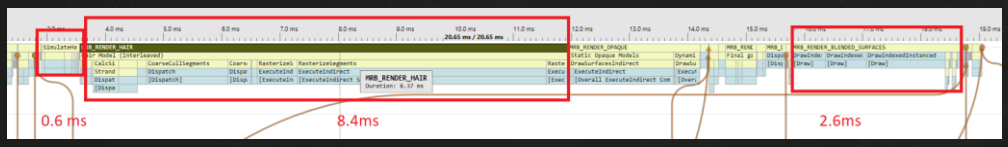
2 hair models on screen

Per pixel lighting

Simple software rasterization

8.4ms to cull and rasterize on Series X

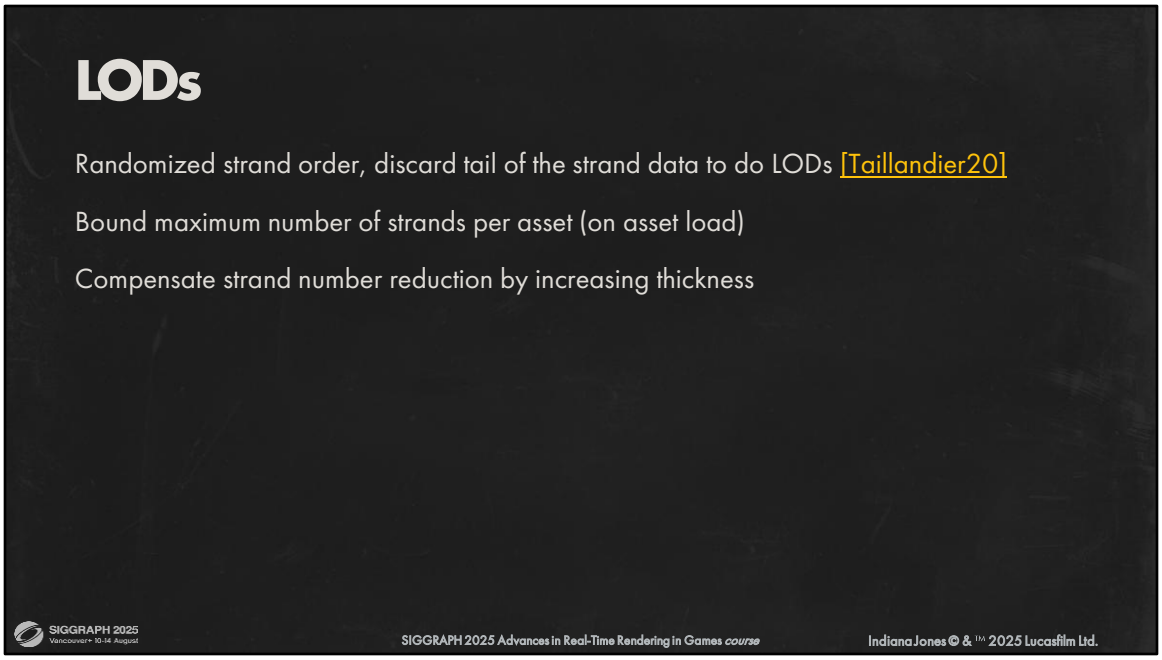
2.6ms to shade and compose



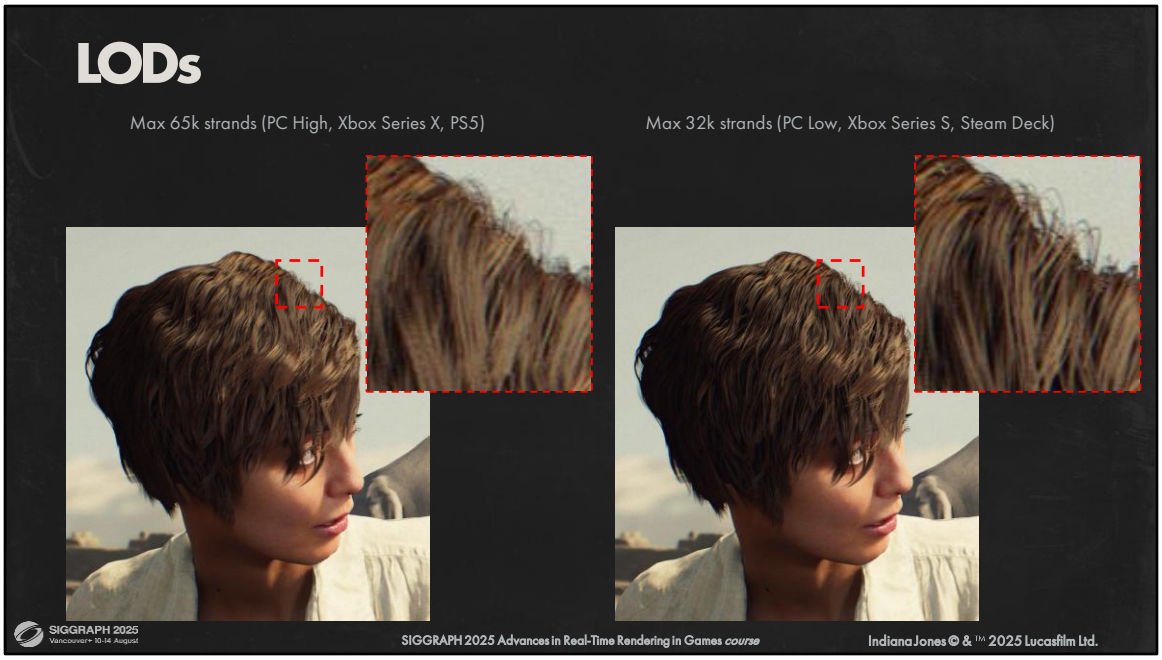




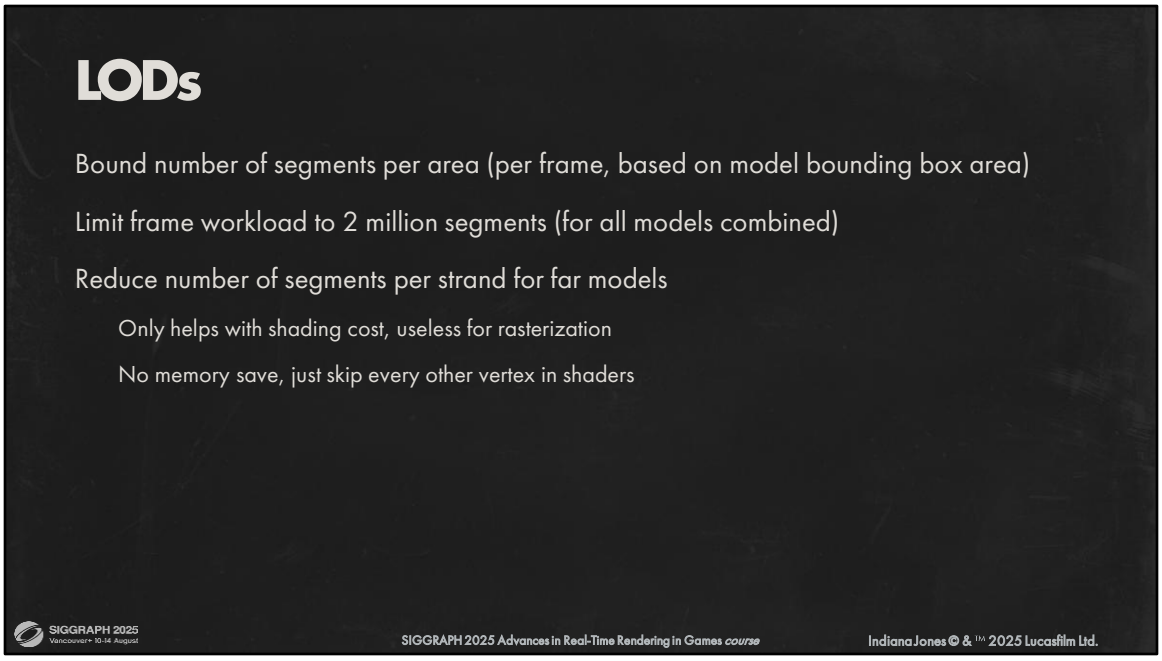
And for perspective, here is one of our first performance measures we did for the hair system. Most visual features were already there, except shadows from hair on the skin. But as you can see, we had a very long road ahead of us to make it work with our frame budgets.




The first thing we have to do is to find the level of detail for each hair model in view. We don't use separate LOD models, instead we use the common practice of randomizing strands order. Then we can take only a section of strands buffer, scale thickness of each strand accordingly and hair shape will still be the same. So the only thing that we need to calculate is the strand number for each model. We do that in 3 phases.
First, on model loading time we limit the maximum number of strands according to quality settings. Small hair models are not reduced because they don't contribute much to rendering time and reduction is more visible for them.



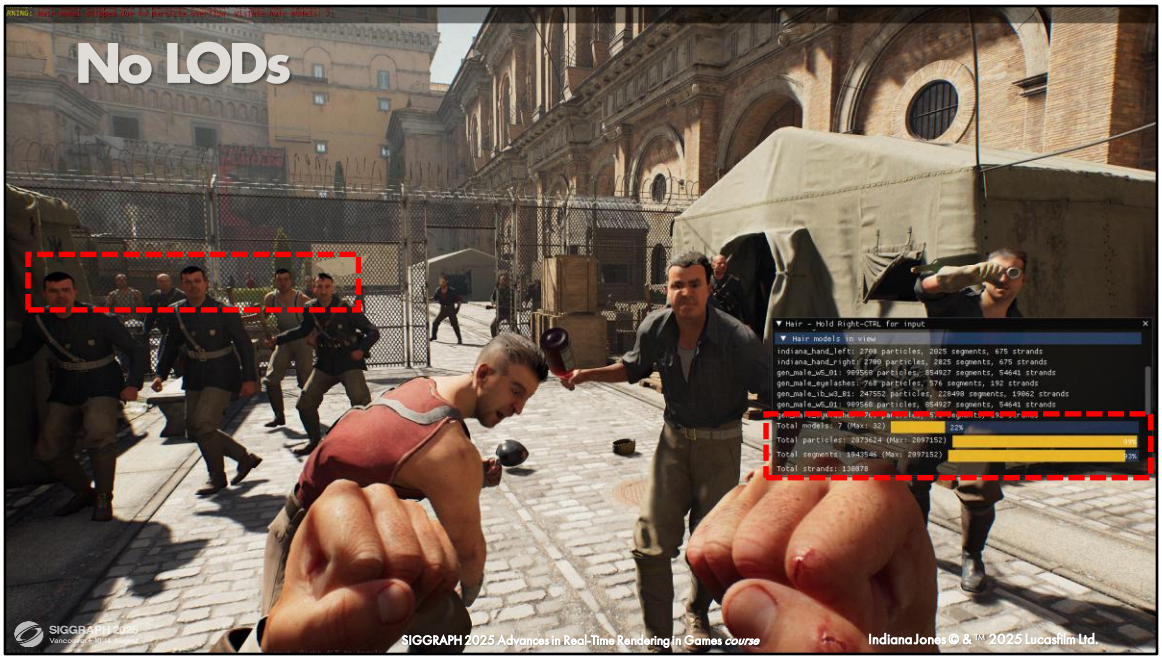

Here you can see 2 different options for maximum number of strands in a model.
On the left is our maximum quality with 65 thousand strands, and on the right is our minimum quality setting that we use on less powerful hardware.
As you can see, the visual results are close, but lower quality version is darker.
This is related to how we do self-shadowing, we will explore that a bit later.



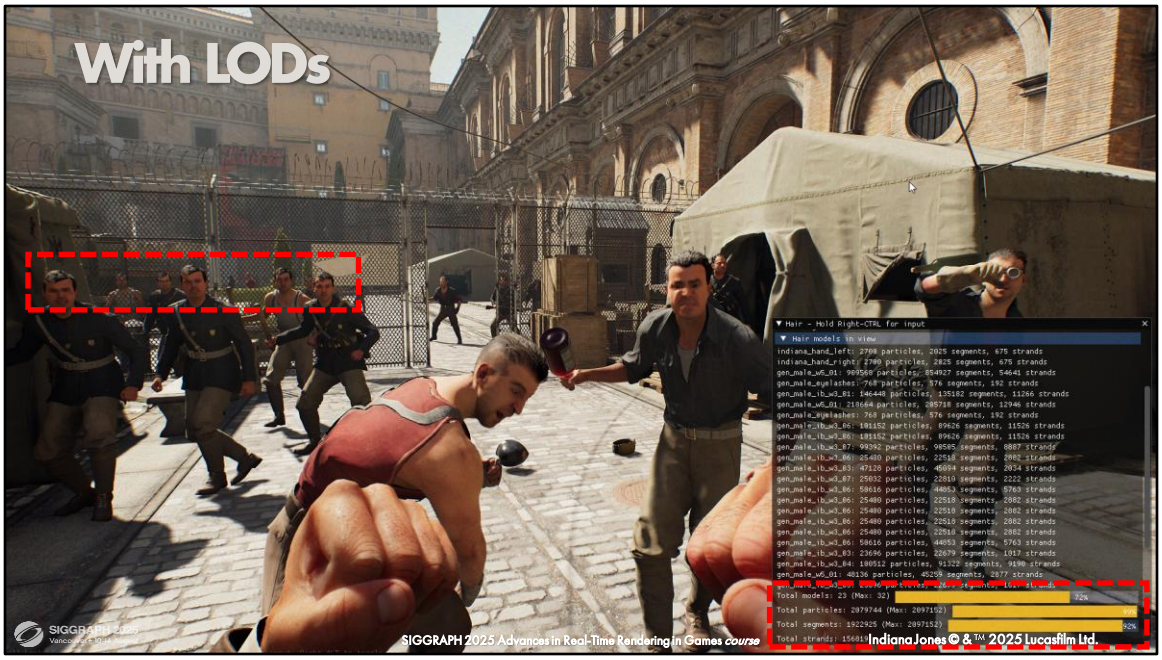

Then, each frame we compute screen area for each hair model and scale number of strands to maintain a certain limit of rendered segments per visible area. We use an upper boundary of 2000 segments per 64x64 tile on all platforms, as it gives a good visual result and performance for us.

Once we have our initial LODs calculated for each model, we look at the total segment count for all rendered models and limit it at 2 million segments. If we exceed the limit, we proportionally reduce number of strands in each model until we fit into limits.
This gives us an upper bound for all per-vertex calculations and memory.

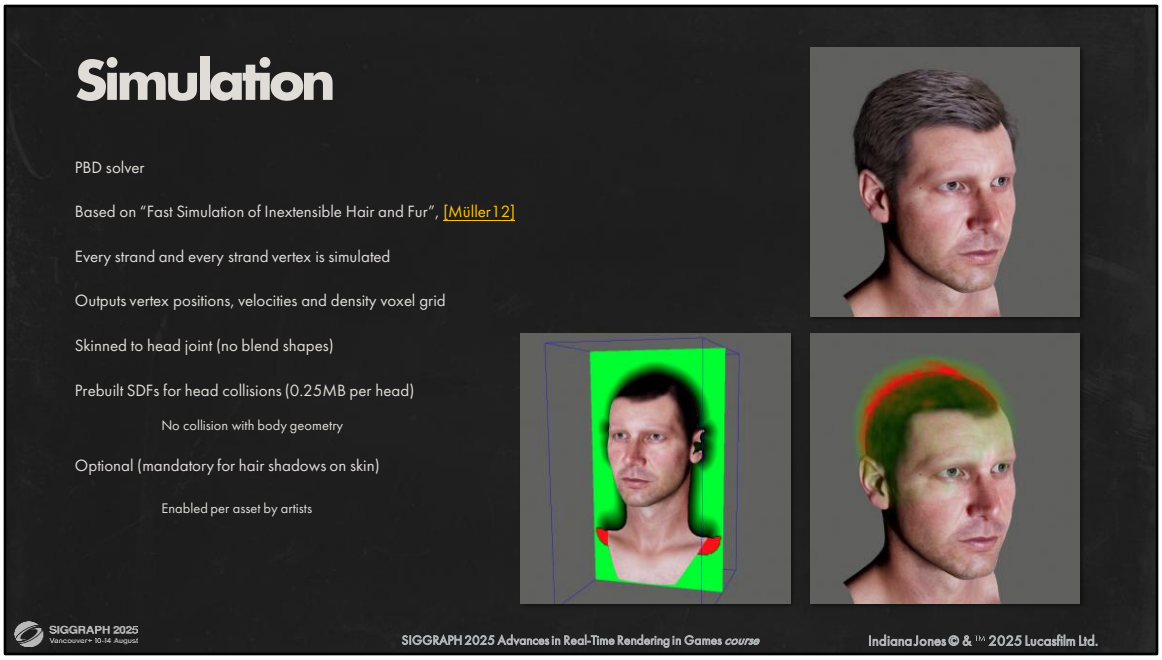
Additionally, to reduce per vertex calculations further, we can dynamically skip every other hair vertex in a strand. Its done on each model individually based on the distance. It reduces the number of shaded vertices by half, but doesn't help with rasterization cost, because that scales with the number of pixels covered, and not with the number of segments themselves.




Here is a demonstration of how this helps us maintain a large number of characters with strand hair on screen. Without LODs we can only fit 7 hair models in frame, and 2 of them are Indiana's hands ☺ Other characters in the back don't get anything. Which is an issue in a game where you can walk into a crowded enemy camp




But with LODs we can fit everyone and maintain good visual quality, and we still render the same number of segments




For simulation, we based our solution on Mullers work "Fast Simulation of Inextensible Hair and Fur". We simulate all strands and all particles in each strand. It happens before head skinning is applied, so we only account for main head joint movement at the time of simulation.
For collisions with the head we pre-bake head geometry into SDF. We don't do anything for hair-to-body collisions because hair styles in our game are not that long. Once simulation step is over we output local positions for hair vertices and their velocities for following passes. We also output voxel density grids for each simulation. We need that for hair collision and for hair shadow calculations.


Next step is to find final positions for each hair vertex. Here we skin all hair strands directly to triangles on the base mesh. On asset import stage, for each strand root we compute the closest triangle on the parent mesh, its barycentrics and tangent frame.
We store 80 bytes of information per strand root: index of parent triangle, packed barycentrics, triangle normal and tangent. For tangent frame we use octahedral encoding. Head meshes have multiple LODs with different triangle indices, so we need to bake this information for each LOD of parent mesh.

At runtime, we always pre-skin all head meshes, so we can just fetch transformed triangles from skinning buffers and transform hair strands accordingly.

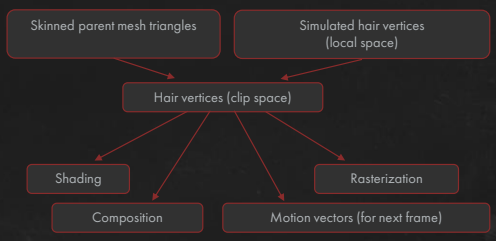
# Skinning

Store post-transformed results in global cache (before division by w)

32MB for 2 million vertices (1 million on low spec)

Takes 0.17ms on average for Xbox Series S

Bounded by memory bandwidth







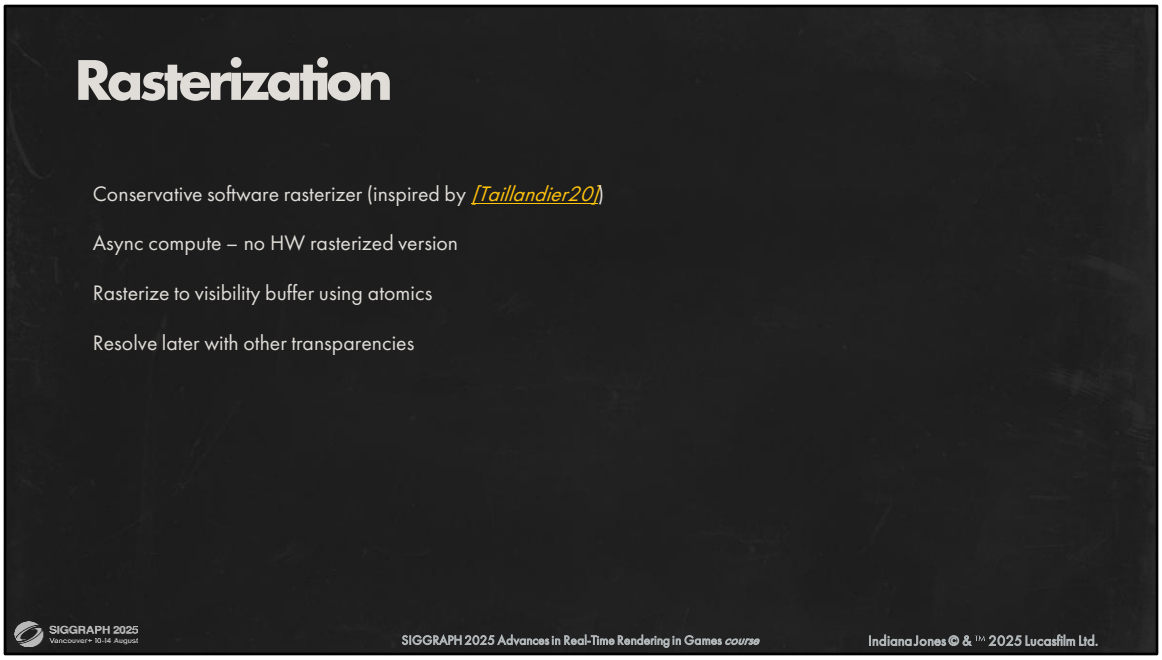
After skinning is done, we transform hair vertices to clip space and store results for future calculations. It requires quite a significant amount of VRAM, but since we can reuse this data in multiple passes, it is well worth it. Later, on composition stage we need positions per pixel to calculate motion vectors and interpolation coefficients for shading, so it is critical for us to cache transformed vertices.


Now that we have our transformed vertices, we need to emit a list of segments to rasterize. We do a coarse culling stage here to discard invisible segments. We do 3 types of culling:
Frustum culling, occlusion culling and hat culling.
For occlusion culling each segment is tested against downscaled depth buffer. If you have close-up shots where each segment takes a significant portion of the screen, you might want to build a full Hi-Z chain, but in our case just using 16x downscaled depth buffer was enough.

Last step is so called “Fedora culling". Hat is an iconic part of Indiana Jones and we have a lot of scenes where he puts the hat on and off, so we needed a solution to prevent hair clipping through the hat. We tried different approaches, but what worked for us was having 2 shapes. First is the exclusion box, it culls any segment above the hat. But as you can see, there is a small area right under the hat where we don't want to cull any hair. To prevent it, we have a sphere, that we cut out of culling box to preserve these segments.

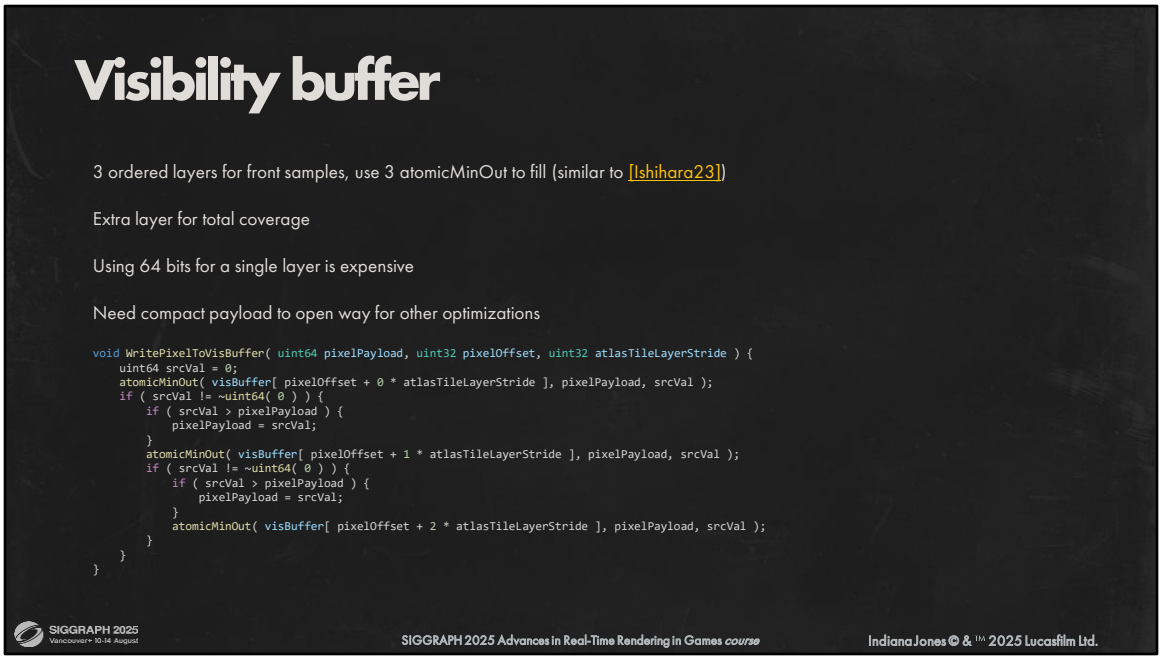
If segment passed culling, we atomically raise a visibility bit for the segment. To reduce number of atomic ops, we utilize wave intrinsics and only do actual

atomic write once per wave.
Also, during that stage we compute real screen boundaries for each hair model to calculate how much space it needs in the visibility atlas.



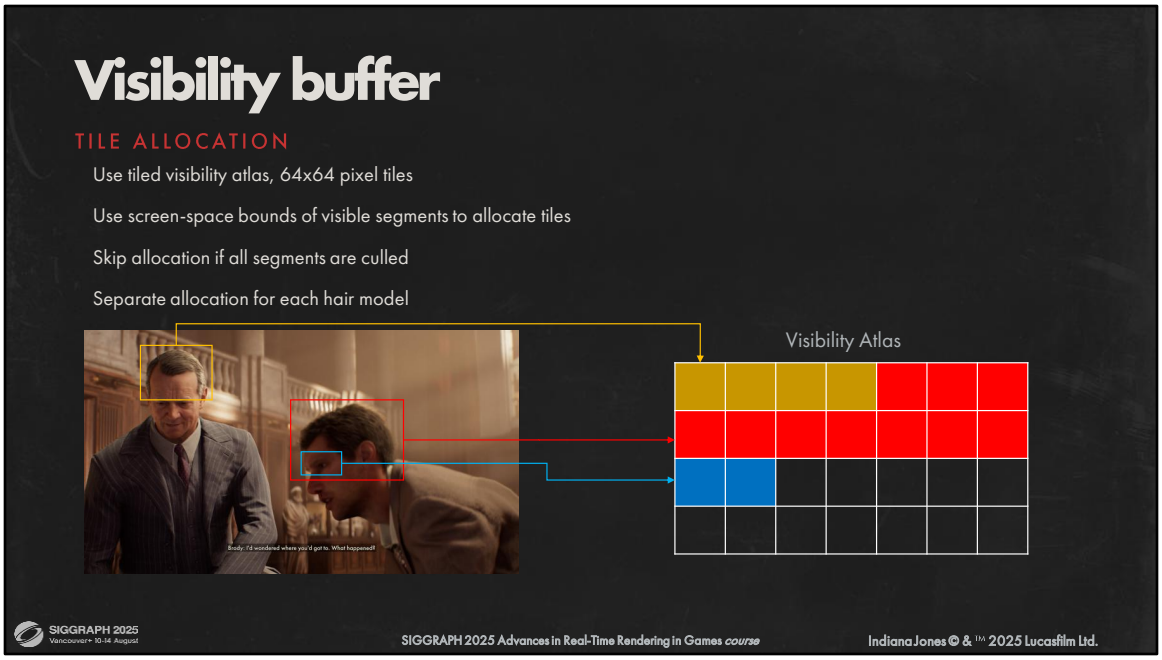

Let’s move to rasterization. As you might know, hardware rasterization has suboptimal performance when primitives you are trying to render are small. And hair segments are definitely small, often much thinner than a pixel. There were other reasons for using software rasterization as well. We wanted to be able to offload as much work to async compute as possible, and it’s easier to render perfectly anti-aliased lines like that.
All these reasons combined moved us to writing our own software conservative rasterizer, optimized for short line segments.


We consider each line segment a trapezoid and rasterize them into visibility buffer. Each shader lane gets one segment and rasterizes it using a simple 2-dimensional loop, similar to one shown on the slide. For each pixel we calculate its coverage, do a depth test against full precision depth buffer and output it to visibility buffer using atomic operations.
The algorithm is very simple, but also very slow. It has a lot of dependent reads. GPU cache utilization is far from ideal and lane divergence is high. Not to mention that writing to multilayered visibility buffer is a very slow operation. How can we make it run faster?



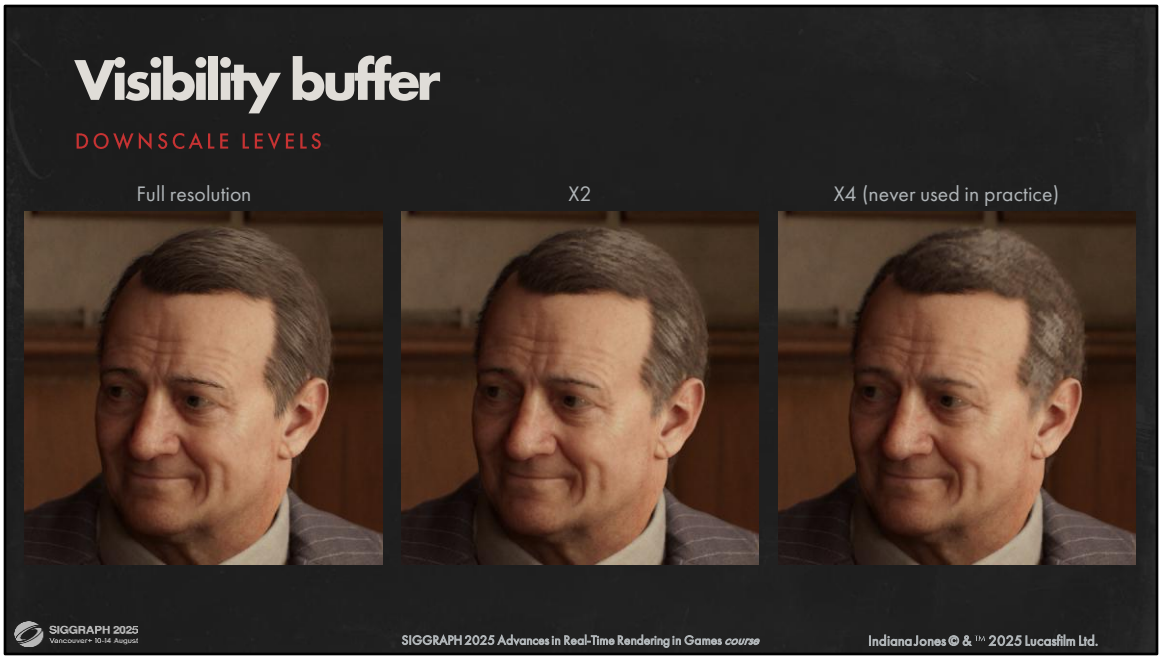

Let’s start by looking at our visibility buffer, since writing to it is our most expensive operation by far. We have 3 layers of fragments in each pixel, ordered by depth, and a separate layer with additive pixel coverage. To write the pixel to visibility buffer we use a simple code shown on the slide. This means our visibility buffer need to be quite beefy, with 3 64 bit values per pixel, and 32 bits for additive coverage. That resulted in 56 megabytes for 1080p just for visibility buffer, which is a lot and we don’t want that. Using 64bit atomics is ok performance-wise, but to enable later optimizations and reduce memory footprint we want to use 32 bits for our payload. Another issue is that this code has a lot of dependent atomic operations, it is quite slow.


To fit the payload into 32 bits we need to compromise. We don't want to render too many segments on screen anyway, so limiting the maximum number of segments in a model to 22 bits was acceptable for us. That leaves 10 bits for everything else. We can reconstruct everything from segment index, but depth has to be in the payload for atomicMin to work correctly. However, 10 bits is not nearly enough for scene depth. What we do instead is we renormalize and linearize depth values inside model bounds. That way 10 bits is enough. We have a couple of z-fighting artifacts here and there, but they are not very noticeable. Another important consequence – we don't' have any space left for hair model index in the payload. So we can't have 2 models output fragments to the same visibility buffer area.

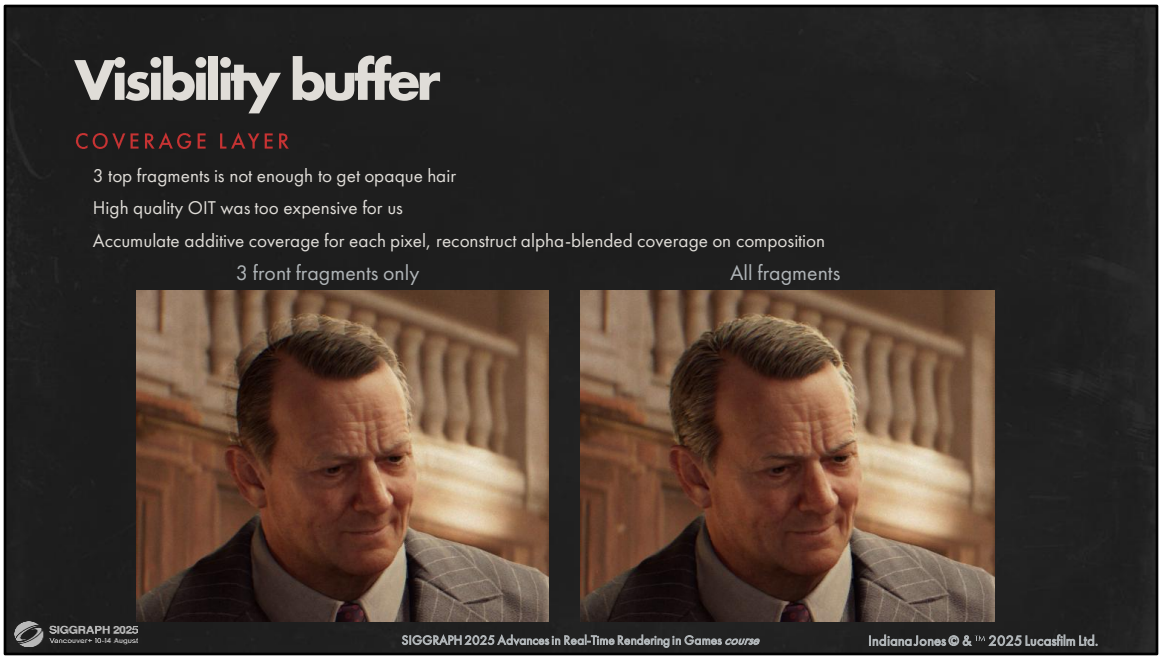



So instead of single full-screen visibility buffer we use an atlas of small tiles. Each hair piece allocates tiles from the atlas based on its screen-space boundaries. Overlapping hair pieces will allocate different tiles, so each tile contains only one hair model. It also allows us to decouple memory spent on hair atlas from resolution. But it also comes with its own set of issues.


Our atlas has a finite size, and overlapping hair models might require more tiles than we have in our atlas. We detect when it happens and downscale each tile until all tiles fit into our atlas. It also has a nice side effect – we always have an upper boundary for hair rasterization, because number of pixels in the atlas is fixed.
To avoid visible pixelization on downscaling, we stochastically sample our atlas on final composition.




In theory we can fit almost unlimited amount of hair, but in practice we never had to go further then half res. Here you can see how downscale looks visually. Usually it is active when you have multiple characters close to the camera and your display resolution is high.




I mentioned earlier that we have 3 top layers in our visibility buffer. But usually a typical strand covers only about 20-25% of pixel area. 3 front samples is not enough to get opaque coverage. We didn't want to do order-independent transparency because its not very fast and complicates rendering considerably. We needed a fast approximation for combining dozens of fragments in a pixel. What worked for us it to accumulate coverage additively using an atomicAdd, and reconstruct alpha blended coverage later, on the composition stage

# Visibility buffer

## COVERAGE LAYER

Numerically simulate coverage as a function of number of strands and their thickness

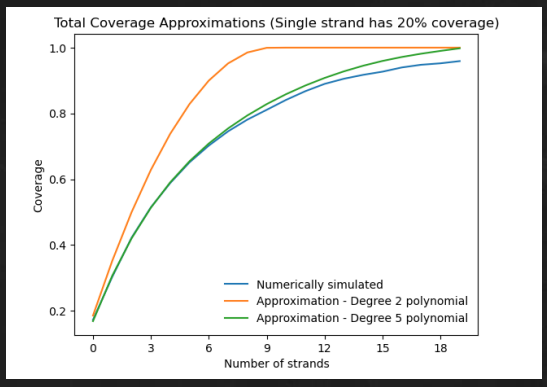
Assume constant coverage for each sample

Approximate with a polynomial

Shipped with quadratic polynomial

Gives more volume visually compared to more precise fit

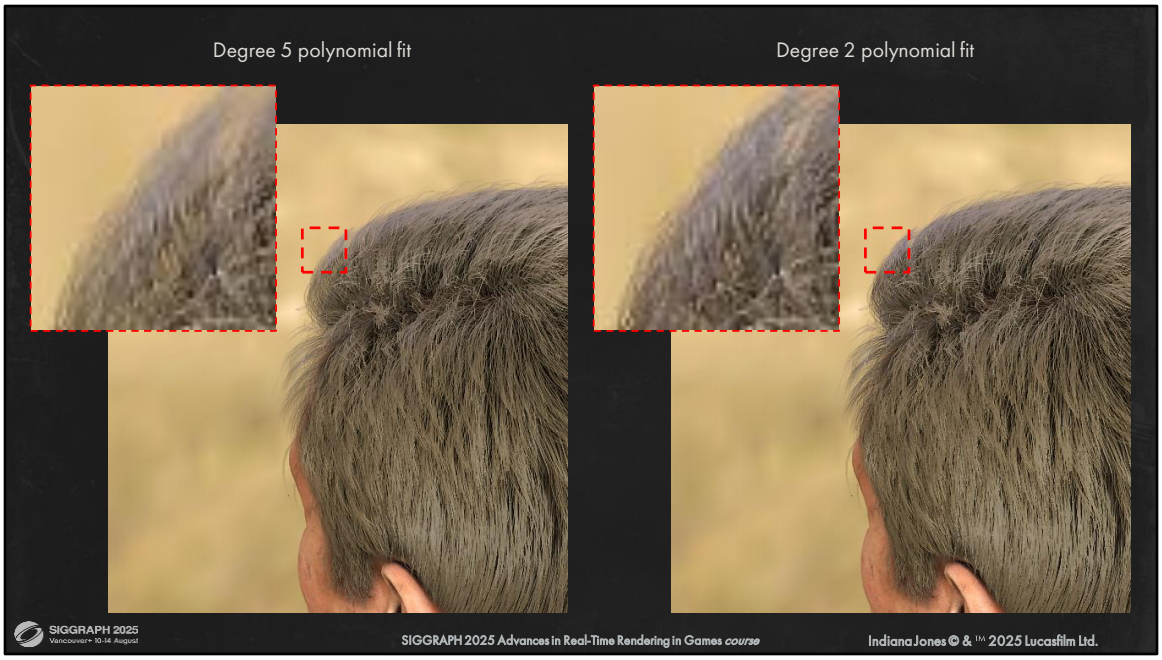
```
void WritePixelToVisBuffer( uint32 pixelOffset, float coverage ) {
    uint64 srcVal = 0;
    // Write all fragment layers...
    ...

    // Add fragment coverage
    uint32 additiveCoverage = ( coverage * 1024.0f );
    atomicAdd( coverageBuffer[ pixelOffset ], additiveCoverage );
}

// Later, on composition, reconstruct total coverage
float x = additiveCoverage / 1024.0f;
float totalCoverage = saturate( x - 0.5f * x * saturate( 0.5f * x ) );
```







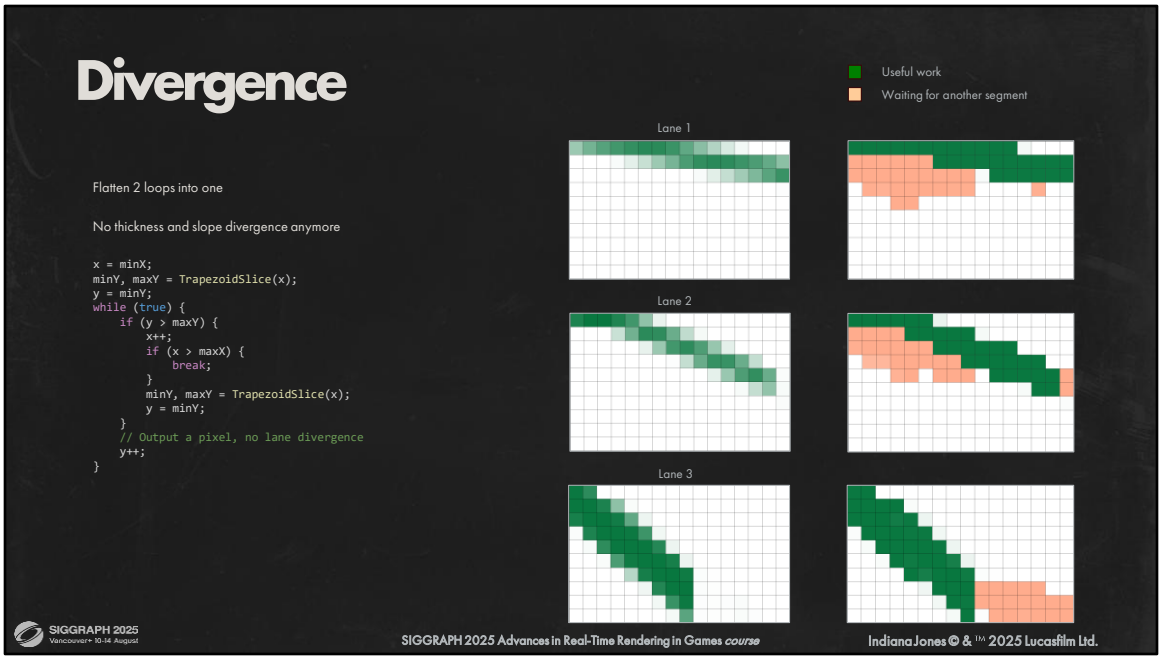
To convert additive coverage to alpha blended, we make a few assumptions. First, we assume that all strands have similar screen-space thickness. Second, we assume similar BSDF properties for all strands covering a single pixel. Given that, we can numerically simulate total coverage as a function of additive coverage by drawing a bunch of opaque lines and calculating precise total coverage after each line is drawn. Then all we have to do it to approximate numerical results with a polynomial. We found that $5^{th}$ degree polynomial is a very good fit, but shipped with a quadratic approximation as it is slightly cheaper and gives more visually opaque result.




Here you can see the visual difference between 2 approximations. As you can see, visually results are very close to each other.


One other sketchy, but very efficient optimization – we can cull early against the last layer of our visibility buffer. The catch is we can use non-atomic load to utilize caches. Mixing atomic and non-atomic operations for a single memory location is a "grey area", that's why its sketchy. It saves quite a lot, up to half of rasterization time for some models.


Lets move on. Other thing that we had to fight was very high lane divergence. On average we had less than half of lanes doing any useful work at any given time. We had 2 nested loops with divergent bounds, and very different segments can end up in the same wave. Some might be thicker than others, some might have different inclination.
And because all instructions in a wave are executed in a lock step, each lane must wait for its neighbors, even if this lane could have moved to the next column. You can see it illustrated on the slide. We rasterize each segment first top to bottom and then left to right. If any segment has active pixels in current column, all segments in the wave will wait for that segment. This is illustrated as orange "dummy" pixels.

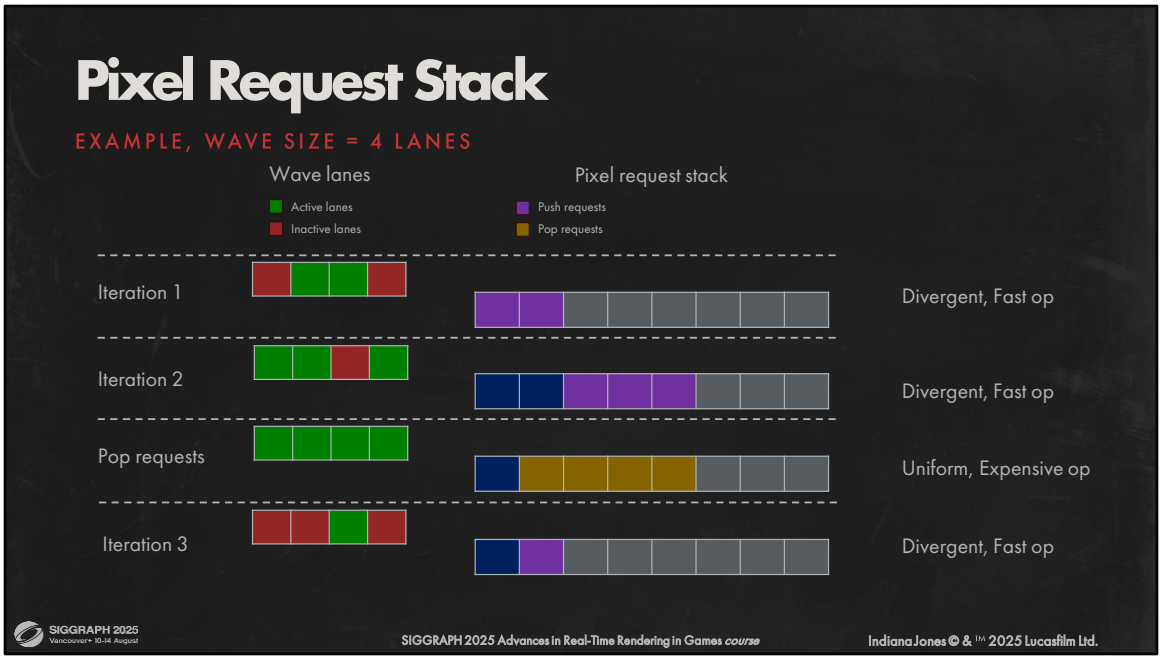



What we can do in that situation is we can flatten 2 loops into one and transform inner loop into a conditional statement that moves to the next column. So we rasterize different columns for different lanes at the same time. That way we can remove any divergence that comes from different inclination and thickness.
We will still diverge if the total number of rasterized pixels is different for segments within a wave.


Now that we removed one source of divergence, lets look at other sources. Lanes with less pixels still don't do anything at the end of rasterization loop.
Also we have to do fine grained depth test, and it is a source of divergence for fragments that passed the test and didn’t pass the test.
Plus all these operations are sequential and introduce stalls that are hard to hide.


To deal with that, we defer expensive global memory operations. We allocate some LDS for pixel write requests and execute them in a bulk once we have a full wave worth of requests. Lanes that don't have anything to do will skip the actual rasterization work on each iteration, but they will do global memory operations for other segments.
Initially we only did that for writes to layered visibility buffer but later added the same mechanism for depth tests. You can add more stacks, but each stack adds some overhead on LDS, so I would only advise to use it for the most expensive divergent operations.


Each wave in a workgroup gets its own stack space, and these memory segments do not overlap. We need to store the payload for visibility buffer, which is 32 bits in our case, and also the pixel position and precise coverage. Because waves don't touch data of other waves, we don't need group LDS barriers, only wave barriers. As you can see, we need quite a lot of LDS, so keeping request payload small is key to get reasonable LDS usage and keep good occupancy.




Here is an example of how pixel request stack works on an imaginary hardware with 4-lane waves. On the left you see the active lanes, on the right is the state of the stack. On first 2 iterations we don't have enough pixels on the stack, so we just add more requests, it's a fast operation and we are fine with divergence. As soon as we accumulated enough data to fill a wave, we take top requests from the stack and write them to visibility buffer. Its an expensive operation, but we have all lanes working on it. After that we continue to accumulate samples, flushing them to the buffer and so on.

# Divergence

## RESULTS

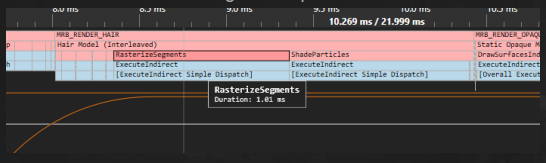
~80% lanes are active on average (20% divergence)

All heavy operations utilize all lanes

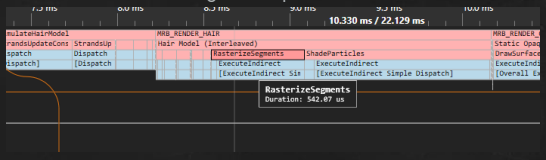
55% -> 45% divergence with loop flattening

45% -> 20% divergence with pixel request stacks

Before divergence improvements




After divergence improvements







Here are the results of our quest to lower lane divergence. We started with more then half of our lanes inactive on average, flattening the loops reduced divergence by 10%, and pixel request stack removed another 25%. And you can clearly see how it affected the timings, the shader runs almost 2 times faster.

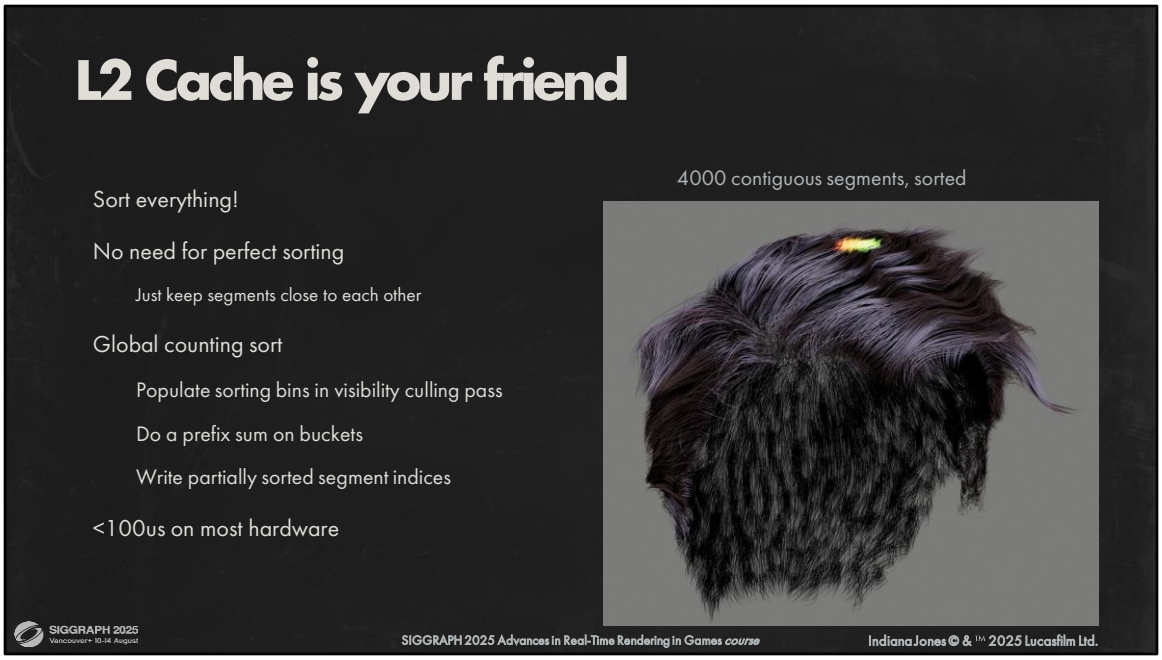
# L2 Cache is your friend

- Divergence is low
- Cache hit rate is low
- Strands are randomized
  - To simplify LODs
- No spatial coherency for pixel writes

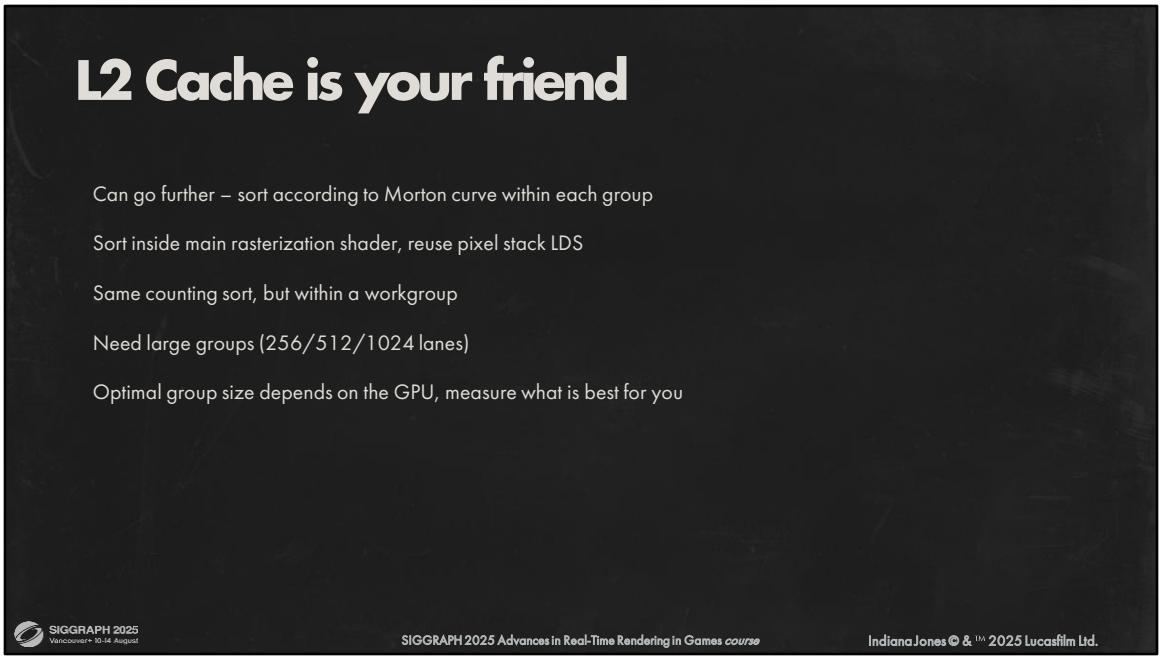
4000 contiguous segments, unsorted



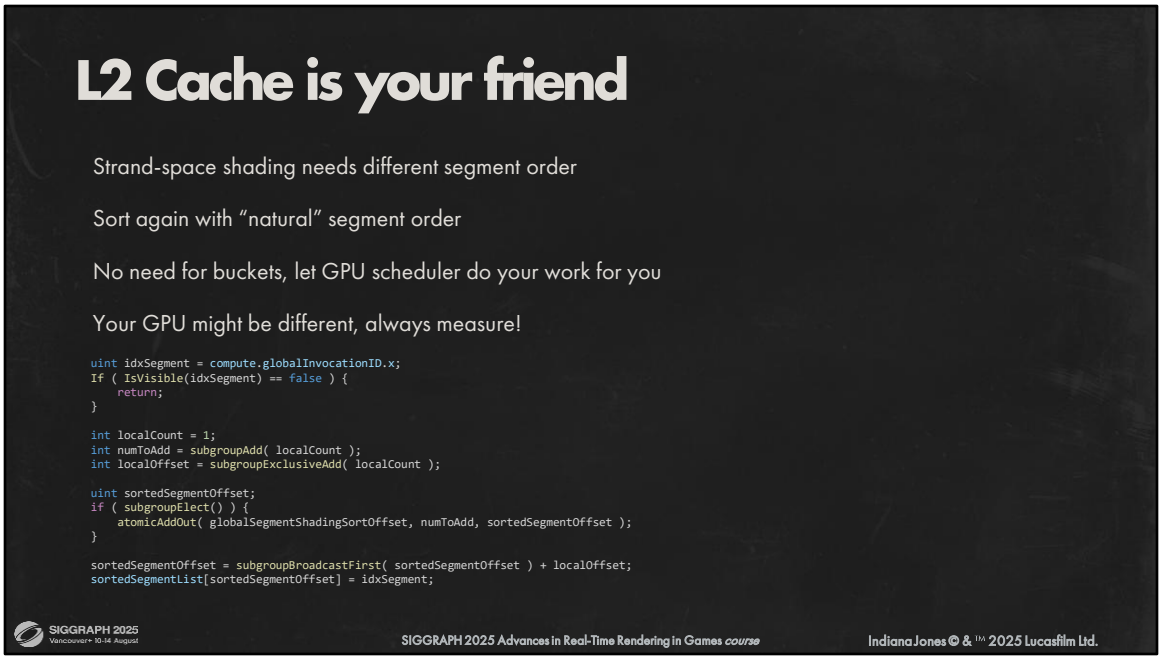

Now that our work is highly parallel, we face the next issue – low cache utilization. It happens because we randomize strands in the resource. Each lane tries to access different portion of the visibility atlas. Here you can see 4000 contiguous segments. They are all over the place. If we want good cache utilization, we need good spatial coherency for hair segments



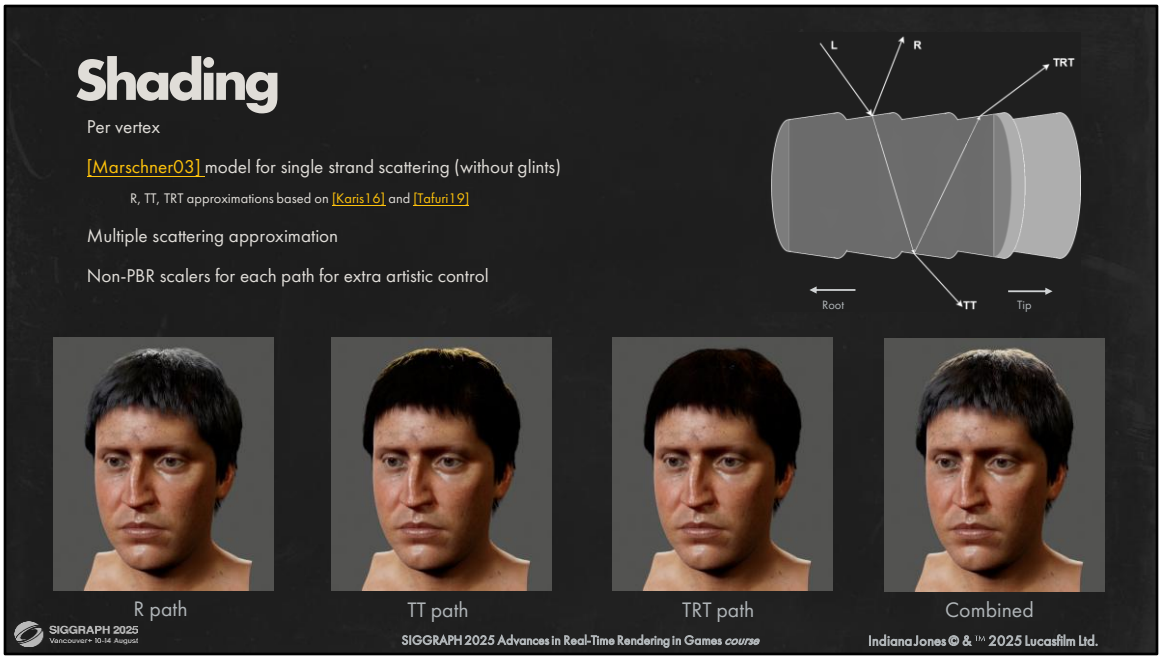

The answer for us was global sorting for all segments in a model. We don't need it to be perfect, we just need segments in the same workgroup to be close to each other spatially. We divide the screen into buckets, size of a bucket is close to the size of a visibility atlas tile. In the culling pass we populate bucket counters and do a simple counting sort. It is very fast, takes less than 100 microseconds. Here you can see the result, 4000 segments after sorting



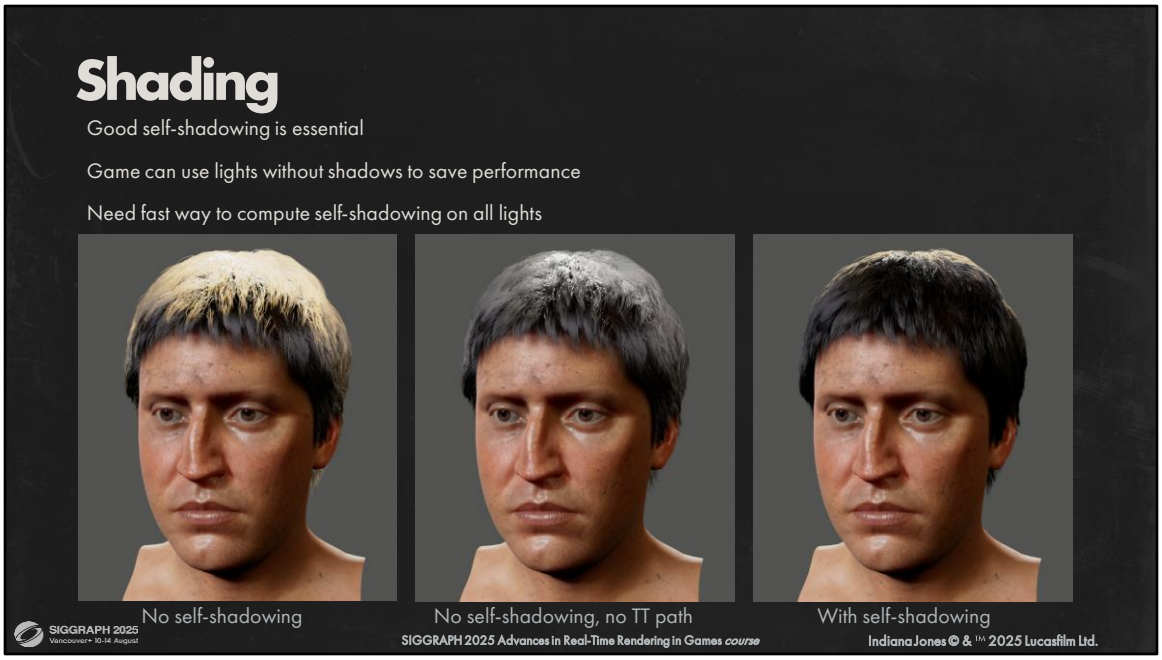

But the granularity of this sorting is not perfect. We have found that doing another sorting inside each workgroup right before rasterization has a positive effect on performance as well, although to a lesser extent. We use large groups to rasterize hair and sort segments inside each workgroup according to Morton curve. This is a minor optimization, it doesn't save too much and depends on the hardware. We noticed that different hardware performs better with different group sizes.




Later we do strand space shading, and shading fetches a lot of per-segment data. We can’t use the same sorting order that we had for rasterization, L2 cache hit rate will be low. So we need to sort the segments back, to their “natural” order. This time you don’t need buckets, you can abuse the order in which gpu launches waves. Another way would be to store 2 lists, for visible segments sorted spatially and for visible vertices without any sorting. It would work just as well, but then you’ll need more memory.




That concludes the rasterization part, let's move to shading. We shade strands per vertex to get smoother shading and save performance. We use Marschner shading model with approximation for reflection lobe from Brian Karis presentation at Physically Based Shading course and other lobes from Sebastian Tafuri presentation at "Advances in Real-Time Rendering". The model consists of 3 specular lobes – untinted reflection, transmitted lobe, visible when hair is backlit, and the second tinted reflection lobe. We also added non-PBR scaler to each lobe for artist convenience and last-minute tweaks. I won't be diving too much into the shading model itself, there is a lot of good material on the topic. What we will focus on is self-shadowing.




Marschner model gives very nice realistic results for human hair. However, both R and TT paths are very bright for most incoming light directions.
As a result, having good self shadowing on all lights is absolutely critical for hair to look good. That places us in a tough position for 2 reasons.
First is the game can use lights that do not cast shadows to save performance.
Second is that even if we only use shadow casting lights, we have to keep the number of lights affecting the hair to a minimum, and setups for cinematics can be very complex.
One easy trick we could do is to disable transmission path for unshadowed lights, but as you see, the results are still far from ideal. Reflection path still has a lot of energy.


For self shadowing we considered multiple approaches.
Deep opacity maps were considered at first but ultimately discarded due to memory requirements. Also having to rasterize each hair piece for each light was really a non-starter for us, just rasterizing the main view already takes a significant portion of our budget

# Self-shadowing

APPROACHES

Density volume raymarching for each light

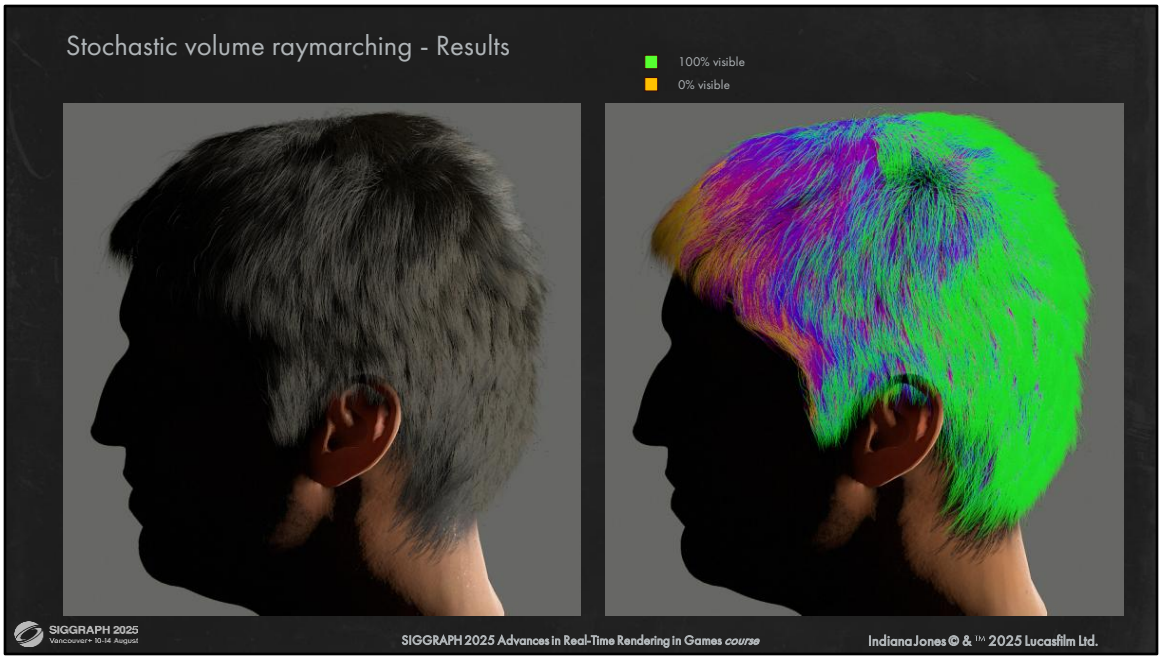
- Similar to [Jansson19]
- No extra memory
- Reuse density volume from simulation
- Scales linearly with number of lights
- Needs high grid resolution (up to 256x256x256 for some grooms)
- Generally good visual results
  - High-frequency details are missing




Second option that we thought of was to estimate global scattering at runtime by raymarching hair density volume we have from simulation passes. It is free on memory and doesn't require extra rasterization passes.
We can control the tracing speed by adjusting density volume resolution and number of ray marching steps. But it has scaling issues, each hair shading point has to raymarch a volume towards each light. But visual results that we can get from that are generally good on most grooms. However, you can’t get high frequency shadowing with this approach.


Also, some hair pieces require rather high resolution for density grid, for example, you can notice that this hair piece has quite visible self-occlusion artifacts. For our content, some grooms needed to have 256 voxels on each axis to resolve shadowing without artifacts, be that overocclusion or leaking


Another option that we tried was a modification of the previous approach. Instead of ray marching towards the light, we do a fixed amount of ray traces stochastically to get a spherical visibility function for each shading point and encode it into SH or SG.
Compared to previous option we can adjust number of these "visibility probes" independently of shading points (we tried 1 probe per 4 vertices and the results were very similar to 1 probe per 1 vertex). Plus we are no longer limited by the number of lights, we do one lookup to solve visibility,
no matter how many lights affect the shading point. The downside is that it takes a lot of time and grid resolution has to be very high to avoid hair being too soft, similar to previous approach.
For example, here the visibility pass takes 6 milliseconds for a single hair piece. We could have optimized it much further and throttle the workload over multiple frames, but visual quality will suffer, and its already not the best.




Here is how it looks for a simple groom. On the right you can see visibility representation for a single light. Lighting is plausible, but a little bit flat

# Precomputed self-shadowing

Trace visibility rays for each hair vertex offline, bake results to a spherical function

Fetch baked visibility at runtime and use for all lights

Essentially free

Can't handle strong deformations if hair local neighborhood is too distorted

Might not hold up for styles with very long hair if it moves too much

Acceptable for us

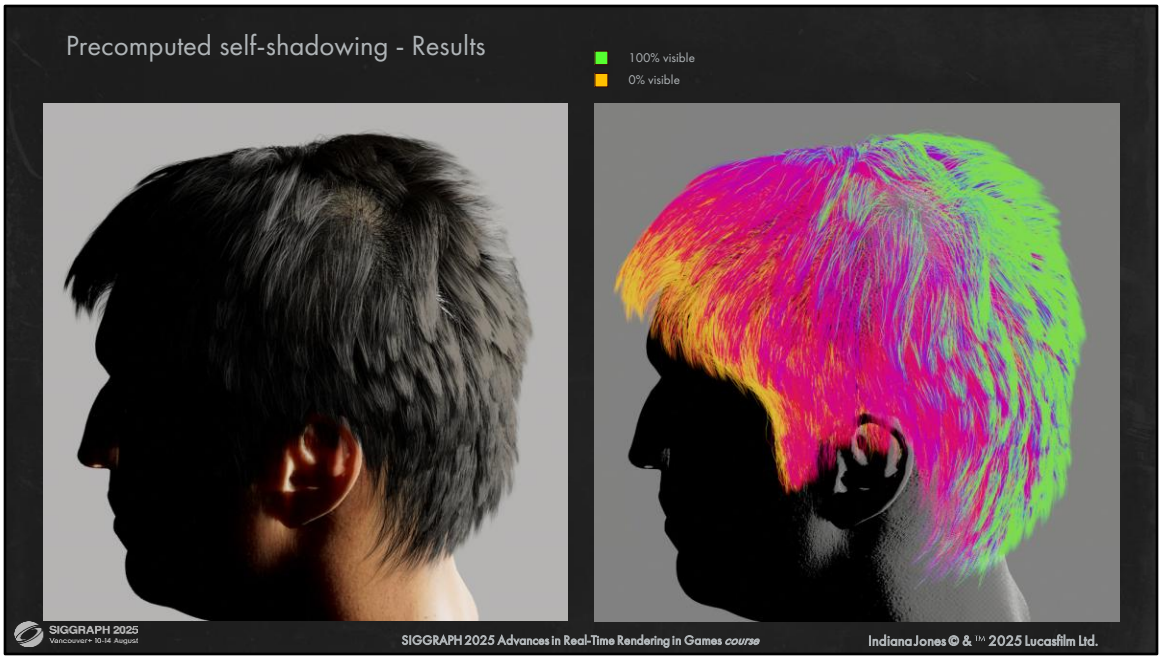
Longest hair style used in the game



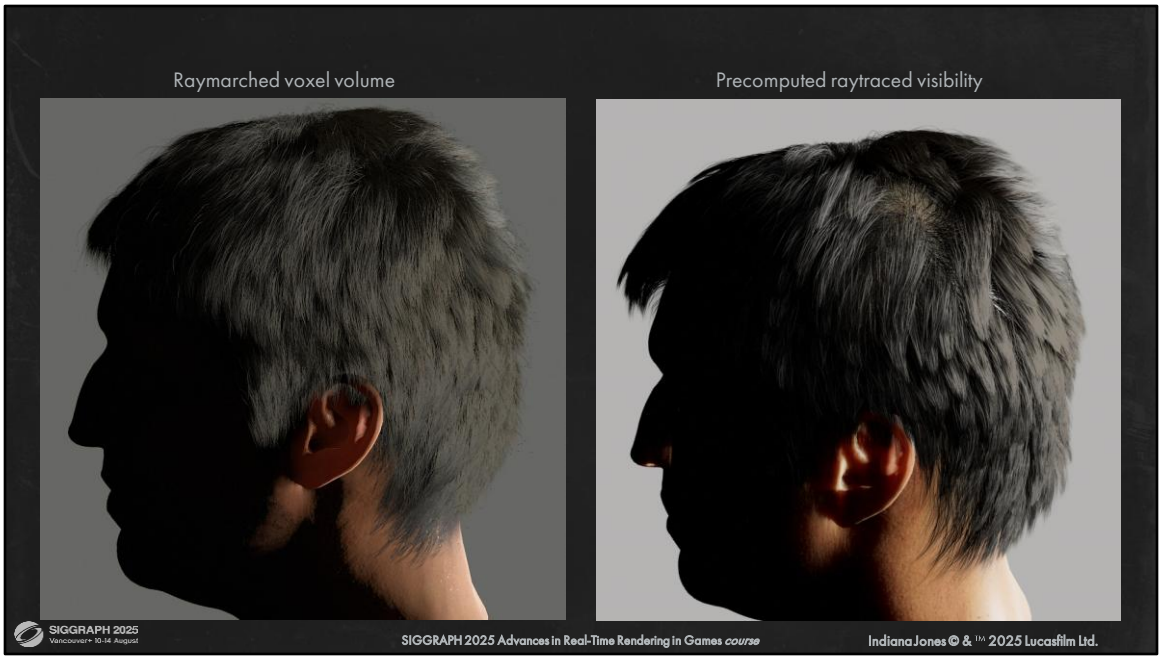

However, doing that teached us a few things:
- Most of our grooms are rather static and short, you can see the longest hair style we have on the slide
- For self-shadowing what matters most is short neighborhood of each segment, and it doesn't change too much under wind

With these constraints, we could just bake our visibility one time and reuse for many frames. Or, what we actually shipped with, do this baking offline with a software ray tracing instead of raymarching.
Using a ray tracer allowed us to have good high-frequency self-shadowing for each hair strand essentially for free, independent of the number of lights. We trace against the parent mesh as well so hair is always shadowed by the head.


I was afraid that using baked visibility will break down under motion, but honestly the results hold up quite well. You can see some artifacts under strong winds, like here, for example, but it was acceptable for us.




Here are the results for baked ray traced visibility. You can see that it handles high-frequency details very well.

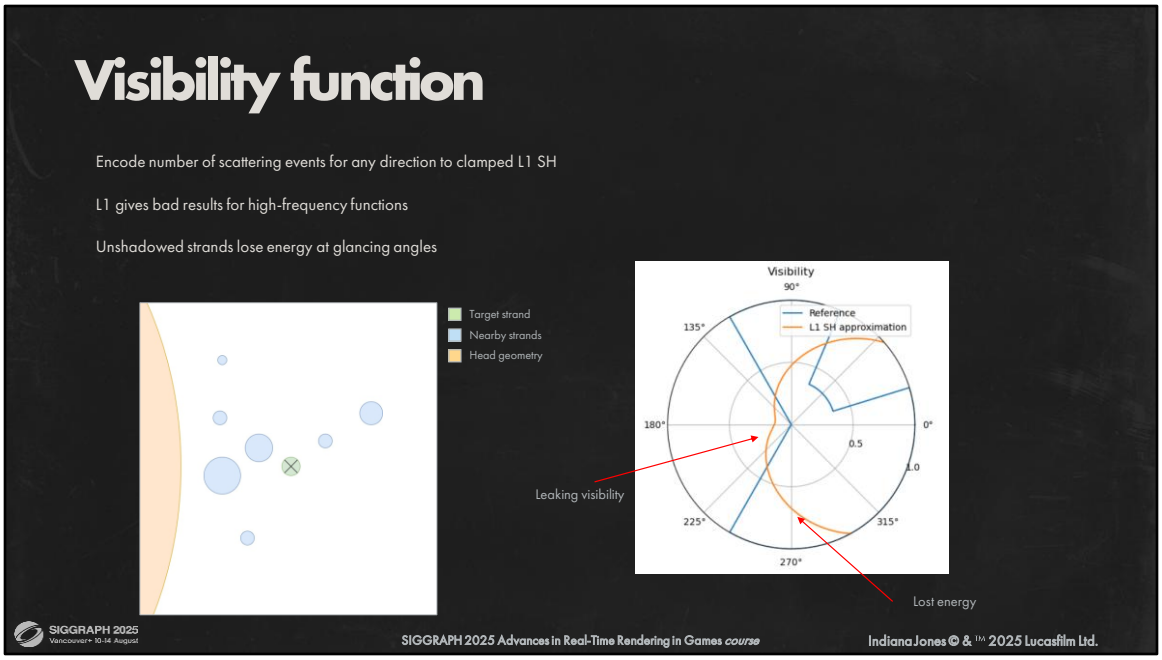



Here are some comparisons between volume raymarching and precomputed ray tracing. The results are much less flat and you can see proper shadow from the ear


Here are some comparisons between volume raymarching and precomputed ray tracing. The results are much less flat and you can see proper shadow from the ear


And some other grooms


You can see that self-occlusion is not a problem when we don’t use voxel grids




We use clamped low order SH as our visibility function because of integration simplicity, low memory requirements and lack of ringing artifacts.
Low-order SH can't capture high-frequency data very well, so instead of encoding visibility directly, we encode the average number of scattering events in any given direction.
Although, when using SH directly you can clearly see on the polar plot on the right that we lose high-frequency angular details. It causes light leaking and energy loss.

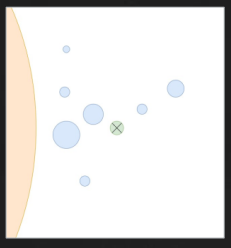
# Visibility function

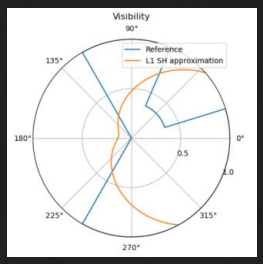
Bias ray misses to have negative number of scattering events when integrating SH, then clamp the result

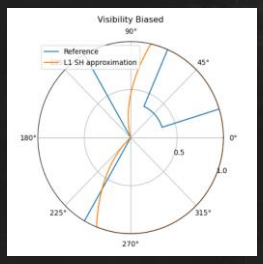
Gives better match to reference functions

Shipped with HAIR_MAX_SCATTER_EVENTS = 5

```
// When baking
float numScatterEvents = CastRay( rayOrigin, rayDirection );
float zeroEventBias = -HAIR_MAX_SCATTER_EVENTS;
if ( numScatterEvents == 0 ) {
    numScatterEvents = zeroEventBias;
}
visSH += EvaluateSHBasisL1( rayDirection ) * numScatterEvents;

// When fetching visibility
float numScatterEvents = max( 0, ProjectL1SH( visSH, lightDirection ) );
float visibility = exp( -absorption, numScatterEvents ); // Simplified
```



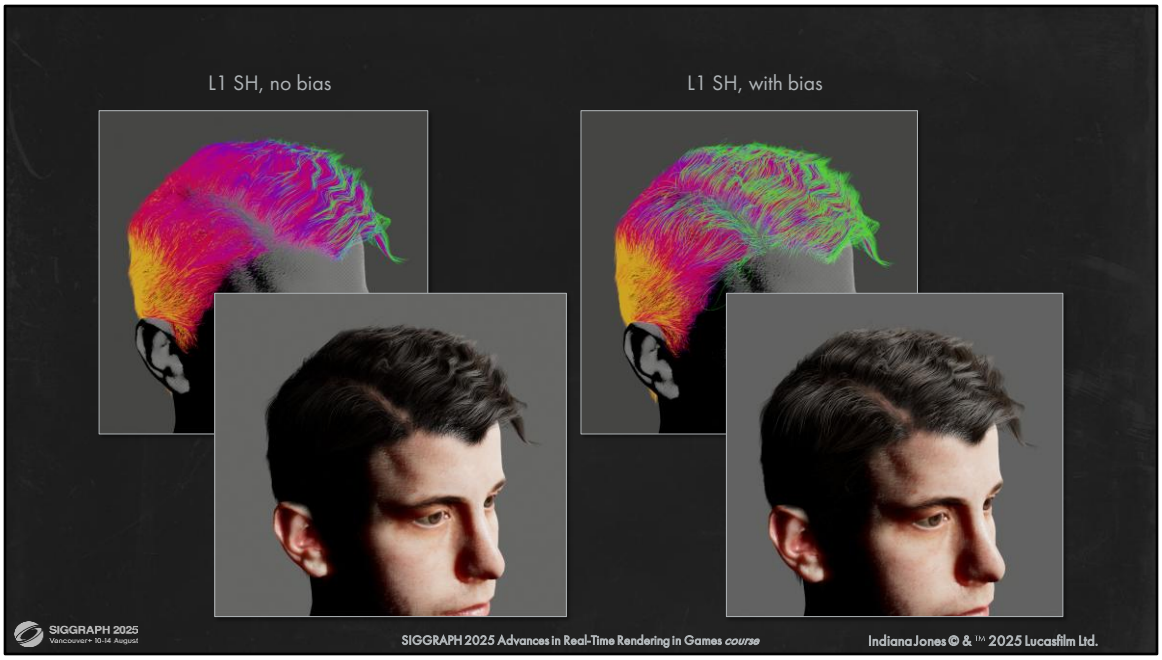



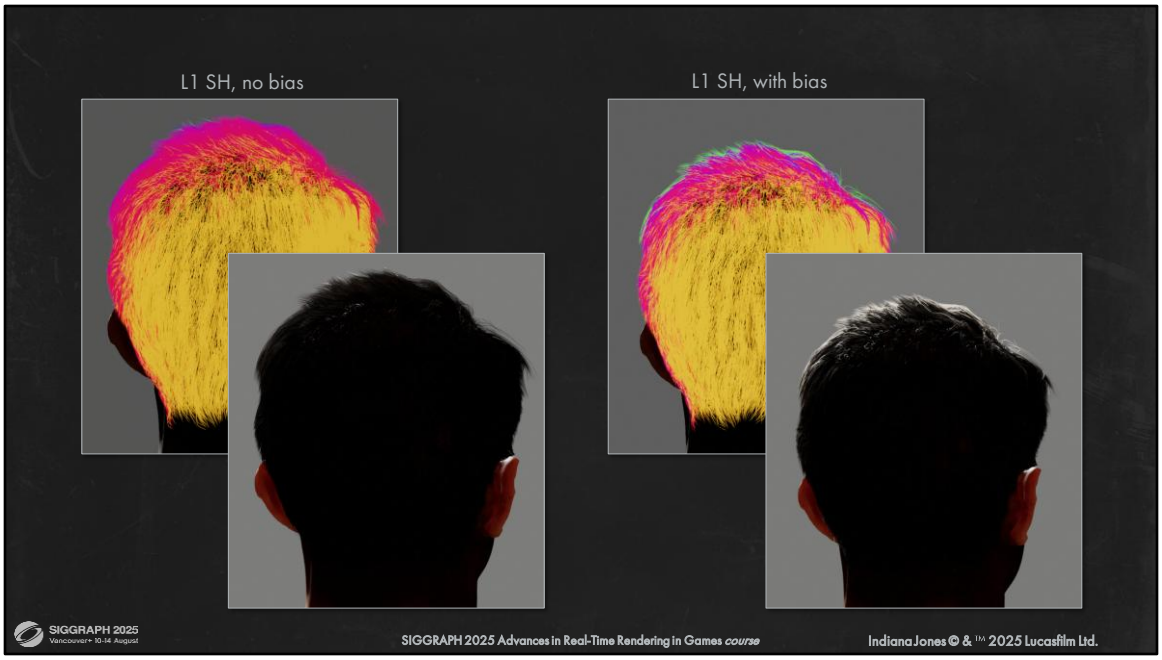




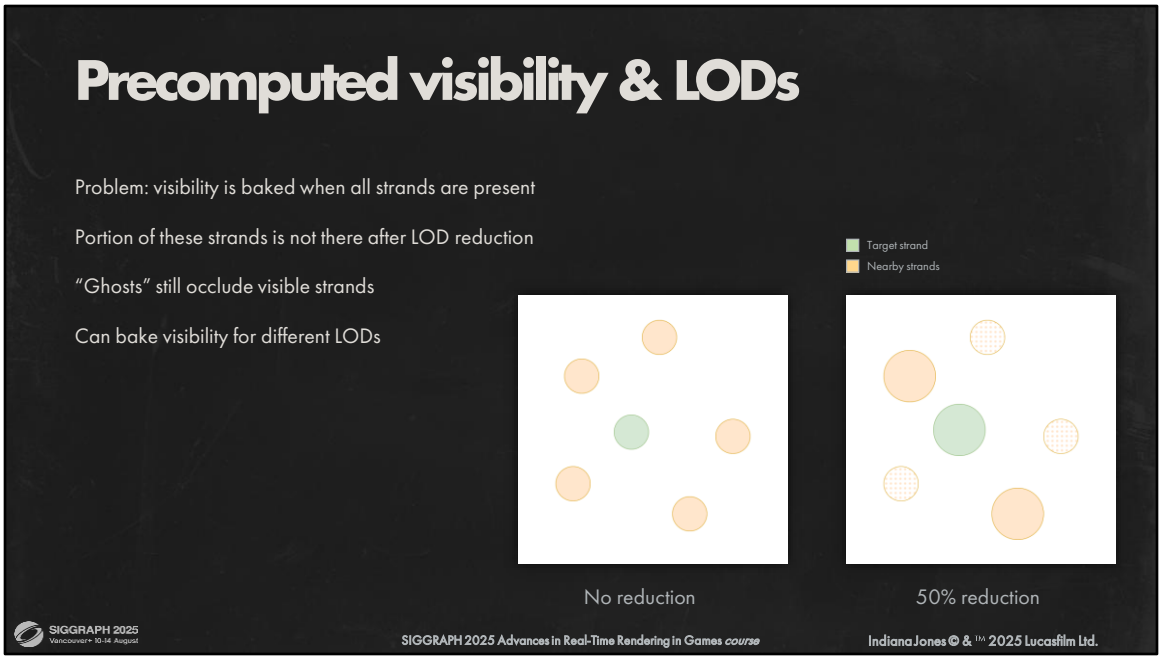
We combat this by biasing rays that did not hit anything. It helps us avoid overshadowing on glancing angles and get sharper transition. Otherwise backlit hair lose most of their "glow".



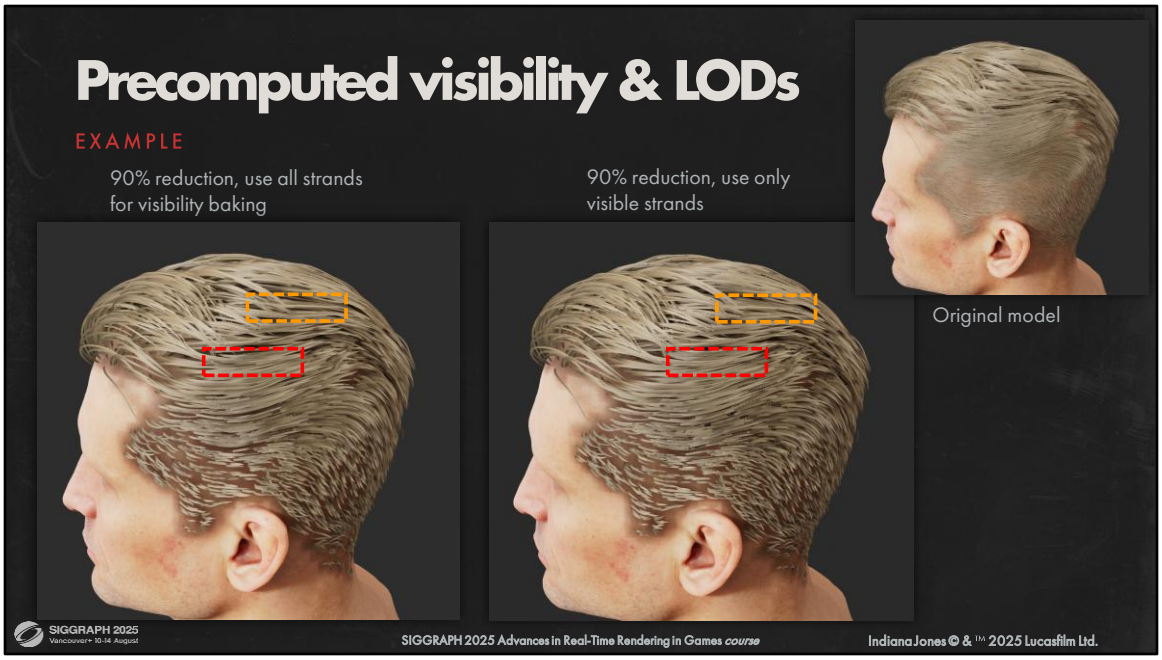

Here you can see how it looks on actual content. Using biases gives much more correct image and contour backlit lighting is not lost




For future work it would be interesting to experiment with other spherical functions (maybe learned functions?) and dynamic recalculation for visibility, so we could handle strong winds and very long hair better.




One other peculiar issue we have with precomputed visibility is its interaction with LOD system. We do LODs by taking a portion of strands from the original model directly, that applies to precomputed visibility as well. But this data was computed for unreduced model and contains collision with strands that were removed by LOD system. Ideally we would need to rebake visibility for every possible LOD, but it would take too much memory, because we have as many LODs as we have strands. That is why doing classic fixed-step LODs might be a better option.




But in practice the artifacts weren't too visible for us so we kept the original LOD system. You can see some difference if you squint, but nothing serious

# Hair volumetric shadows

Raymarch hair density volume for each skin pixel for each light

6 steps per light, 1 ray per pixel, 48x48x48 grid resolution

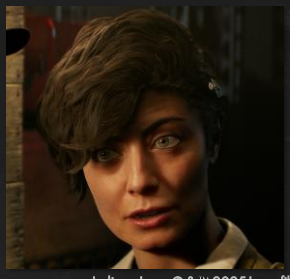
Also handles ambient occlusion (1 ray with 2 steps)

Artists can control shadow intensity per asset

~0.3ms on Series S in cutscenes (for both AO and direct shadows)

No hair shadows

Volumetric hair shadows






Now that self shadowing is taken care of, we need to calculate shadows from hair on other objects. For skin pixels we use volumetric raymarched shadows, like one of the approaches we tried for hair self shadowing. One extra bonus is that we can get ambient and specular occlusion with the same method. It is fast enough when you only do that for opaque pixels.

# Hair volumetric shadows

Pros

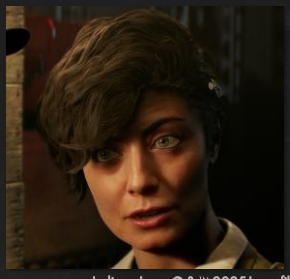
- No rasterization
- Scales with screen resolution
- Can enable selectively
- Nice diffused look

Cons

- Skinning to parent triangles not available
- Does not work well for animals
- No shadows from hair on walls, floor, etc.

No hair shadows

Volumetric hair shadows






It worked very well for hair, when the shadow is usually naturally diffused. We don't need to rasterize strands into shadow maps, which saved some engineering efforts, performance scaling is acceptable. Unfortunately, the method is not perfect. Our density volume comes from simulation, so it only supports skinning to one head joint, and thus it can't be used for fur on animals, because animals bend ☺
Also, we still need shadows from hair on other objects, and we didn't want to make some kind of acceleration structure for hair volume intersections.

# Hair shadows

No hair shadow on the walls – breaks character silhouette

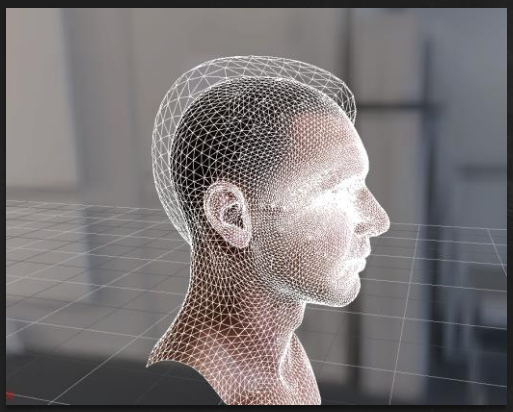



Usually missing hair shadow is not very noticeable. But it's not the case for the player character. You can see that missing shadow is very visible on the floor here, and we don't want to break the silhouette.

# Hair shadows

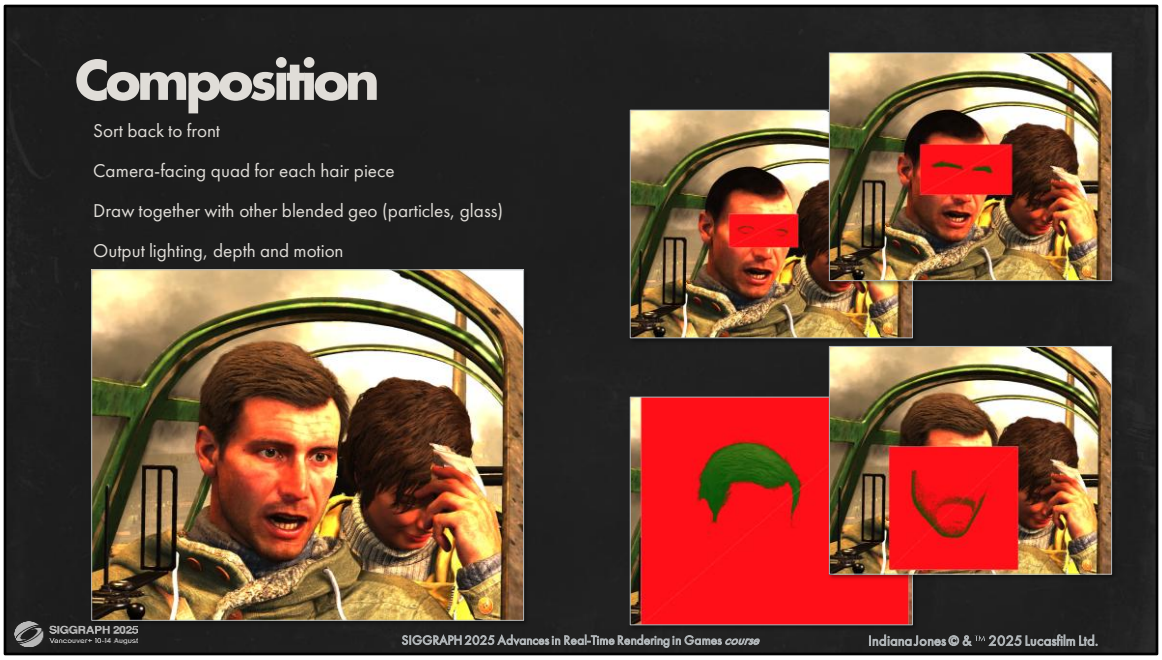
Use simple sculpted shadow mesh proxy for player character

Invert faces to avoid double shadowing on skin

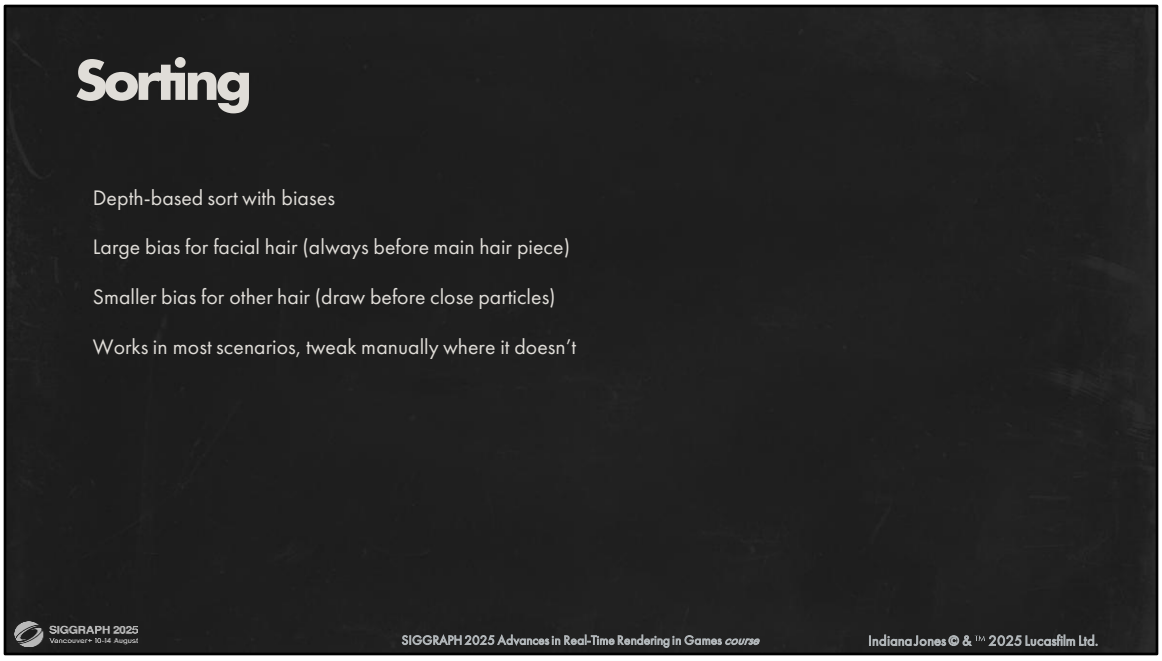




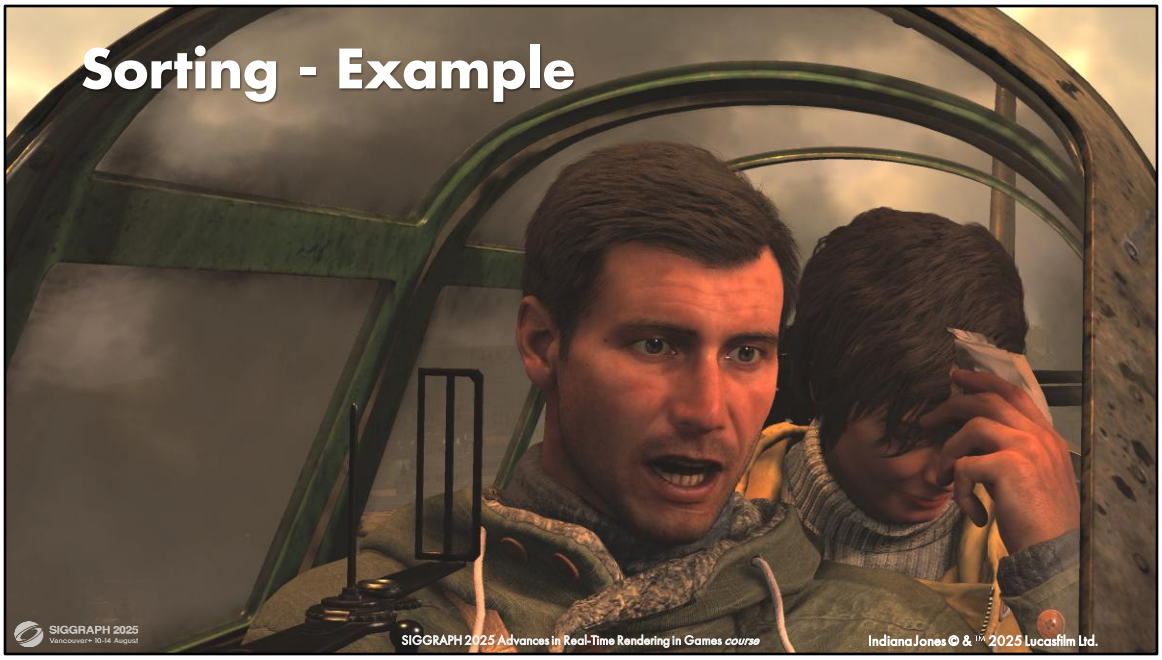

For this case we did a very simple sculpted shadow proxy with inverted faces, that we draw into shadow maps. It's very fast to render and very simple for artists to make.




Now that we have our visibility and shading information for each hair vertex, we need to compose it to the final image.
Since each hair model has its own space in the visibility atlas, we must compose each model separately. We do that by drawing a camera-facing quad for each model.
This is done together with other blended geometry in back-to-front order. It allows us to handle scenes with complex transparency setups in a unified way.




Sometimes that might be a source of bugs, if the sorting is not correct. We have a few heuristics to help with that.
- We always try to draw facial hair before the main hair piece
- We bias all hair by ~0.5 meters when we sort objects, so smoke particles with soft blending are always drawn after hair when they are close together
- We write hair to depth as we render it (so particle soft blending works). It helps us to hide a couple of places where sorting has gone wrong,
but we still had some less noticeable artifacts when sorting wasn't correct




We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

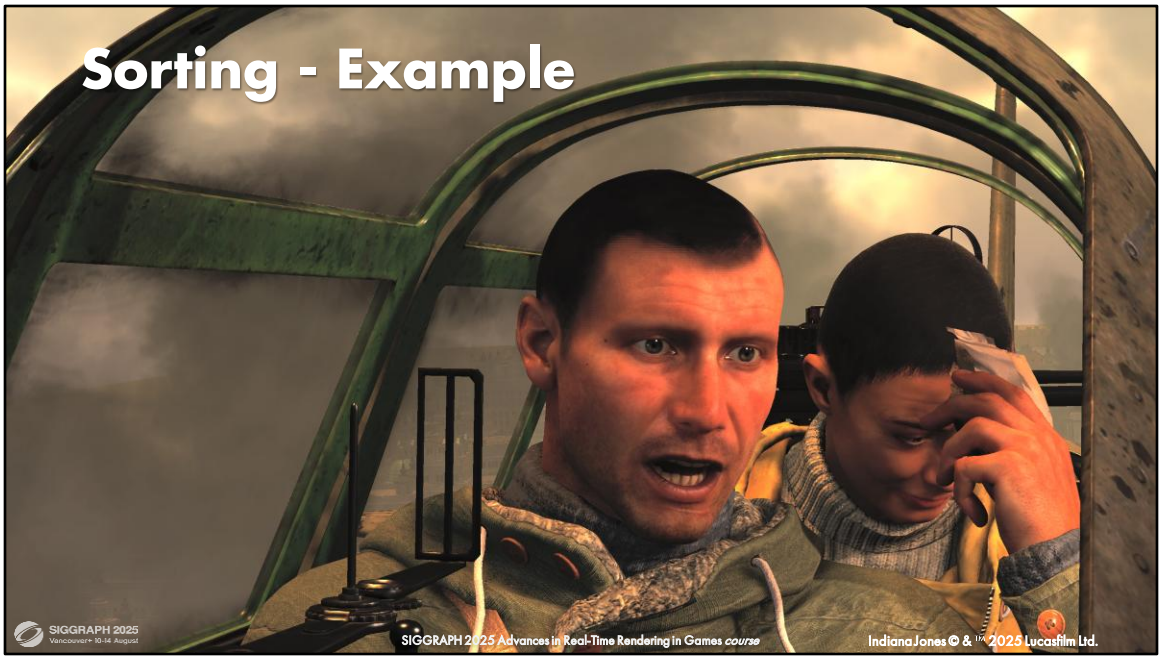
We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.


We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.


We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.




We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.


We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

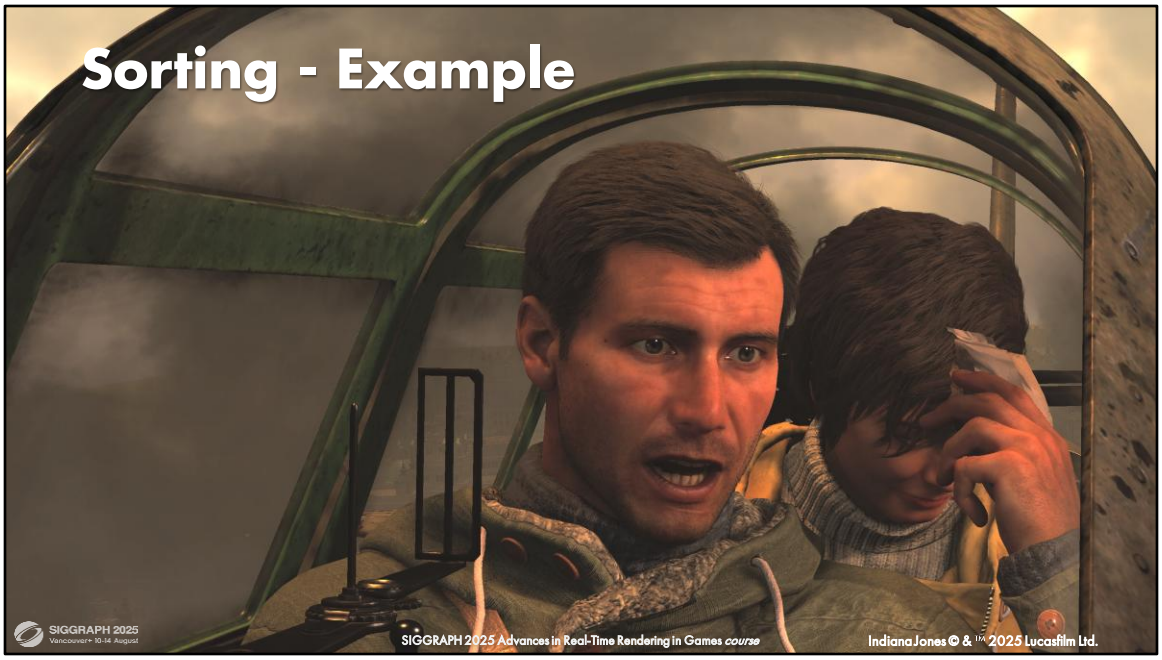
We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.


We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.

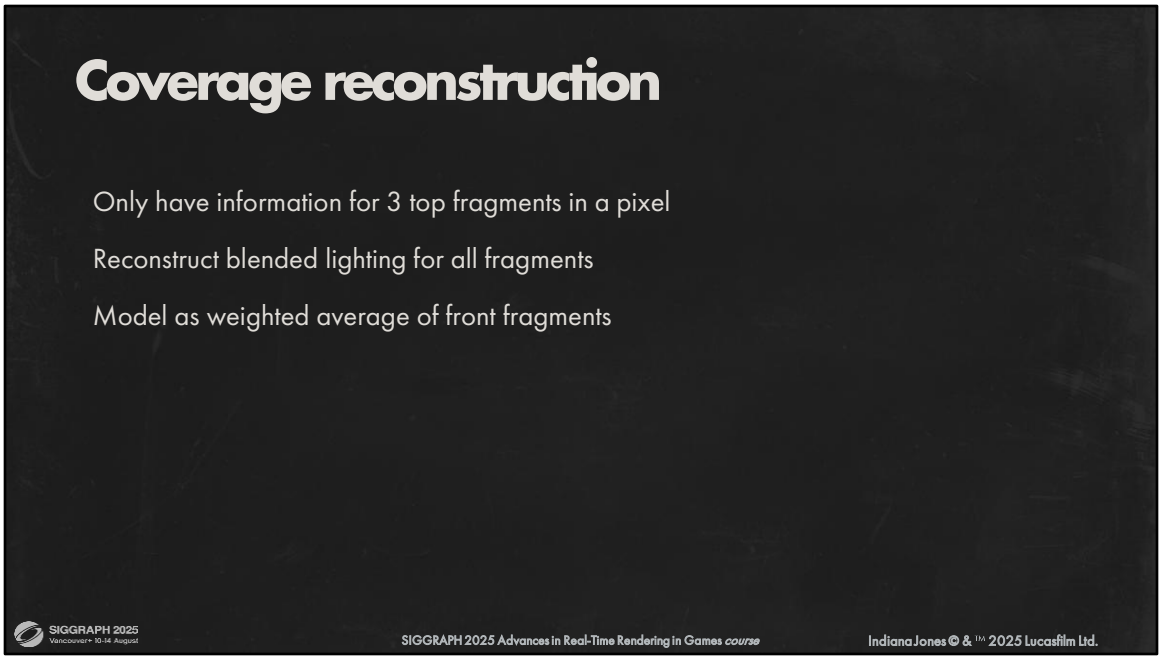

We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.




We didn't need to do a lot of manual tweaking, most setups worked out of the box. But some setups were complex and needed some fiddling with sorting biases.
For example, take a look at this shot. Here we have a setup that was a bit problematic for us. Lots of overlapping transparencies and hair that is tricky to sort correctly.
- First we render smoke behind the plane
- Then its glass behind our characters
- Gina’s hair
- Indy’s eyelashes and eyebrows
- Main hair pieces
- Front glass

We had to tweak sorting biases on the glass and render front faces and back faces separately to draw everything in the correct order.




Let's look at how we composite one single hair piece. As you might remember from the previous section, we only store 3 front segments and total coverage for each hair pixel. And we need to reconstruct the combined lighting for all hair fragments

# Coverage reconstruction

Calculate coverage from front 3 samples

Take $C_{total}$ from visibility atlas

Approximate lighting for missing coverage from front samples

Favor samples further from camera

Calc front samples lighting with simple alpha blend

$$C_{front} = 1 - (1 - C_1)(1 - C_2)(1 - C_3)$$

$$C_{total} = C_{front} + C_{missing}$$

$$L_{total} = L_{front} + L_{missing}$$

$$L_{missing} = C_{missing} \frac{\sum_{i=1}^{3} w_i L_i}{\sum_{i=1}^{3} w_i}$$

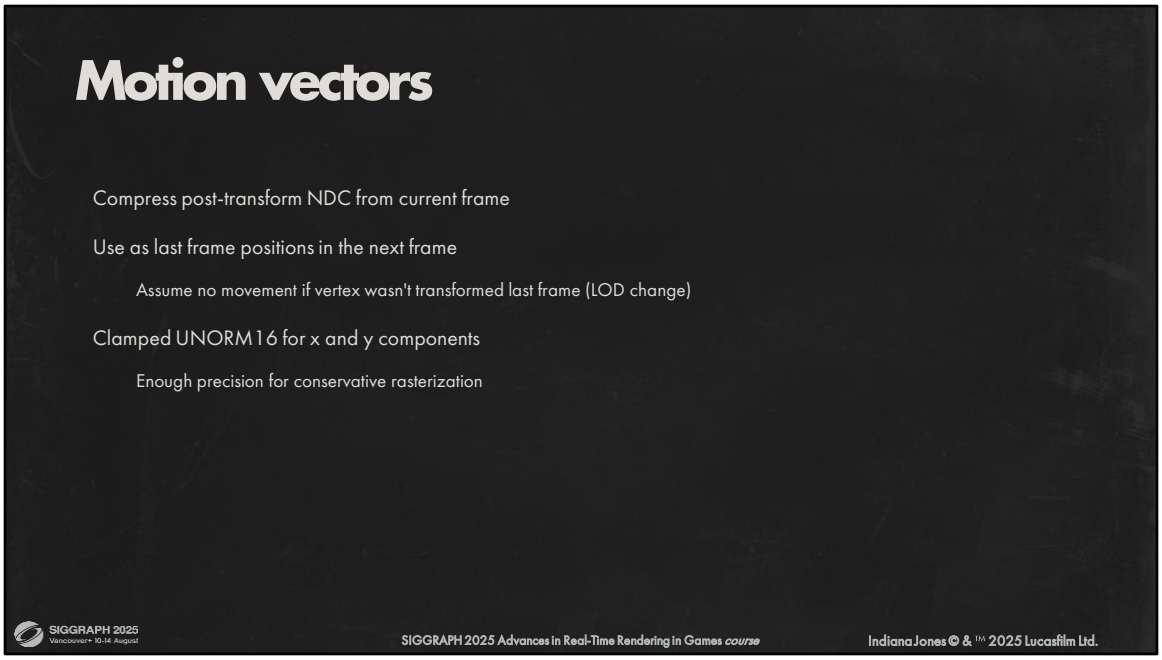
$$w_i = C_i * 0.25^{3-i}$$



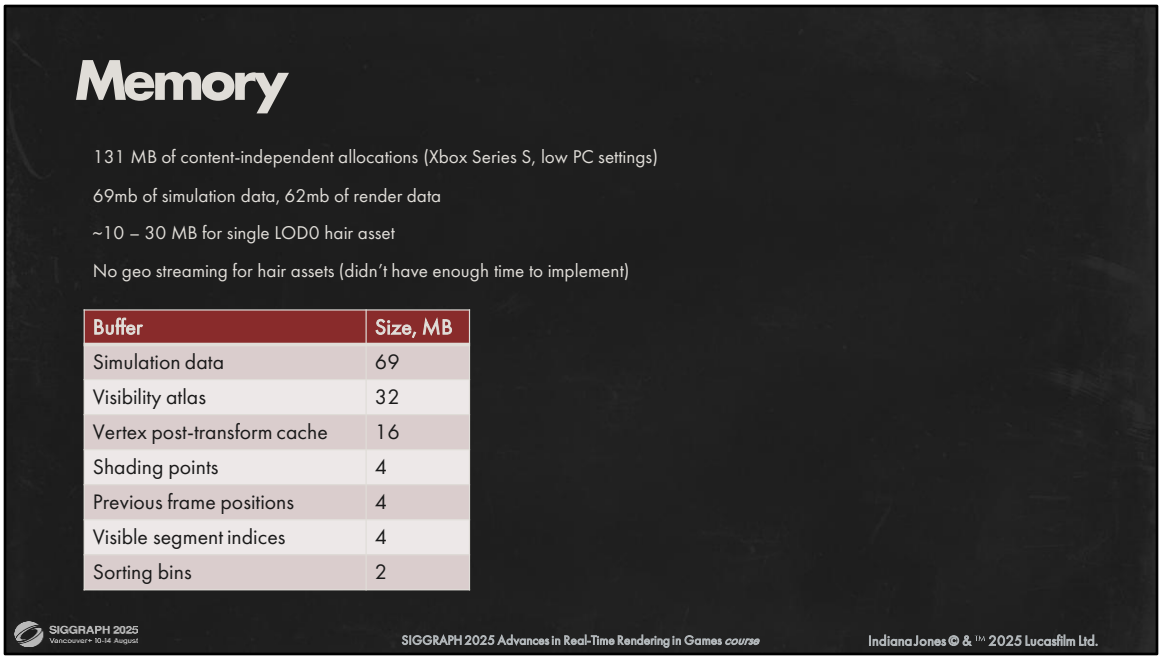

We model the coverage from the first 3 samples as a regular alpha-blended coverage,
We also have our total coverage value from visibility atlas. We model the radiance from missing part as weighted sum of existing samples.

We favor samples further away from the camera because they are closer to samples that we want to reconstruct.

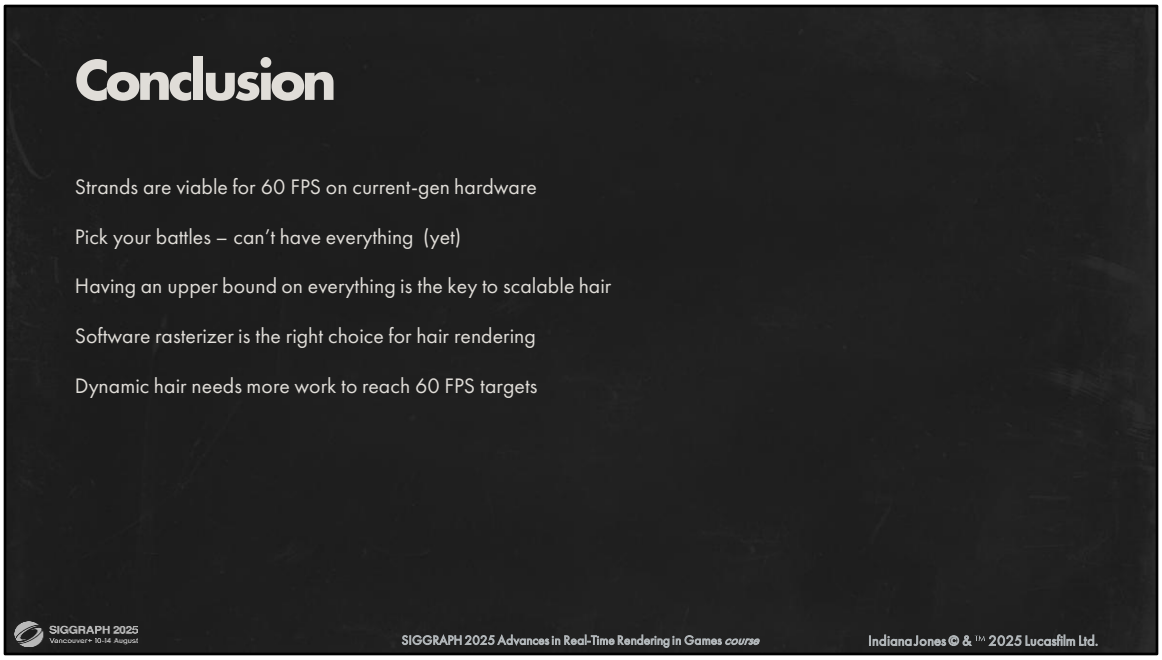

This gives us a good appearance up close, where usually 3 front samples is enough, and at the distance, where more lighting information is reconstructed.




Last, but not least, motion vectors. Having good motion vectors is crucial for upscalers, TAA and motion blur to work, so we wanted them to be perfect.
Having a global post-transform vertex cache comes in handy here. What we do is we compress the cache we have and reuse it in the next frame to calculate perfect motion vectors.
Consoles and non-pathtraced PC version use x and y components of post-transform hair vertex positions, packed to 2 bytes each. It gives good results with conservative raster that we use.
However, if your rasterization is not conservative, you rely on TAA to reconstruct the hair and you can afford extra couple of megabytes, I would recommend using full precision for this, it noticeably improves image clarity.




And here you can see how much memory we need for our low spec. For high spec we allow twice more strands, so its 90 megabytes of render data instead of 62. We didn't have time to implement hair streaming for assets, so resource-dependent memory load depends on the scene.
For future work we would like to reduce simulation requirements and implement asset streaming.




On that note I want to summarize what we have learned from our time developing strand hair. The main point, in my opinion, is that strand hair is fast enough now to be the primary solution for hair on current gen hardware. But you need to know your target and pick your battles accordingly. Also having a well-defined worst case scenario for all parts of the system helps tremendously, because otherwise content will break all your limits eventually.
Software rasterizer was the right choice for strands. It takes considerable amount of time to get it running fast, but the result is well worth it.
And lastly, we need more work on the optimization front for dynamic hair, right now you have to pick between performance and better hair movement.
Thank you!

# References


[Taillandier20] – Robin Taillandier, Jon Valdes (2020). **"Every Strand Counts: Physics and Rendering Behind Frostbite's Hair"**

[Müller12] – Matthias Müller, Tae Kim, Nuttapong Chentanez (2012). **"Fast Simulation of Inextensible Hair and Fur"**

[Ishihara23] - Toshiaki Ishihara, Hitoshi Mishima (2023). "**Resident Evil 4 Hair Discussion**"

[Marschner03] – Stephen R. Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, Pat Hanrahan (2003). **"Light Scattering from Human Hair Fibers"**

[Karis16] – Brian Karis (2016). **"Physically Based Hair Shading in Unreal"**

[Tafuri19] – Sebastian Tafuri (2019). **"Strand-based Hair Rendering in Frostbite"**

[Yuksel08] – Cem Yuksel, John Keyser (2008). **"Deep Opacity Maps"**

[Jansson19] – Erik Jansson, Matthäus Chajdas, Jason Lacroix, Ingemar Ragnemalm (2019). **"Real-Time Hybrid Hair Rendering"**

[Gerard23] – Martin Gerard (2023). **"Fast Real-Time Shading for Polygonal Hair"**

[Ciardi22] – Francesco Cifariello Ciardi, Lasse Jon Fuglsang Pedersen, John Parsaie (2022). **"Probe-based lighting, strand-based hair system, and physical hair shading in Unity's 'Enemies'"**

Matt Jen-Yuan Chiang, Benedikt Bitterli, Chuck Tappan, Brent Burley (2015). **"A practical and controllable hair and fur model for production path tracing"**

Arno Zinke, Cem Yuksel, Andreas Weber, John Keyser (2008). **"Dual Scattering Approximation for Fast Multiple Scattering in Hair"**

Eugene d'Eon, Guillaume François, Martin Hill, Joe Letteri, Jean-Marie Aubry (2011). **"An Energy-Conserving Hair Reflectance Model"**