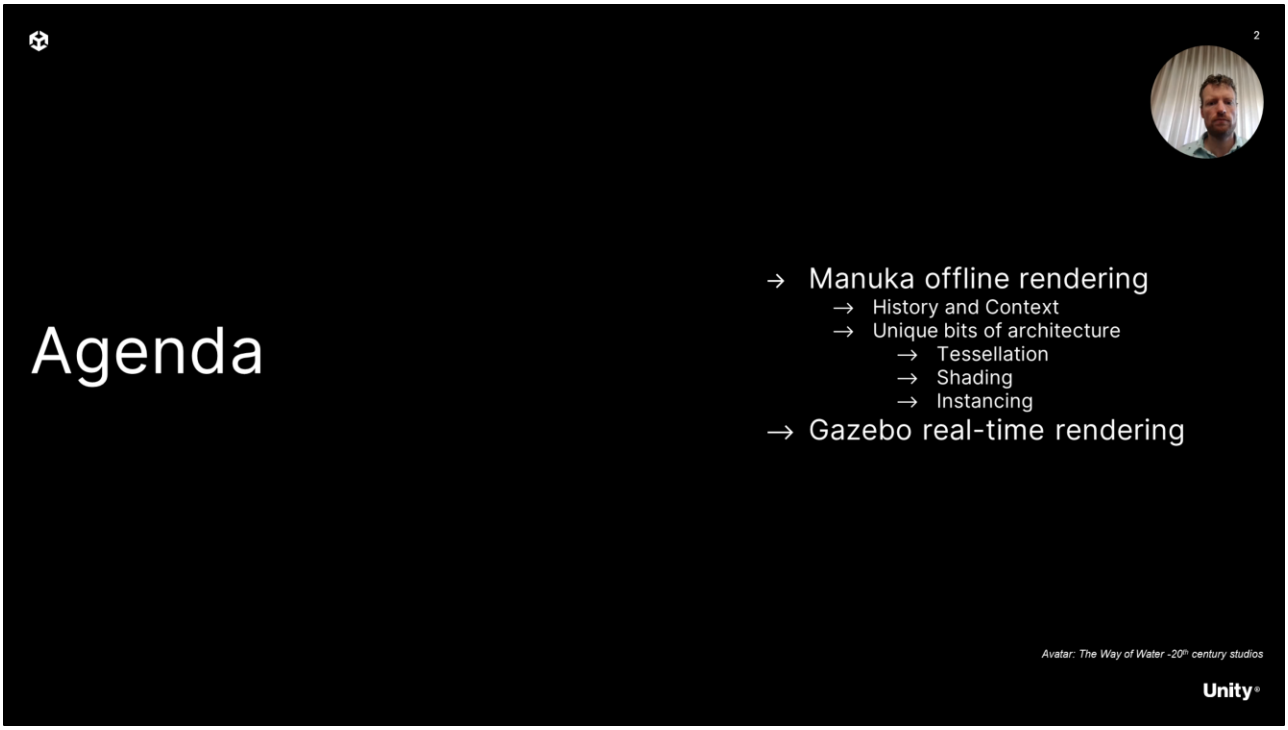
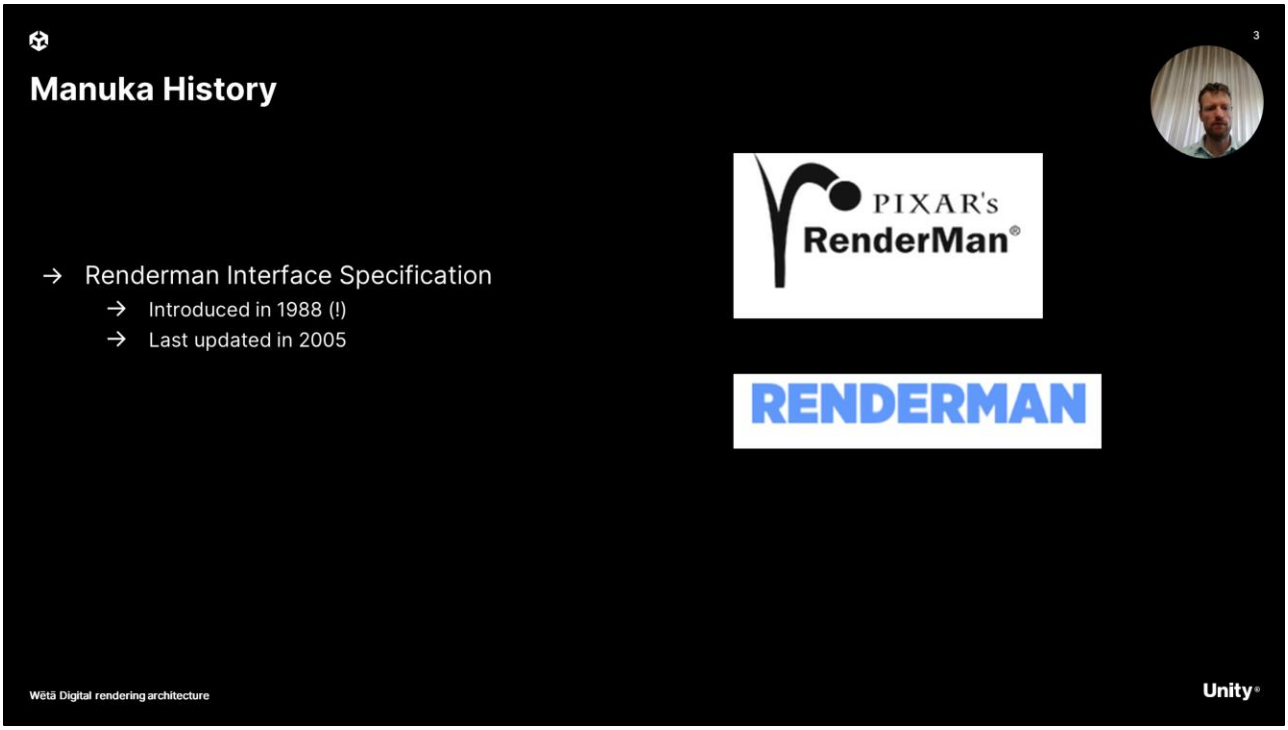
Hello and welcome. I am Sander van der Steen, one of the lead engineers on the Manuka renderer since 2022 and virtually with me is Robert Cannell, the lead engineer of the Gazebo renderer. Today we will explore our rendering architectures of the 2 in-house renders used at Weta Digital.



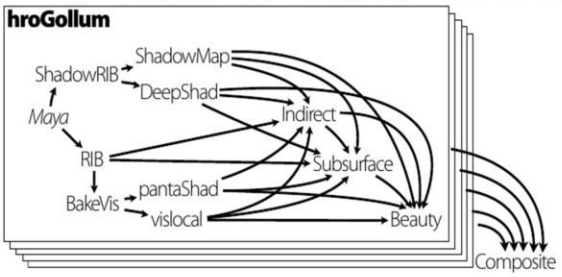
- That is right, this is 2 talks wrapped into one. First up is Manuka which is our offline, final frame rendering and the second renderer discussed is Gazebo, which is used on stage and also on artist workstations in order to provide interactive feedback
- For Manuka, I will briefly discuss the history before focusing on details of Manuka's architecture that stand out and might not be quite what you expect. In order to do so we will also leave other parts of the renderer uncovered. After discussing Manuka, I will hand over to Rob to discuss Gazebo in more detail and
- At end of this talk there should be time for a small Q&A where we can try and answer any further questions you have.



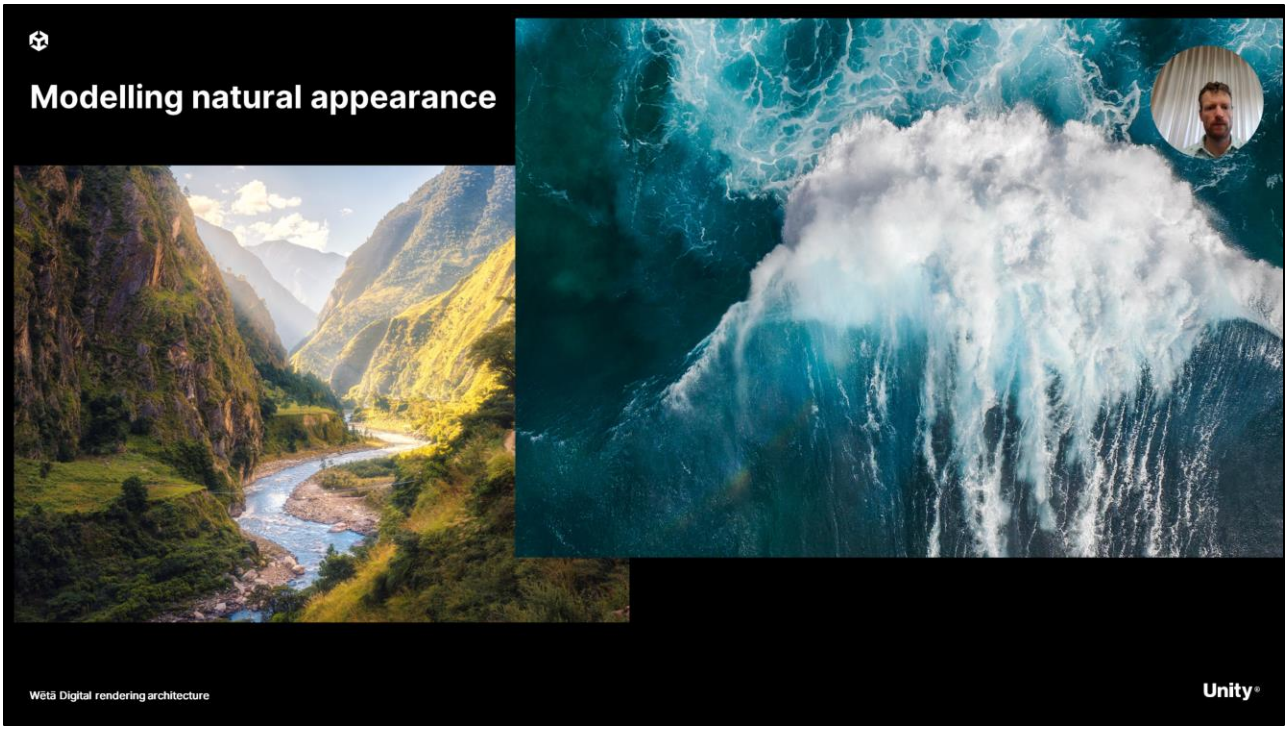
Manuka dates back about 10 years or so, and to understand the history of Manuka we first need to quickly touch on Pixar's renderman, which 10 years ago was sporting the top logo. Pixar's renderman, or PRMan in short was and still is one of the most widely used offline renderers for film production today. PRman is what is called a "Renderman compliant renderer" which means it adheres to the Renderman interface specification from 1988. There is a lot that we can say about this, but what is important here, is that it basically means that the renderer ingests data using a "open" API or scene description files called RIB.

Manuka – Design Philosophy

- → PRMan was “difficult” in our pipeline
  - → Lots of maps/baking etc.
  - → Manuka: A batch-shading architecture for spectral path tracing in movie production
  - → https://dl.acm.org/doi/10.1145/318216
- → Make our own! (“renderman compliant”)
  - → Easy pipeline integration
  - → Research alongside production (spectral appearance, LightTransport)
  - → Whole scene at once, uncompromised visuals



- Weta 10 years ago was also using Prman as a renderer. This means that our pipeline was geared towards delivering these RIB files.
- However, we were having scalability issues with PRMan. At the time, our RIB based pipeline required many bakes as “intermediate products”, and these all needed to have their dependencies tracked. There was a desire to have a renderer that “would just render the scene at once, no matter what”. There was also the desire for researches to implement the latest technologies directly, without bounds. Weta has always had a strong academic connection.

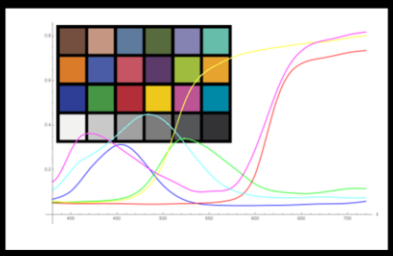


In other words, we wanted to render reality with a minimum of trickery which meant modeling natural appearance.

Manuka – Spectral rendering

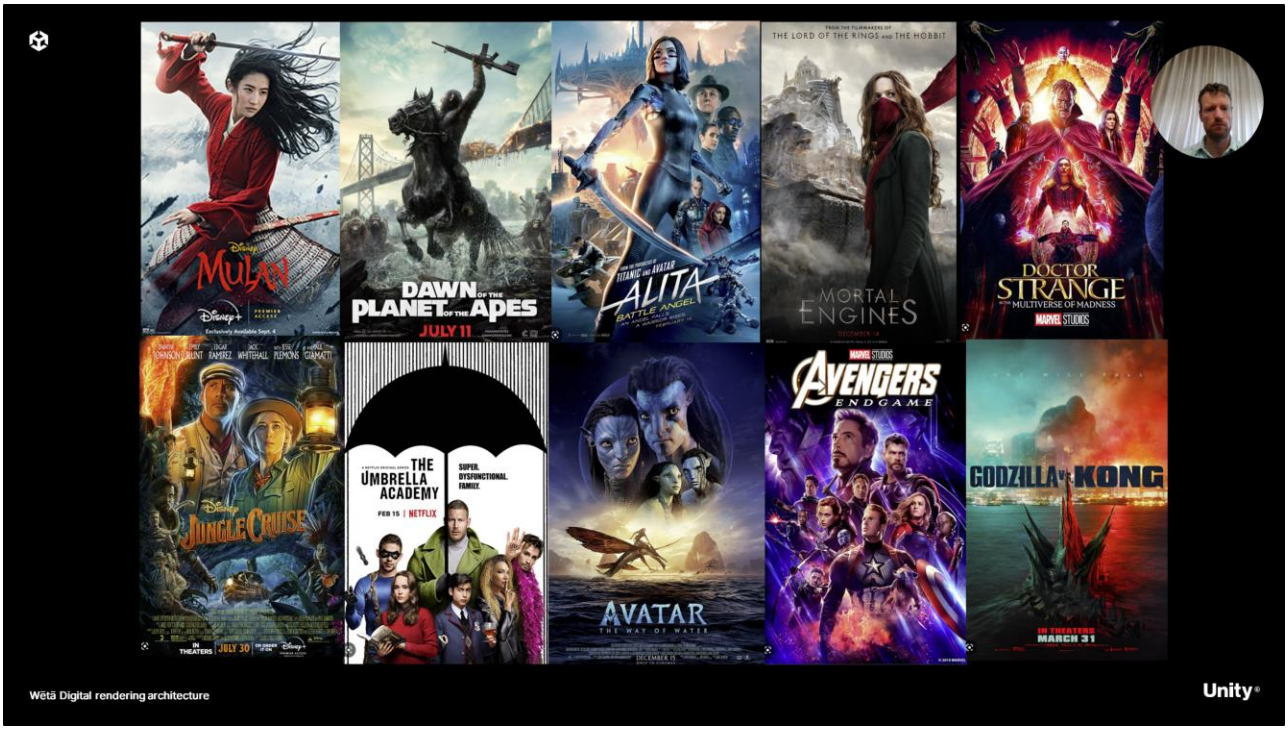
- → Color as a power distribution
- → Improved color accuracy
- → Want to know more about spectral rendering?
  - → Siggraph 2022 spectral rendering course
  - → https://dl.acm.org/doi/pdf/10.1145/3532720.3535632

Redacted image ☹

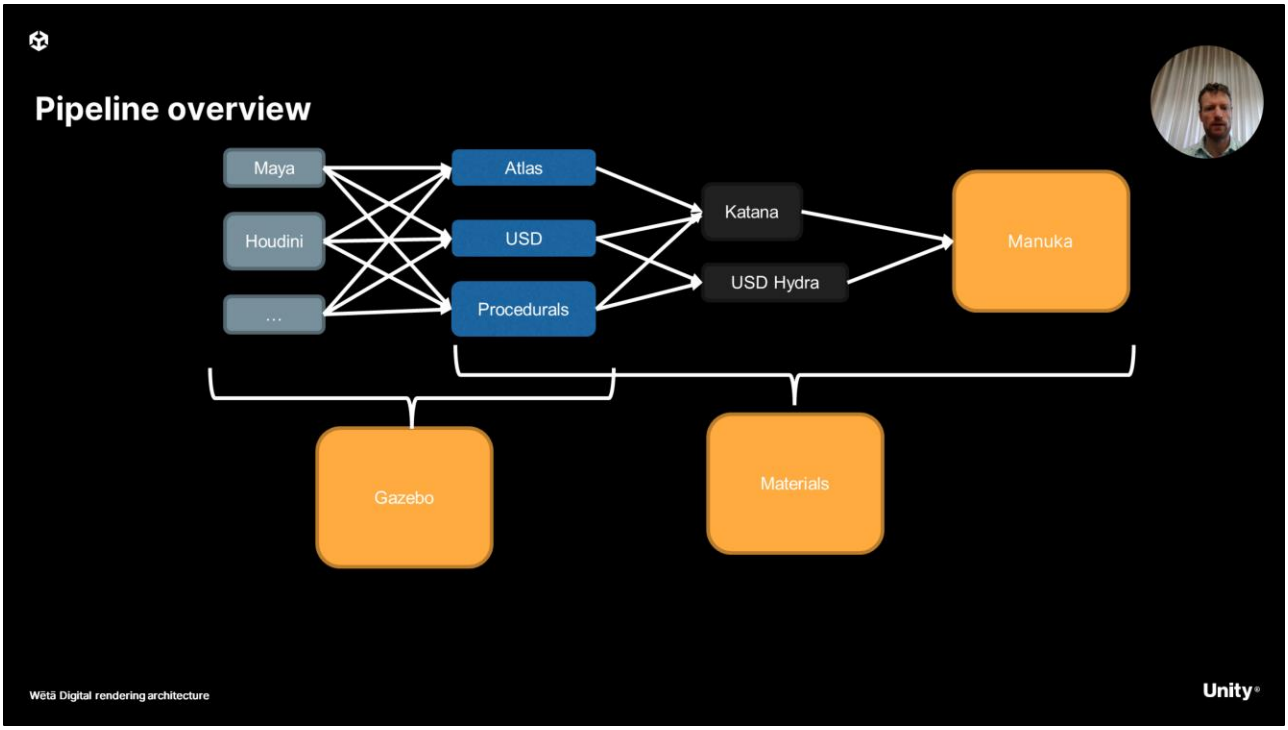


A key example of Manuka being a research renderer is the fact that Manuka is a spectral renderer. It is the first spectral renderer that was used in commercial VFX. Spectral rendering implies that we treat colour as spectral power distributions, as opposed to the normal three channel representation. This gives us improved colour accuracy.

As interesting as spectral rendering is, it is not the topic of this talk. And so if you want to know more about spectral rendering. I suggest you start with the excellent spectral rendering course done at Siggraph 2022 last year.



- Since making that decision on making our own renderer, Manuka has been successfully used in many VFX productions. I picked a few, but there are many many more, and there is plenty more in the pipeline!

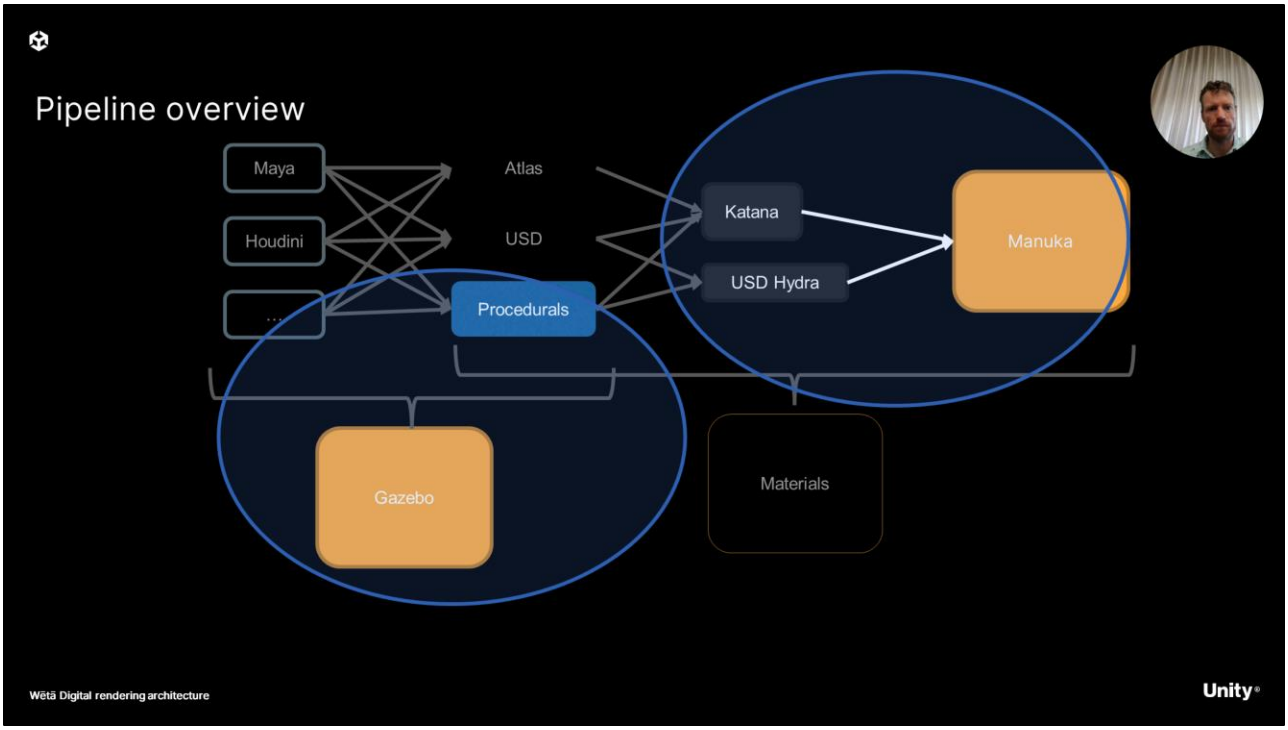


- So we are here to talk about rendering in movie production. To understand this a bit better, let's do a crash course into Weta's content creation pipeline. We start with the DCC's on the left, Maya & Houdini are the main ones but we have a wide range of other products in use at Weta.
- Those DCC's all produce assets (or data), which can be in the form of our proprietary scene description Atlas (which predates USD and is a topic in itself), or the increasingly common Universal Scene description.
- The final category of content are what we call "procedurals". They do data amplification at render time, generating renderable content typically as the content is ingested by the renderer. Think hair, trees, woven cloth etc.
- This data then gets to the renderer, Manuka. But this picture is not complete:
    - There is Gazebo which is used for interactive feedback and the Material authoring pipeline, both hooking into this web.

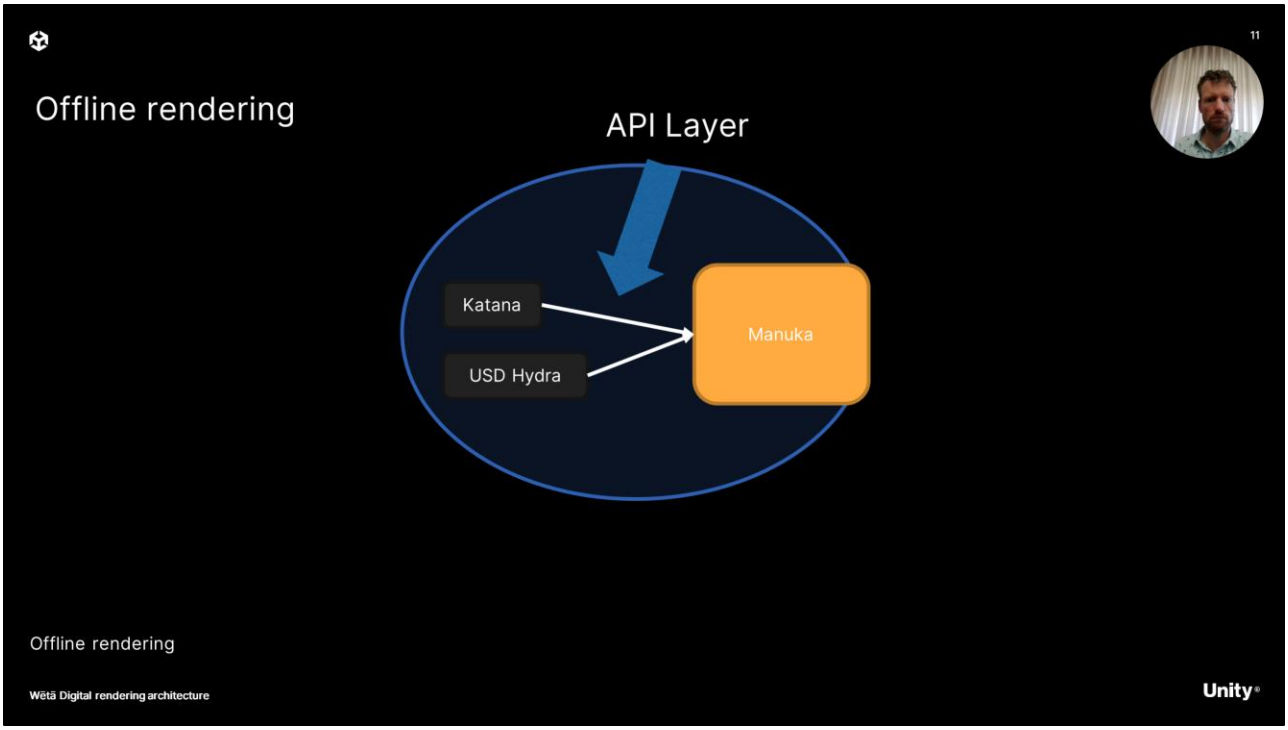
Pipeline reality is complex

- → Notes from the client can influence change across
- → Some pipelines intertwined in reality
  - → Example, animation on top of a boat in the ocean (sim/anim)

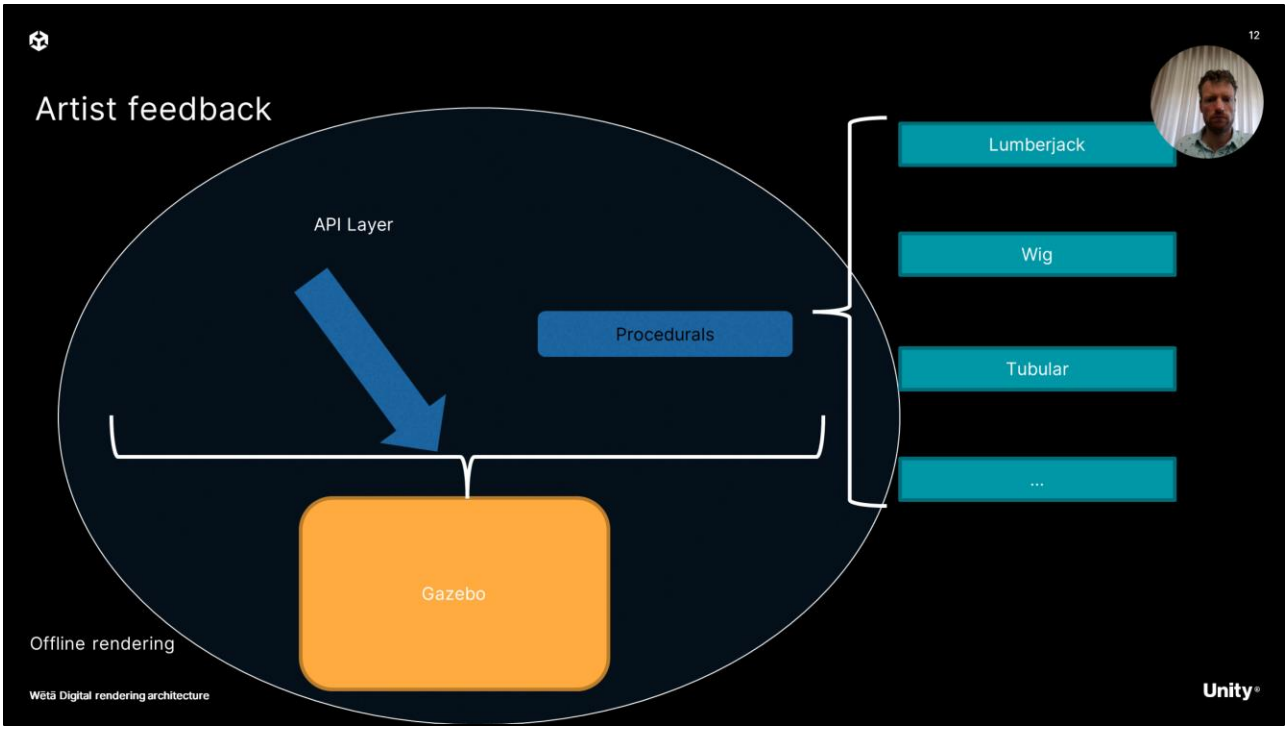
- In reality, a pipeline might as well look a bit more like this. Feedback notes from the director or studio can change anything anywhere back up the chain, causing version churn at any point in time. There is also work that can have an almost cyclic dependency. Think of animation on top of a boat in the ocean, hair interacting with water etc. The VFX pipeline, at least at Weta, is less linear compared to a games pipeline.



- Luckily we are not here to talk about complex pipelines, we are here to talk about rendering. But the diagram does highlight the 2 distinct areas that we will the focus for the rest of this presentation. There is final frame rendering in Manuka on the right and there is Gazebo at the bottom, combined with our procedurals, which is what drives artist experience at weta.

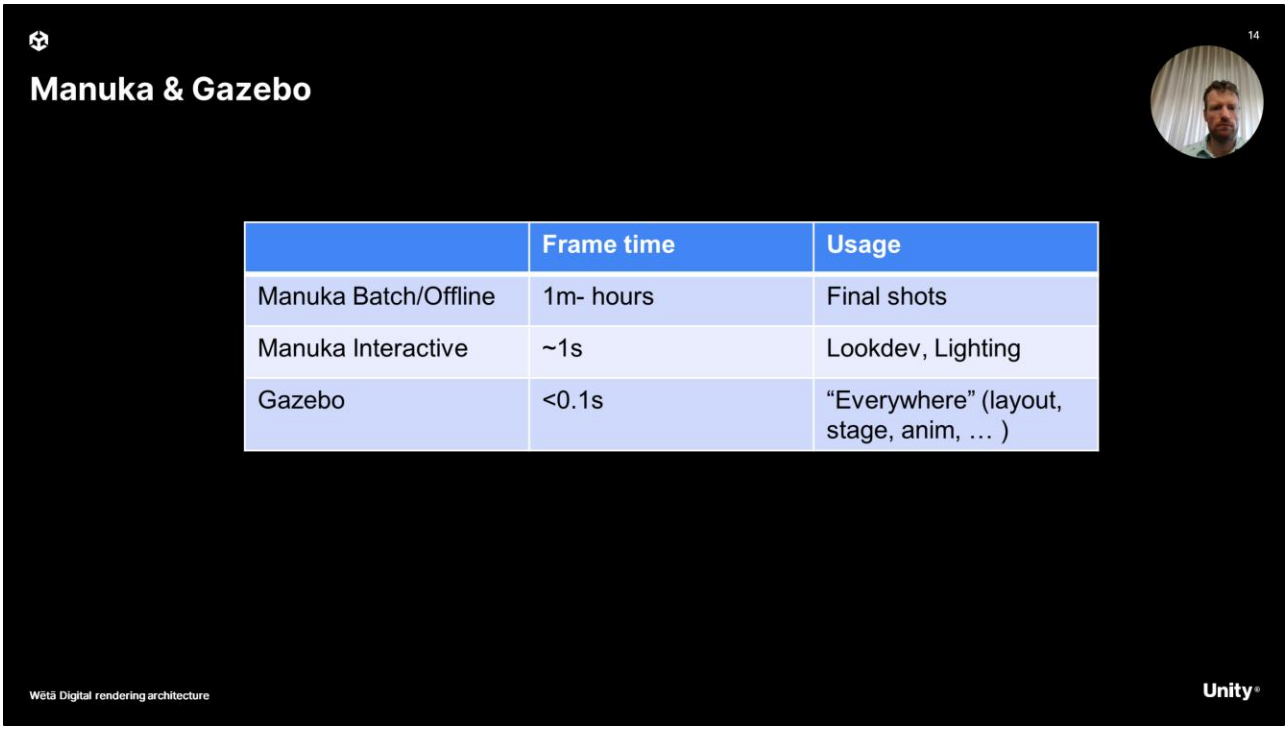


- The pipeline ultimately makes API calls into libmanuka.so, which is the Manuka renderer, loaded as a dynamic library.
- The API layers started life as being100% RISpec compliant, and for large parts we still are though there are places where we deviate or extend the system. One example is our implementation RSL, the renderman shading language. Manuka is spectral pathtracer, and there are differences as a result. Other changes exist in the interactive rendering space to perform live edits
- However, for Manuka we are primarily interested in non-interactive rendering in this section of the talk.



- Later on, things will get more interactive, where Gazebo is discussed. Gazebo provides that interactive preview that artists need to do their work. This is especially true for procedural geometry, which is based on custom data that is not trivially translated to polygonal geometry in the DCC's. The procedurals can generate data for the renderer directly, providing an accurate representation of the final frame in Gazebo. Robert will discuss how this works in more detail in the second part of this presentation

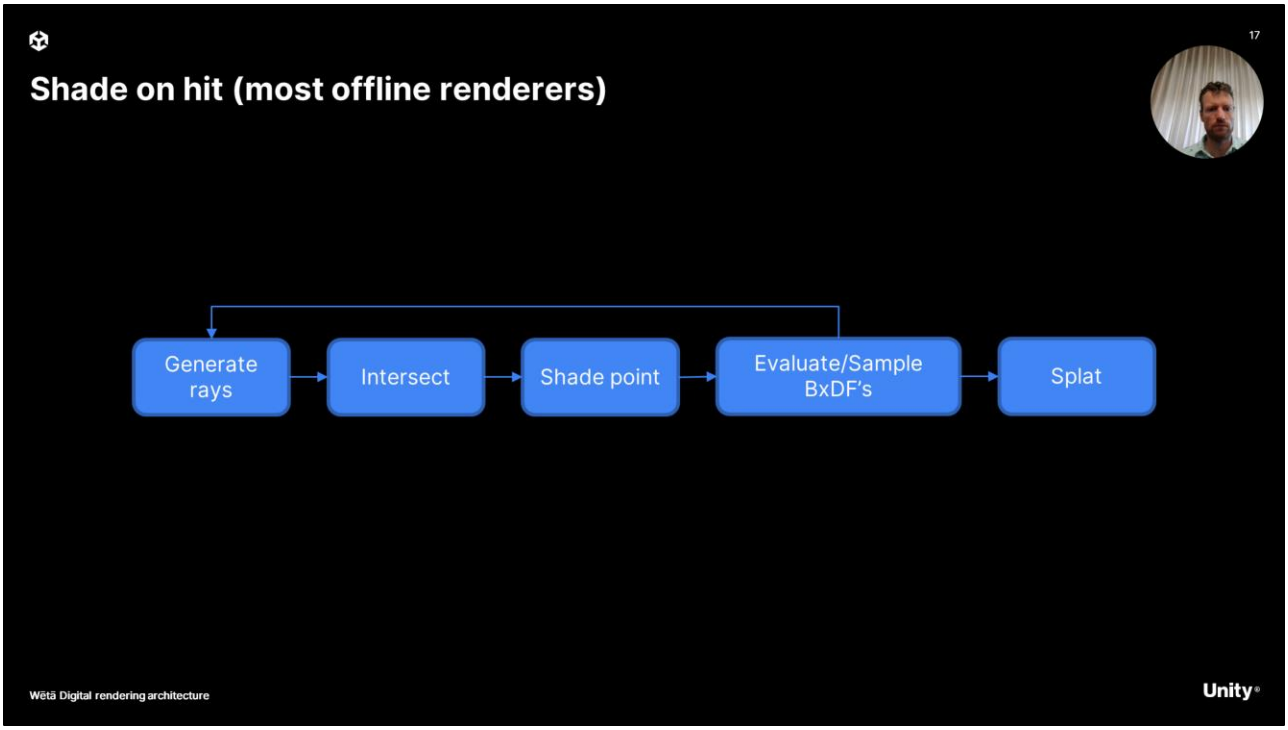
- To visually illustrate Gazebo and Manuka, this example shows a shot first rendered from inside the DCC in Gazebo as an artist might see it, and then rendered in Manuka as final frames.



- So the compare the renderers in a table. You can see the different uses here:
  - Gazebo is used everywhere, stage anim, layout fx etc. The frame time is typically < 0.1s or 10fps, though depending on the use, frame times can vary.
  - Manuka Interactive is a “interactive” version of Manuka, which I will only briefly cover at the end of the talk. It’s use is limited to lookdev and lighting and the frametime is roughly in seconds
  - And finally, there is Manuka Batch, or Manuka offline. Frame times are measured in minutes or even hours and it produces the highest quality images that are used as final frames.

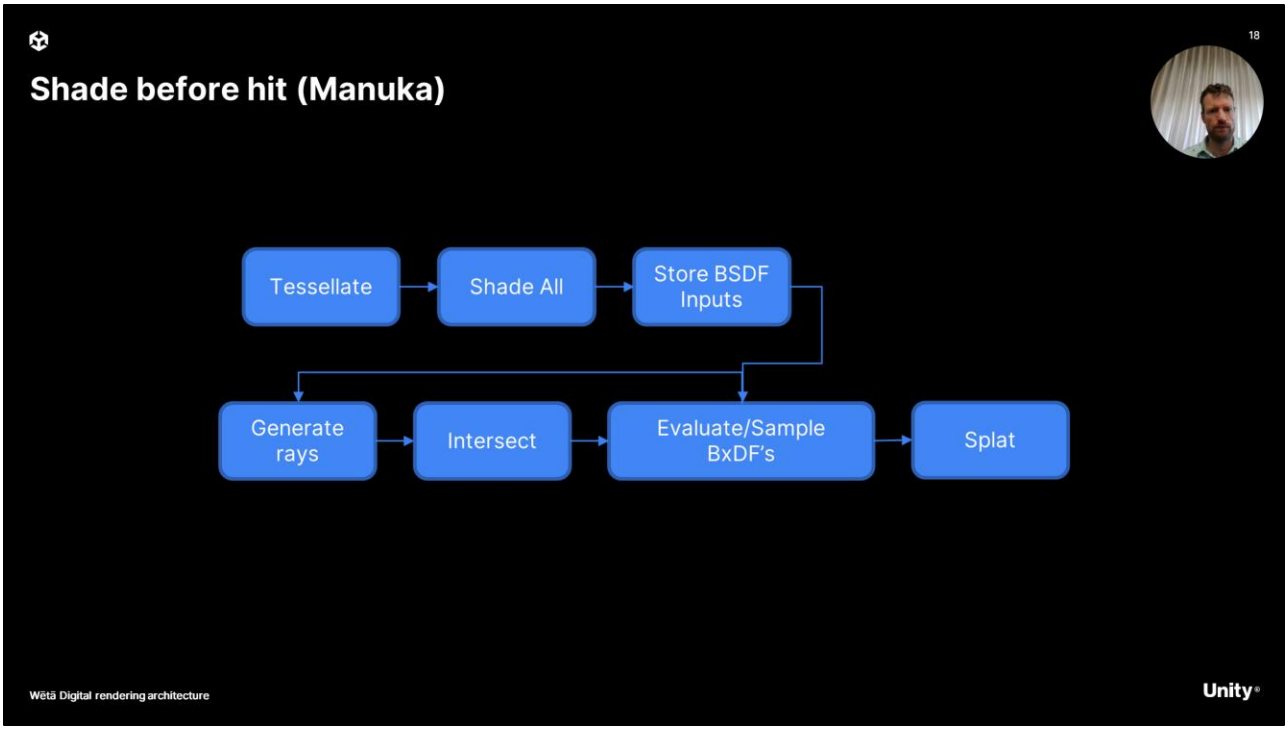
And with that, we can conclude our 10.000ft view of how our pipeline works at Weta Digital. Now let’s get a little closer to the action and zoom into rendering a bit more.

- To setup some boundaries for this talk, we will assume that all that pipeline stuff is covered and that, somehow (and we don't really care how) we have in memory available camera, settings, geometry, lights textures. What we are going to focus on next is unique parts on how Manuka performs rendering.



- One key architectural change in Manuka is the way shading is executed. Now while most if not everyone in the audience will know what shading and shaders are, the definition of a shader is somewhat lose and Manuka takes it's own spin on it due to the way shading execution is done.

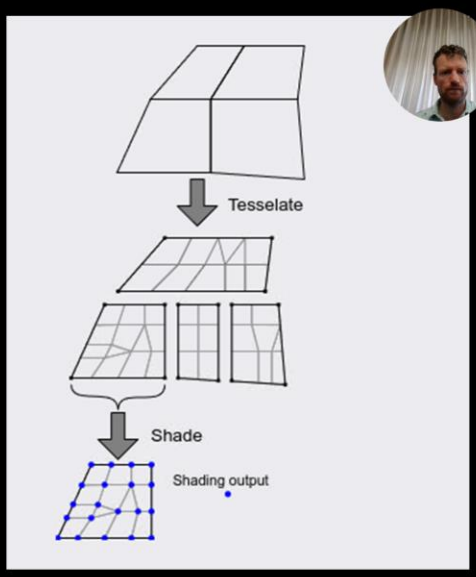
- Most renderers execute in what is known as a shade on hit system. Rays are generated, you run intersection on the geometry, you perform shading calculation, taking into account surface normal, lights, textures etc. You evaluate the bidirectional distribution function to produce color and splat that to your image. You may decide to generate more rays to a certain sampling threshold is reached. The more rays, the smoother your image, but the longer it takes.



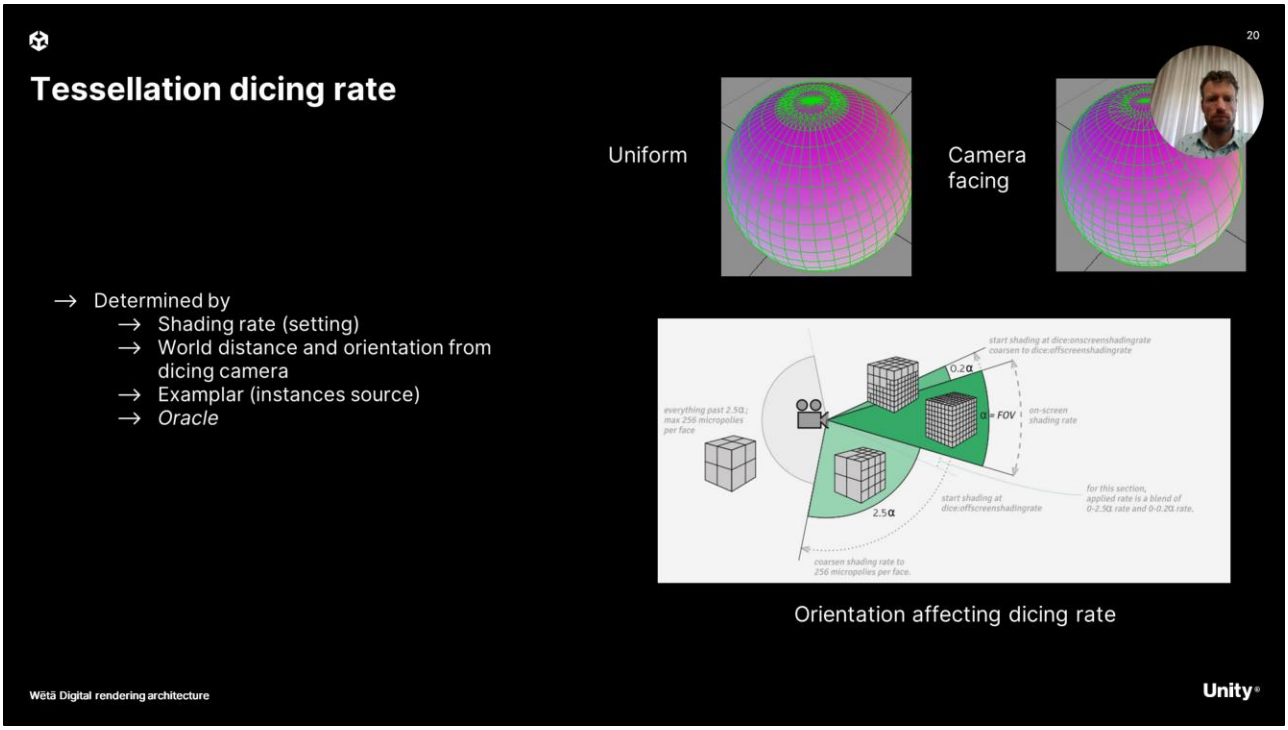
- Manuka does this differently; in that we don't start the generation of rays until we have done ALL the shader executions.
    - This may sound counterintuitive at first, but doing things this way gives us a few advantages which we will cover in more depth later in this talk, but a small spoiler, this is done for performance reasons.
    - Consequentlty shader execution in Manuka is not quite traditional as we don't have access to the light information, making BxDF evaluation impossible.
- So what we do instead is evaluate to a what we call a material, dealing with direction independent computations including texture sampling.
- Beyond shader execution, which reauires more context and information that we will provide later, we also need to determine at what frequency or density we want to evaluate those shaders.
- This is decided as part of the tessellation phase and let's cover this next.

Tessellation

→ Tessellation takes the input mesh and subdivides
→ It produces *micropolygons*



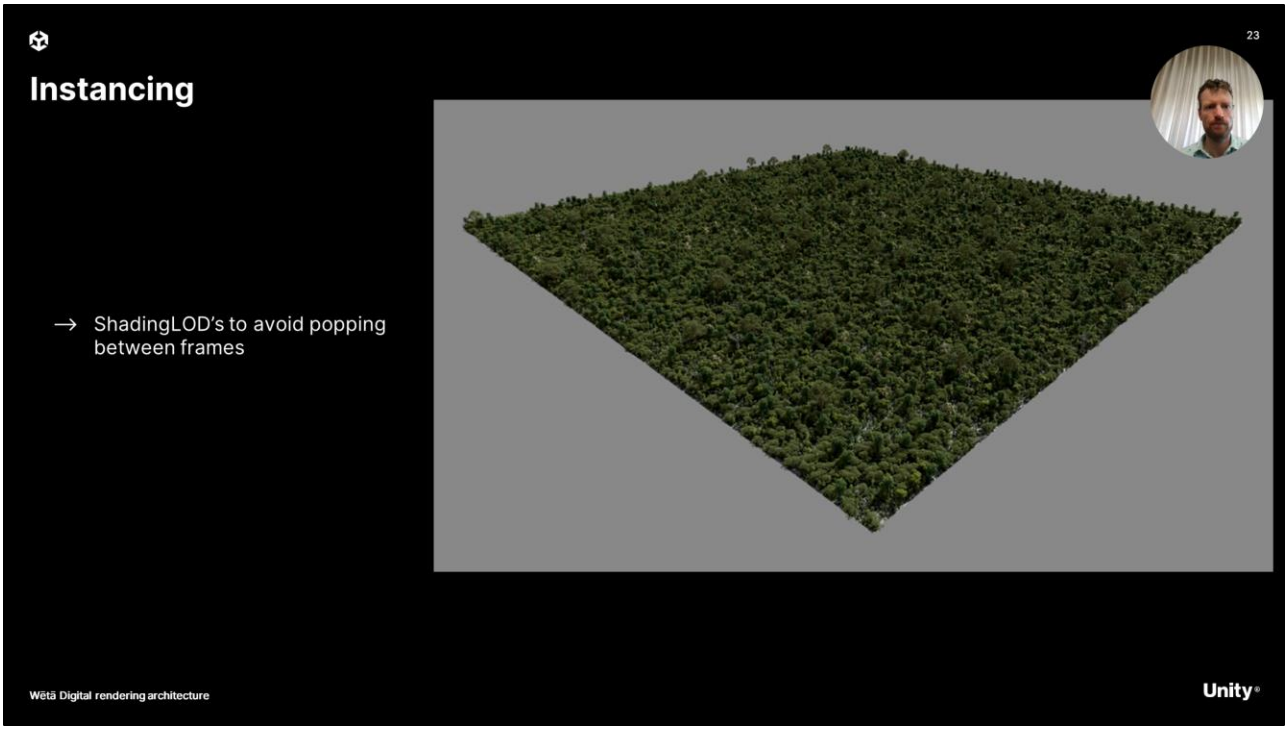
- The purpose of tessellation is to produce micropolygons. Typically for a production render, micropolygons are sub-pixel in screen-space, though adjusting the size of your micropolygons is obviously a common performance/quality trade-off. The micropolygons are fed into Manuka's shading phase, and so the density of those micropolygons controls the density at which shaders are executed and as a result textures are sampled etc.
- The output of Manuka's shading is stored on the micro polygons vertices, and again, these are then the full set of input parameters to evaluate the BxDf when later on we start shooting rays.



- So how far do we dice or tessellate? There is no easy answer to that and it depends on many settings. What is true is that, in most cases, we need a dicing camera (which could be the eye camera) before we can start tessellation. World distance and orientation is a key driver for how fine our sampling will be. Another key setting is the shading rate which is simply a artistic control to the renderer.
- Another option is to run Manuka in a special mode, producing oracle data. These files can be fed into a subsequent renderer with information on how much tessellation you are going to need.
- Exemplars, or instance “sources’ are obviously special case in this mode and come with a few limitations. At a high level, we are simply going to uniformly dice the exemplar at the density needed for the instance that needs the highest density, typically the one most in view of the dicing camera.

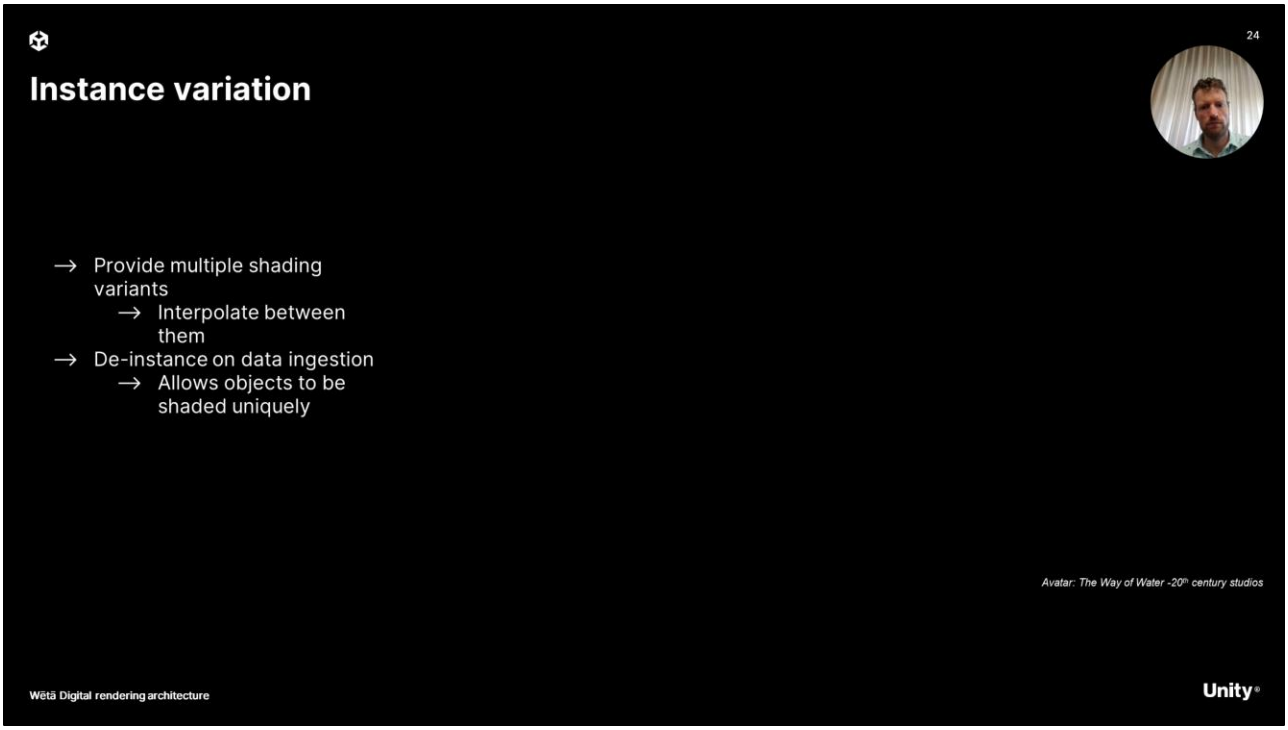
- Manuka has been designed to deal with very large amounts of instances. Multiple billions, and they can be nested, which means instances can have other instances which can have other instances and so on and so forth. This can reach quite a few levels.
- Also, procedurals can produce instances upon execution, but also, procedurals can be instanced themselves.
- Taking a quick look at this video, you can see an extreme amount of procedurally generated instances that are handled by Manuka.

- Instancing typically uses uniform dicing, with the dicing oracle providing the dicing rate required for the instance closest to the camera. However, there are scenarios where this breaks down.
- For example, if you have a single instance of a large object very close to the camera, dicing the exemplar for that instance uniformly might be too costly as it will produce a large amount of data. The instancing system needs to deal with this and it can using “large instance optimisation”.

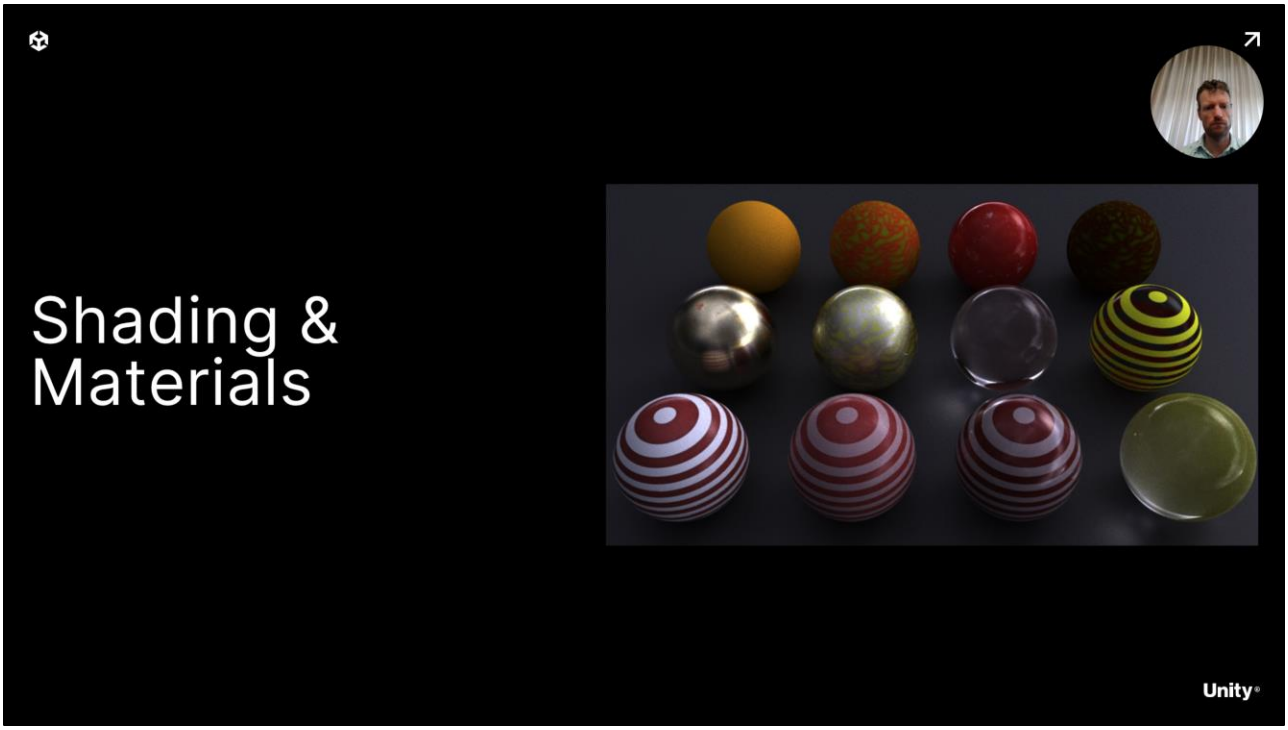


- Another important task of the instancing system is to make sure there is consistency between frames to avoid popping LOD's.
    - The instancing system together with the oracle can generate shading LOD's which results in more stable dicing rages for objects, resulting in temporal stability. This is demonstrated with this flyby in a forest of 1M trees.

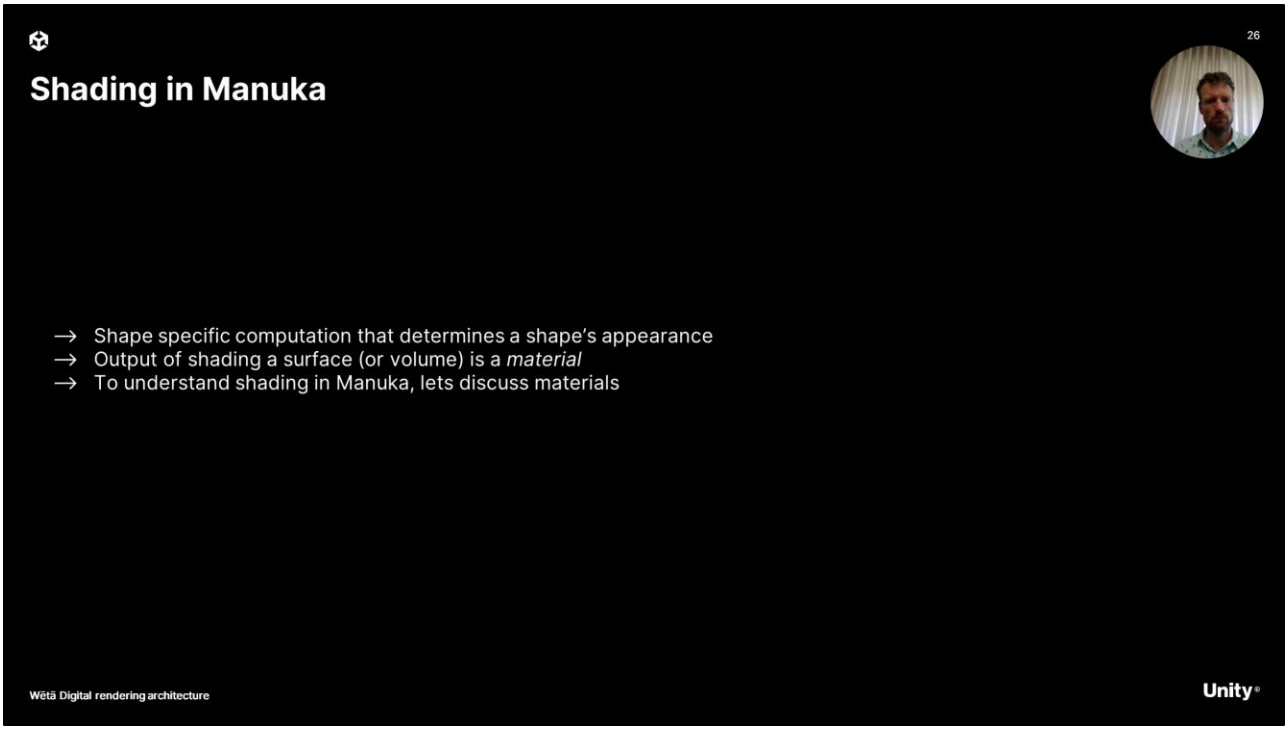
- But, instance counts are all cool, in reality in nature, things are seldom 100% identical. Artists will want to slightly alter the appearance of each instance by varying a texture or shading parameter. This is straightforward to implement in a shade-on-hit architecture where the full shader is executed each time, but for our shade before hit system this presents a problem.



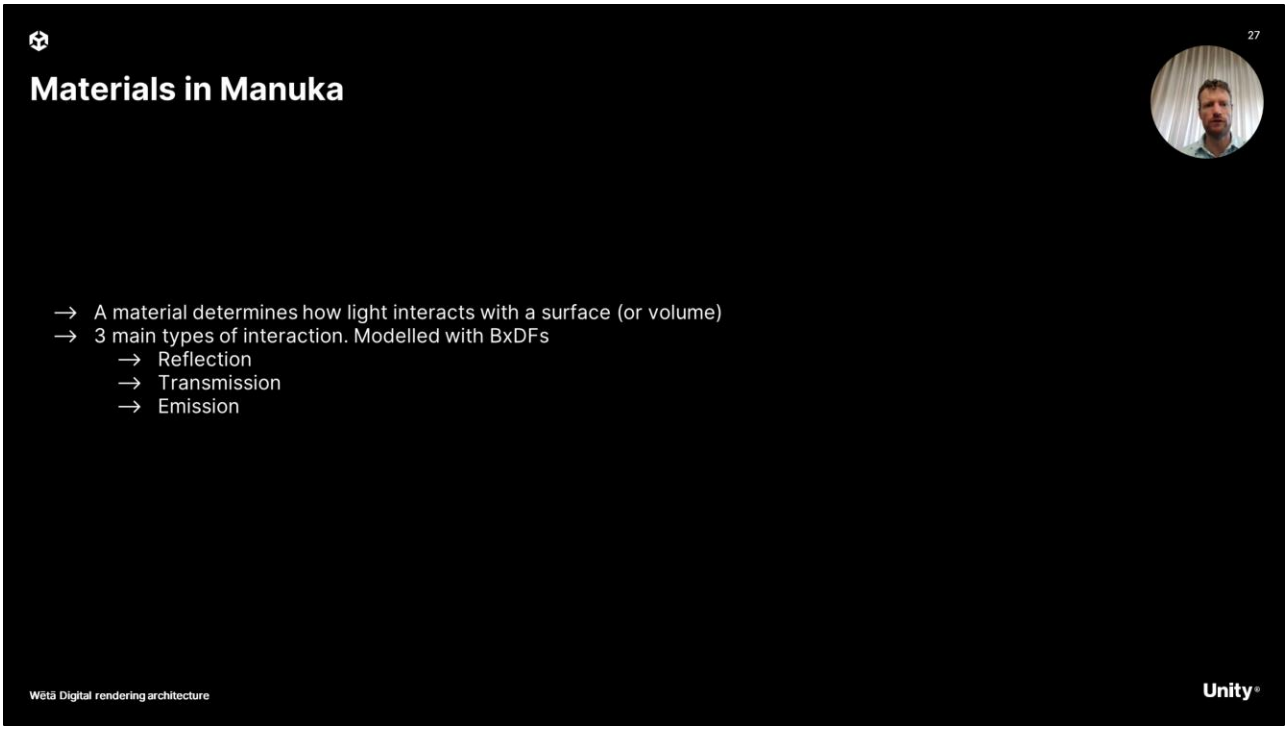
- To allow shading variation. Manuka has a few tricks up it's sleeve.
- First of all, you can provide multiple shading variations for an exemplar, all using the same geometry/topology. This works well when you want say multiple variations of leaf colors. Using those variants, Manuka can interpolate between them at render time. This however breaks down when there is a shading parameter that is driven by a world position, such as a snow or flood line.
- Should artist need this level of control on the instances, Manuka can also de-instance the data on ingestion.
    - This gives artists the artistic freedom of using instances in their content authoring pipeline and have it supported by the renderer without a flattening or baking step for the artist
    - Obviously, this does come at a cost, and in current internal developments we are looking at exploring a more elegant solution to this.



- (6 min)
- Regardless of how and why we tessellated, the next step is shading and no rendering talk would be complete without diving deeper into shading
- We already hinted that shaders deal with direction independent computation. Let’s explore this a little bit more in the following few slides.



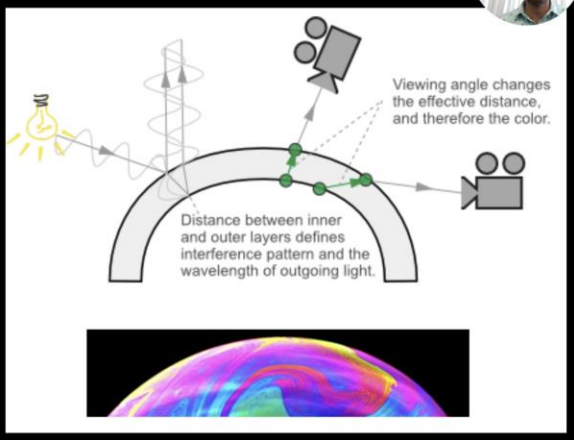
- So shading in Manuka is dealing with shape specific computation that determines a shape's appearance.
- The output of shading a surface for volume is what we call a material.
- And so to understand what shading does in Manuka, we need to understand the concept of materials, so let's look at this first.



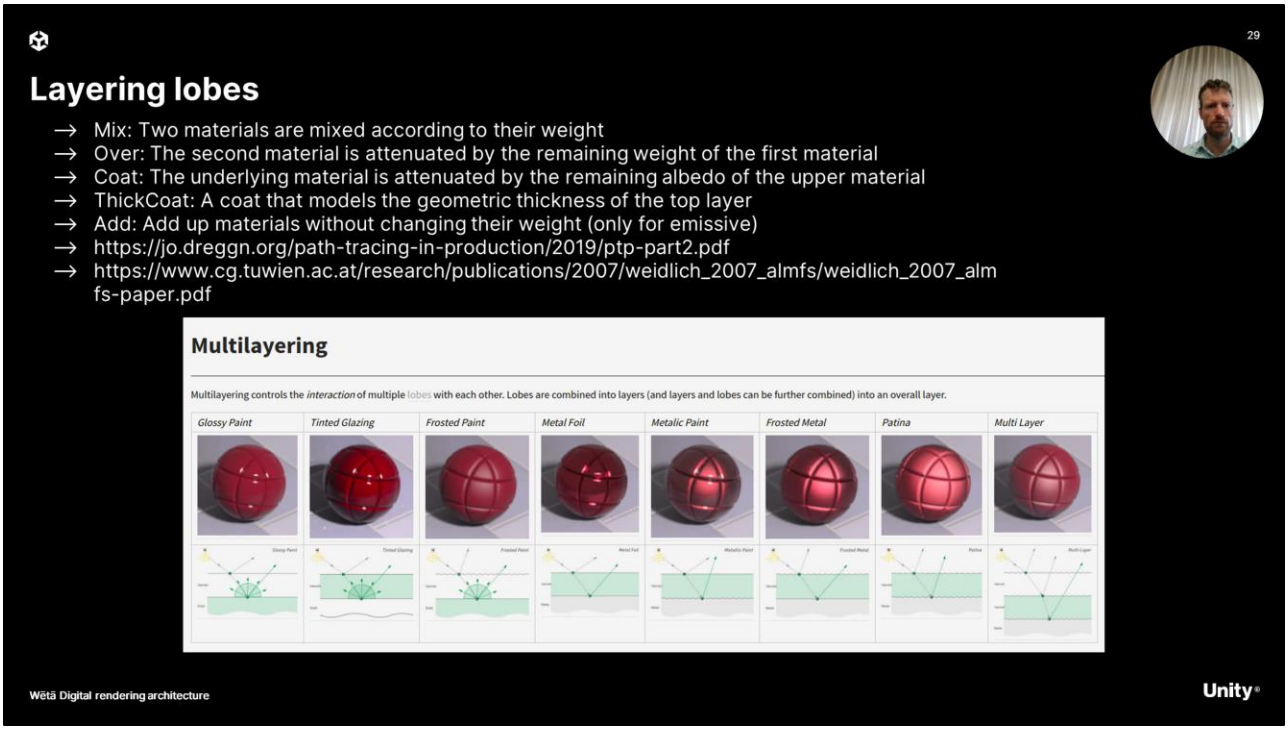
- A material determines how light interacts with a surface or a volume.
- There are 3 main types of interaction, modelled with bidirectional distribution functions, commonly referred to as BxDF’s. The main modes of interaction are reflection, transmission and emission.

Materials in Manuka (continued)

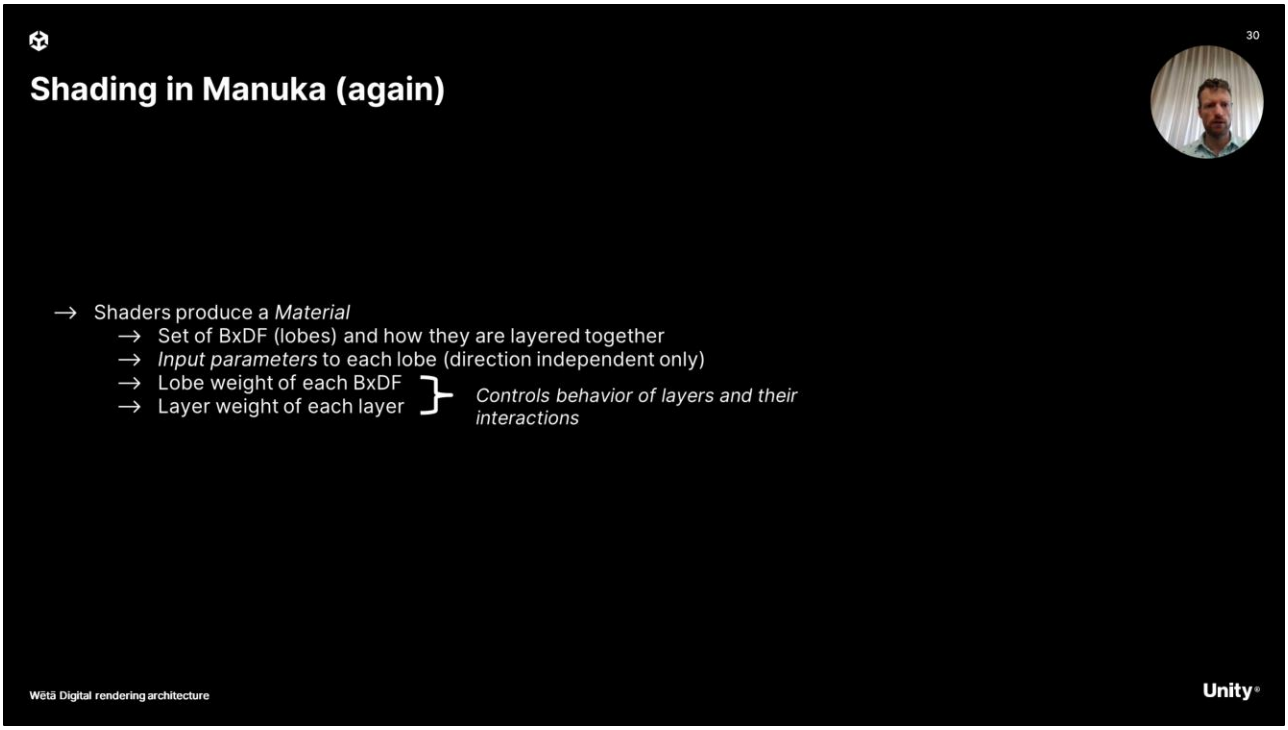
- → No single “uber model”
- → Combine N BxDFs in a user-defined way
  - → Manuka currently defines 187 BxDFs with groups for
    - → Metal
    - → Thin Glass
    - → Hair
    - → Glints
    - → Yarn and Woven cloth
    - → BSSRDFs (subsurface scattering)
    - → ...



- Manuka does not define a single “uber shader”, but rather ships with a set off 187 BxDFs that shader writers can combine and layer to create certain effects. These BxDF’s themselves are hardcoded.
- But with the set of BxDF’s available, we have been able to satisfy the needs of Weta’s production. The BxDF’s can be grouped and we have solutions available for metal, thin glass, hair supporting eccentricity and double cylinder models, glints, yarn and woven cloth, subsurface etc.

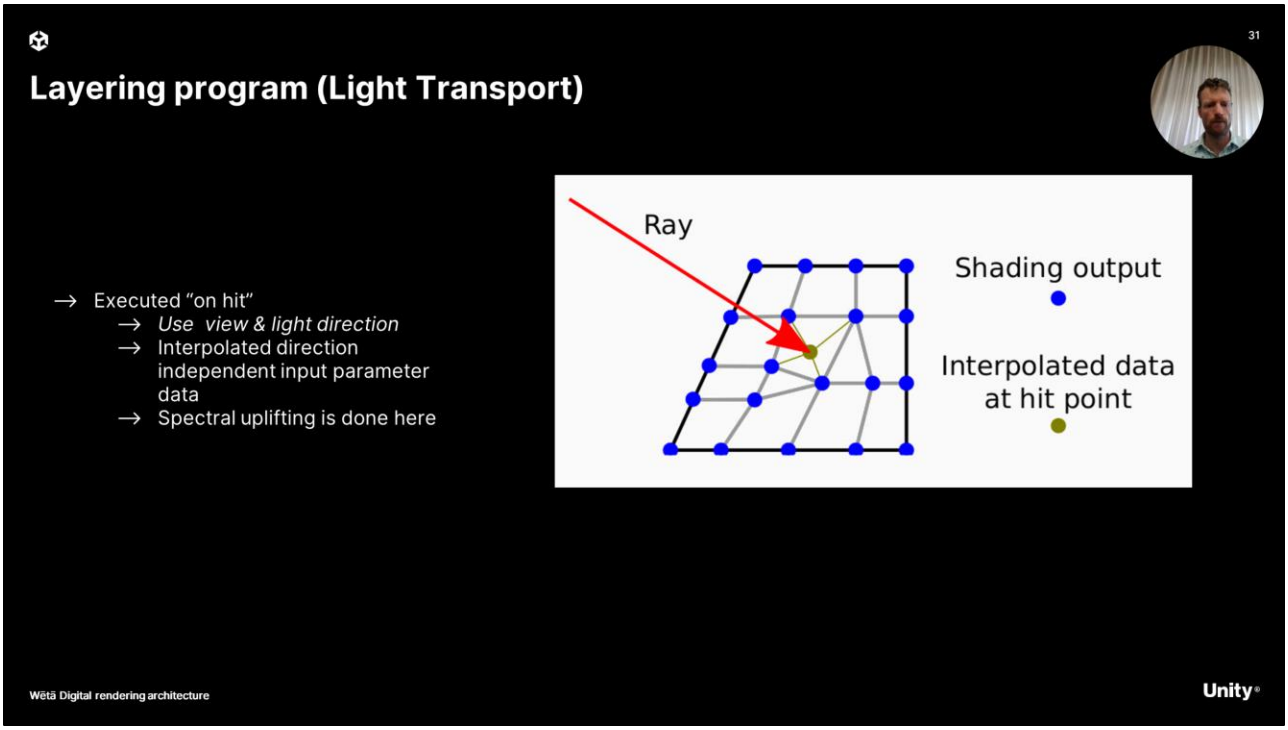


- In Manuka, we call BxDF's lobes, and the way they are combined defined the look of the object. While the BxDFs you can choose from is large but limited, the way you can combine them is user controllable. There are various ways to blend lobes together to produce a wide range of effects as can be seen on this image from the Manuka internal documentation
- A common practical layered material is car paint, but in reality there are many surfaces that combine many "layers" and we need the ability to model them to reach the fidelity we want in our final images.
- These layers are defined per shape and the more layers used, the more data we store on the micropolygon vertex in order to compute it during the path tracing phase.
- The ways in which we combine these is using the above blending modes and somewhat straightforward. Most modes are described in the paper linked.



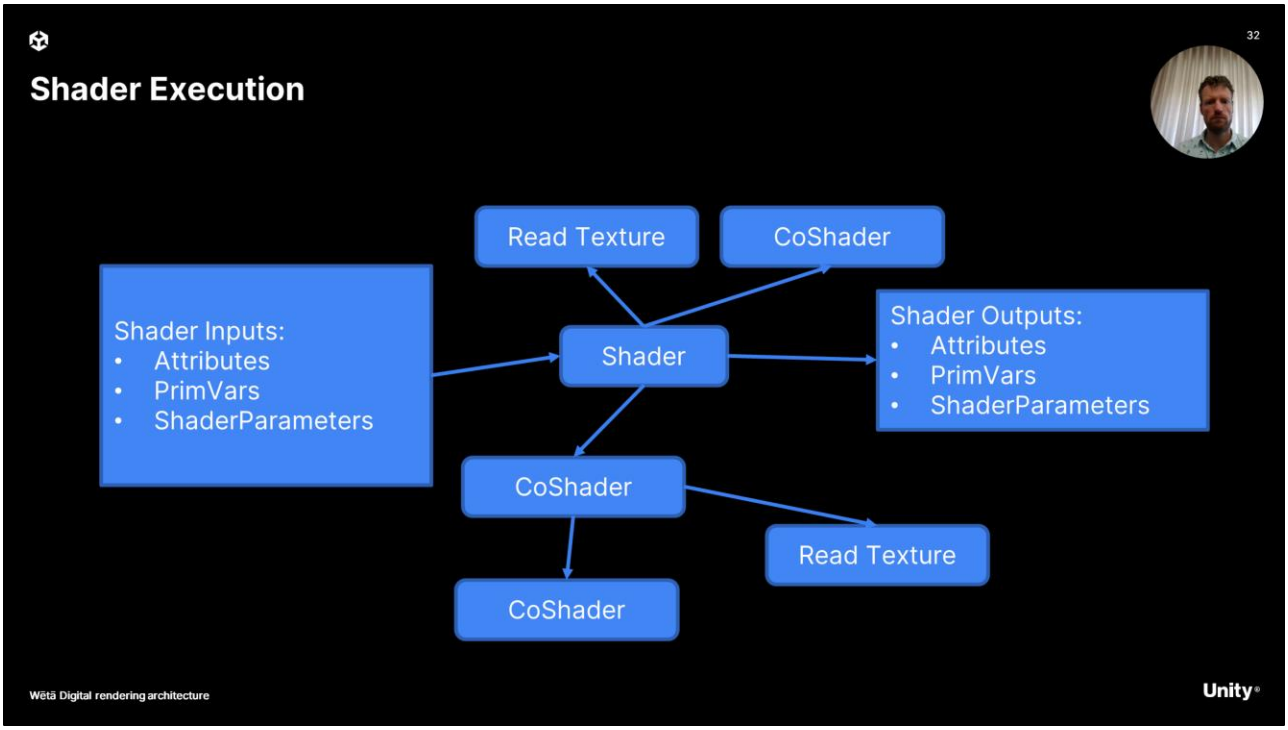
- Now that we know what a material is, we can switch back to what shading in Manuka does.. It creates the properties for a material!
- Breaking that down, we
  - setup a set of lobes or BxDFs and how they layer together.
  - We evaluate the direction independent parameters for each lobe, together with lobe and layer weight. Note that texture sampling is done here, so these parameters are frequently controlled by 1 or more textures.

- The output of the shading is stored on the micropolygon vertices, and again, more layers is more data.



- And with the shaded output we can now, during LightTransport or path tracing, evaluate what we call the layering program. This is what produces the final color or spectrum, taking view and light directions into account. The inputs to the layering program are interpolated from the neighboring vertices which have computed output data.

- The layering program can also be sampled in order to produce a direction for generating new ray bounces.
- Spectral uplifting is also part of this phase of the renderer.
- Spectral uplifting is a (very) under-constrained problem and there are many solutions with their own trade-offs, out of scope here.



- With the overview model of how we produce pictures now covered, let's circle back and look a little bit closer into our shaders.
- Shaders are written or code generated in the Renderman Shading language, or RSL in short. This dates back to the renderman specification I mentioned at the start.
    - The inputs to the shaders are things like attributes, primvars and shader parameters. These are what artists use to tweak
    - Shaders proceed to execute their logic, reading textures and executing user defined logic.
    - Shaders can also, indirectly, execute co-shaders, which are basically other shaders, which can again read more textures and execute further coshaders.
    - Ultimately, we output the material structure, parameters and optionally AOV's for the next stage in the pipeline. It is this material structure that defines the eventual look of the object.
- Now, different sections of a shader can run at different frequencies. Let's have a closer look.

Shader Execution stages

Pipeline Stage	When called	Notes
init	Once per primitive	
interior	Once per primitive	Only for primitives with a homogeneous interior
begin	Once per grid	
displacement	Once per grid	
prelighting	Once per grid	Only for primitives with a boundary
volumeboundarylocal	Once per grid	Only for primitives with a volumeboundarylocal interior
volume	Once per grid	Only for primitives with a heterogeneous interior

- A shader has certain “pipeline stages” and these stages run at different points in the shading execution and also at different frequencies.
- A table of different shading stages can be seen above. While diving into all the stages Is out of scope, let’s do a simple walkthrough of some key stages to understand how this works.

Shader Execution stages

→ Mnk_init() Responsible for determining the material structure

```
public void mnk_init output MnkObjectProperties objectProperties;
                     output MnkMaterial         material)
{
    // initialize default object properties.

    // ...

    MnkMaterialBoundary boundary = CreateBoundary(material);
    AddLobe(boundary, "reflection", LambertBrdf());
    AddLobe(boundary, "transmission", LambertBtdf());

    // ...
}
```


- The mnk_init call is the main entry point for shading. It is executed once per primitive and It is responsible for setting up the material structure.
- As you can see here, we are setting up a material that has 2 Lambertian lobes using lambertion reflection and transmission. While it is possible to have more layers, most production shaders have between 1 and 10 layers, the average varying per production.

Shader Execution stages

→ Mnk_prelighting() – Compute material parameters (per vertex)

```
public void mnk_prelighting(MnkMaterialBoundary boundary)
{
    LambertBrdf reflection   = LambertBrdf("diffuseReflectance",   color(1.0, 0, 0));
    LambertBtdf transmission = LambertBtdf("diffuseTransmittance", color(0, 1.0, 0));

    SetLobe(boundary, "reflection",   0.25, reflection);
    SetLobe(boundary, "transmission", 0.75, transmission);
}
```


- In the pre-lighting stage we compute the material parameters, per vertex.
- Notice that the input is the MaterialBoundary that we defined in the init stage. Here we set the material to be 25% reflection in red, and 75% transmission in green.

Shader Execution stages

→ Mnk_displacement() – Alters the position and normal of a vertex

```
public void mnk_displacement(output point P; output normal N)
{
    P += normalize(N) * height * mydisplacement();
    N = normalize(calculatenormal(P));
    point Pbump = P + N * bumpheight * mybump();
    N = normalize(calculatenormal(Pbump));
}
```

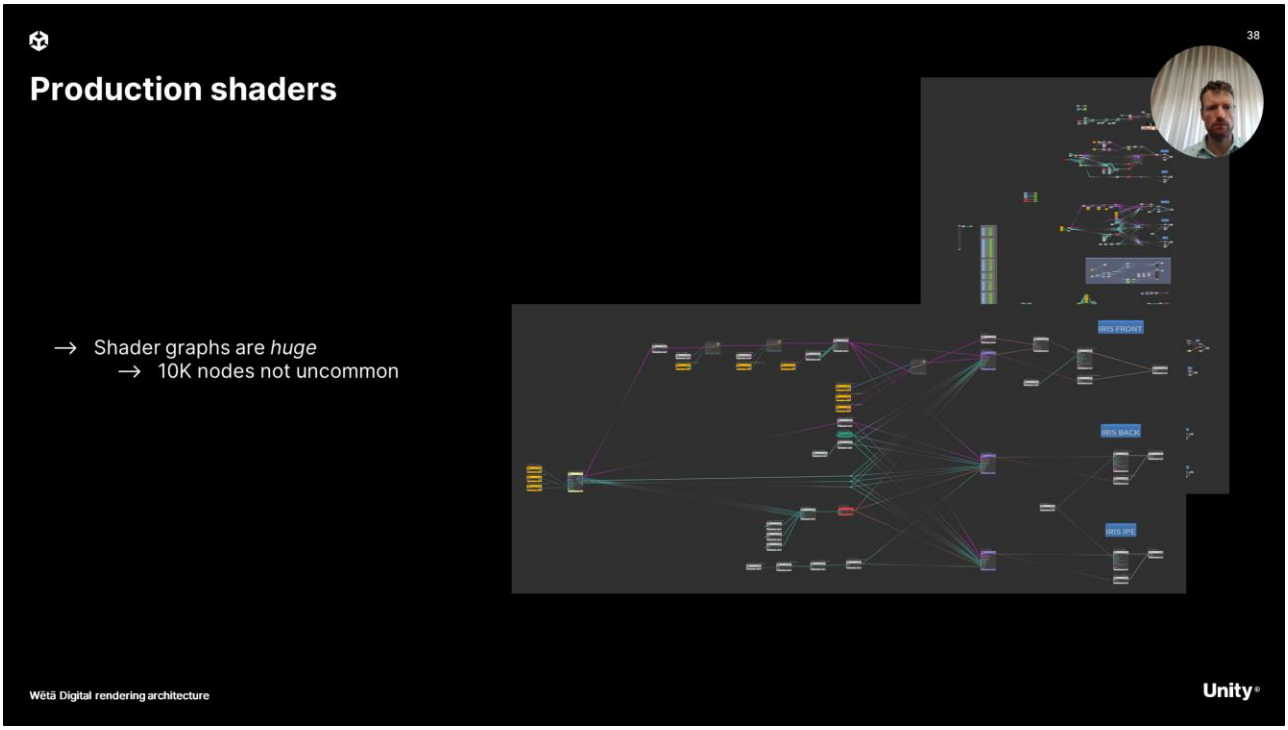

- Displacement is another shading stage, which is somewhat decoupled from the other stages. Displacement is computed per micropolygon vertex and the purpose of a displacement shader is to alter the position and normal of a vertex, usually driven by a displacement function that samples a texture. This is “mydisplacement” here.

Production shaders

- → Shaders written in the Renderman Shading Language (RSL)
- → Shaders are published as MSLO files
- → Hyperion compiles MSLO shader into an MSHD
  - → MSHD is effectively a shared library
  - → Unique per Manuka version
  - → Caching layer to avoid re-compilation

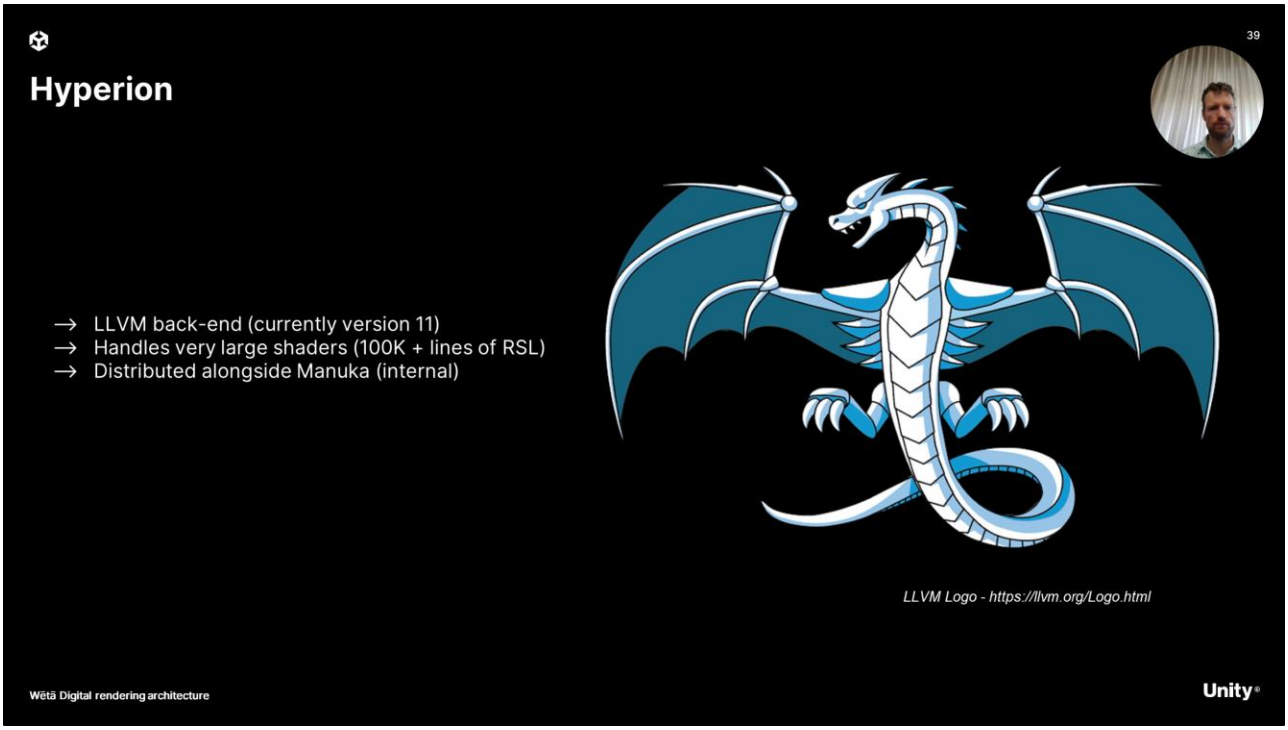
```
        CreateInterior(material,
            "homogeneous", ior);
    AddPhaseFunction(interior, "pf",
            HenyeyGreensteinPfRgb());
}


public void mnk_prelighting(output
        MnkMaterialBoundary boundary)
{
    MicrosurfaceDielectricBsdfRgb_v1
            params =
            MicrosurfaceDielectricBsdfRgb_v1(
            "roughness", color(roughness));
    SetLobe(boundary, "bsdf", 1.0, params);
}

public void mnk_interior(output
        MnkMaterialInterior interior)
{
    HenyeyGreensteinPfRgb pf =
            HenyeyGreensteinPfRgb("meanCosine",
            color(0));
    SetAbsorption(interior, color(0));
}
```

- As we have already seen, shader code is RSL. Now we don't actually publish the RSL together with the asset, there is a transformation step where we generate a MSLO file. You can compare this loosely with a C source file that has been expanded by the pre-processor. As such, it is able to compile with little dependencies. The MSLO is a compressed format.
- The MSLO files are what is read by Hyperion, which is our shader compiler. Hyperion's job is to transform the MSLO into a MSHD, which is actually executable at runtime. Like any compilation, this compilation can be slow and because the MSHD is effectively a shared library, we need to make sure the ABI is stable. To avoid many machines compiling the same shader, we store compiled shaders in a network cache, which is flavoured by the Manuka version used.



- Over the years, Weta has accumulated a large set of shaders. While shaders can be hand-written as RSL, they can also be code-generated from a graph. We have proprietary tools for shader authoring which will be topic for another day, but what I wanted to illustrate here is that shader graphs are very large. Having 10.000 nodes in a shader graph is nothing exceptional. These large graphs, and thereby large amounts of code do put pressure on the shader compilation in Hyperion.

.



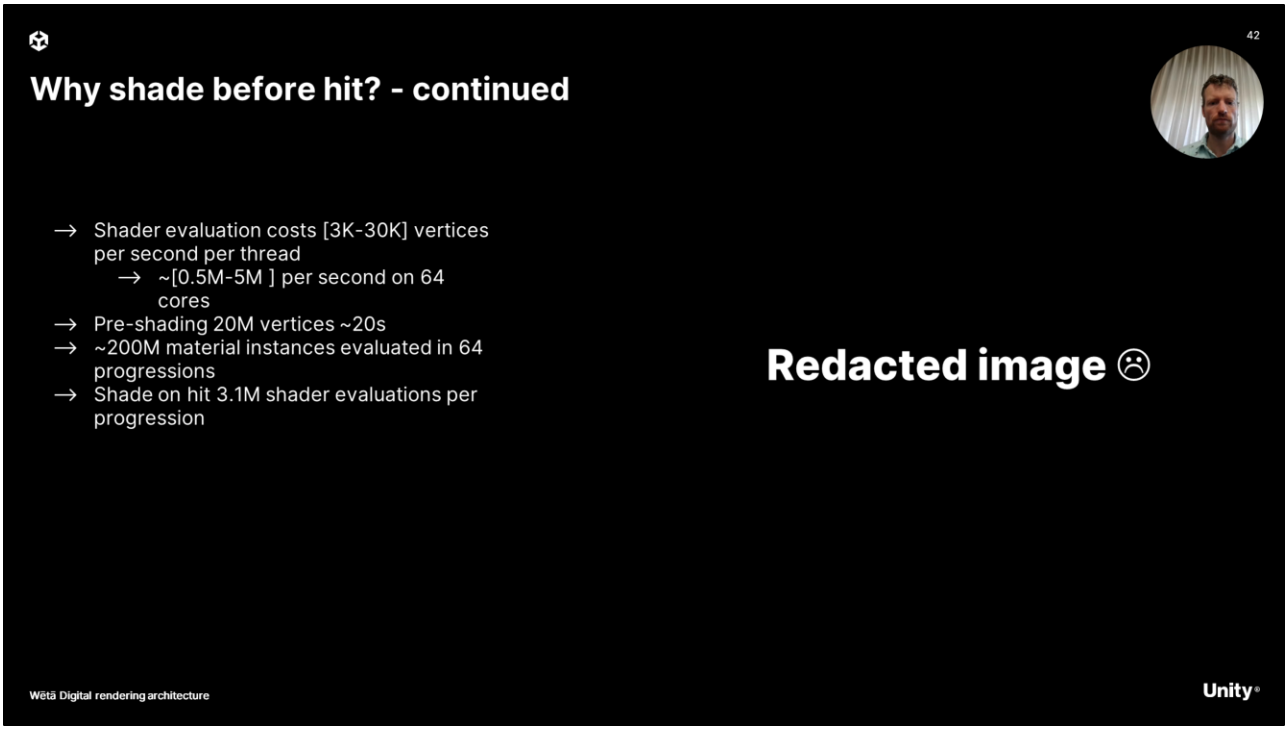
- So a final word then on Hyperion, which is our shader compiler distributed internally with Manuka. It is backed by LLVM version 11 and it scales to very large shaders. 100K+ lines of RSL are not uncommon. And while OSL is not directly supported by Manuka, our common shader size is a concern for moving to other languages. For various reasons we are looking to move to a newer shader back-end but it is too early to go into depth here. We hope to be able to discuss new shading tech in an upcoming talk.

(5 mins)
Now all of this will likely still have you wondering. Why do we do shade before hit? Hopefully you understand *how* we do shade before hit, but I have been pretty vague on *why* we do shade on hit. I mentioned that we do this for performance reasons but have not explained how this works. What I have explained is the difficulty with instancing. Still, lets look into the wins more in this section.

- As mentioned in the previous section, we have very large shaders and that makes executing those shaders expensive. We can get performance if we can reduce the amount of times we execute the shaders.
- By storing shader execution on the micropolygon vertex in a directional independent manner we create the ability to reduce the amount of shader execution in 3 ways:
    - Sampling the same polygon multiple times only executes the layer program multiple times. The shader data is simply interpolated from the neighboring vertices.
    - Because the data is direction-less, the data is re-useable over “similar” renders. Think about an artist placing a light in the scene. To move a light, no re-shading is needed if all the data is correctly cached.
    - Even consecutive frames that are “not too different” can re-use shaded data. Think static objects at medium/far distance from the camera.
- Beyond these advantages, we reduce the load on the path tracing phase. No texture sampling is done during path tracing, avoiding the need to sample textures and reducing the memory load on this phase of rendering as we don’t need to open up all these large files.
- And finally there is CPU caching advantage. Because shading is executed per object, or rather per grid, it is highly likely that all vertices in the grid will access the same textures. This helps limit the IO overhead of our texture fetches as the textures will likely live on a network share somewhere. Remember that render compute nodes in a renderwall will have little to no

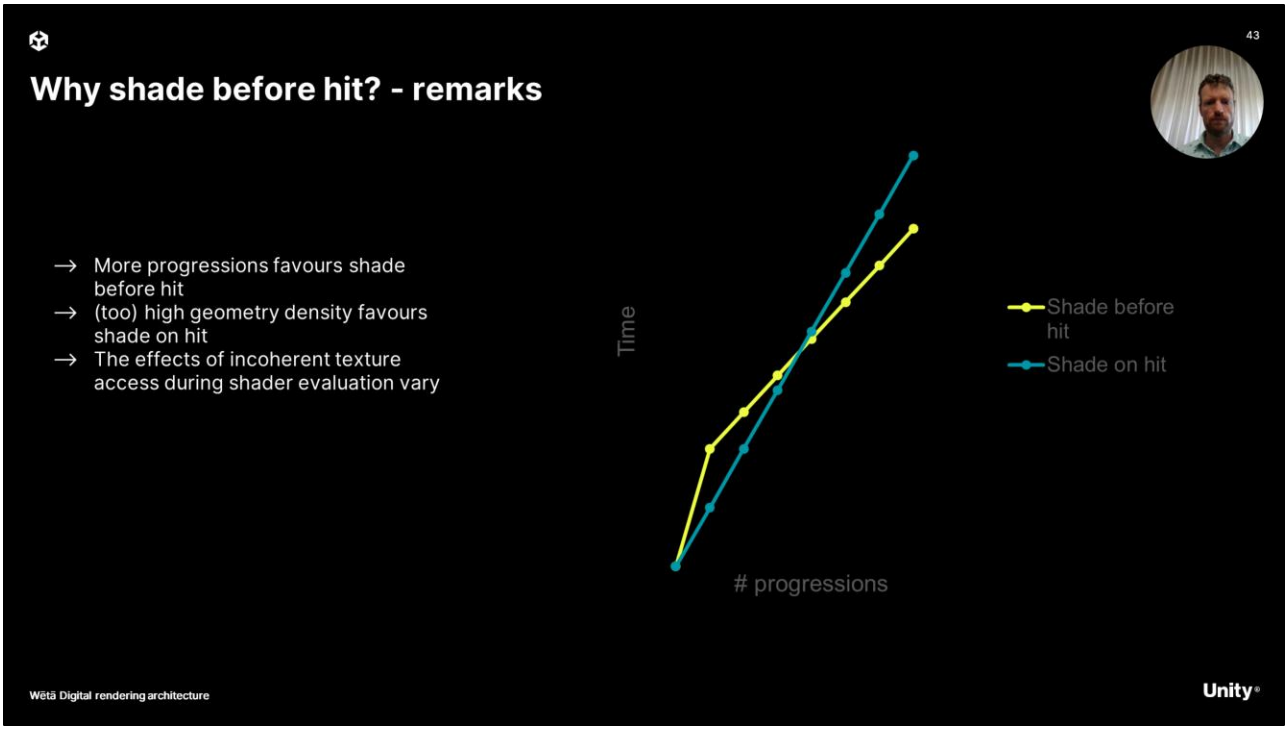
local storage.

And finally, shading can be easily distributed and the result of the pre-shading can be used for resuming renders, avoiding paying the cost for the data ingestion and tessellation a second time when the circumstances allow it.



- To clarify this further, lets do a thought experiment, let's say we can evaluate our shaders in a range of 3 to 30 thousand times per second per thread, yielding a throughput in the order of half a million to 5 million vertices per second on a 64 core machine. These numbers have nothing to do with the image here and are largely illustrative to explain the concepts.

- A scene might have 20M pre-shaded vertices, which would take roughly 20s to compute in this model. A shade-on-hit renderer would obviously not pay for this cost at this stage.
- That scene might produce 200M "material instance", which are basically points at which the layer program is executed and we have evaluated the BxDF lobes. This is done over 64 progressions, yielding that we would need roughly 3.1M hits per progression in a shade on hit model.
- Now it should be clear that evaluating the layer program is substantially cheaper compared to evaluating the full shader. The more times you need to evaluate your shader, the more it makes sense to have these evaluations pre-computed on the vertex.



- Now these numbers vary wildly and the parameters of your render can change to make things slide into favour for one or the other architecture. Typically you could say that the more progressions you do, the more shade before hit becomes appealing. This is an important reason why we state that Manuka is a :time to last pixel: optimized renderer.
- However, if your input geometry is very or even too dense, you would end up pre-shading too much data, swinging the pendulum the other way
- The argument of coherent texture access, while present, is very hard to quantify. There are a lot of variables at play for texture access on the network layer. There is the CPU hardware and it’s caching tiers, the storage back-end and shaders and textures used, combined with things like network traffic and actual filer usage. Testing this fairly is next to impossible as there are always renders going on.

Why shade before hit?

- → Texture reduction
- → BxDF input data per tessellated vertex
  - → 4-200 bytes per vertex
  - → Compressilble
- → Layering program (per shape)

Textures referenced	2.1TB	950GB	5.5TB
Textures read from disk	105.4GB	3.6GB	77.6GB
Number of files	27K	26K	22K
Avg. Number of layers	5.38	1.94	1.58
Avg. per vertex BSDF inputs	35.91B	11.95B	23.25B
Total per vertex data	12.85GB	1.25GB	24.4GB
% stored	12.2%	34.7%	31.4%

- To continue on the topic of textures, this is a table from the 2018 paper I referenced earlier. The numbers no longer reflect current production data, but the point it illustrates around reduced texture access is still valid today.
- So the 3 columns are data from 3 different productions and the top row shows the amount of texture data on disk was used for a typical frame. The second column then shows how much actual data is being read. This reflects the mipmap level and coverage for instance.
- The 3rd row is the number of texture file we have. For modern productions this is way more these days. The number of mipmapped EXR textures we read per frame can be 50-70K these days.
- Next up is the number of layers that we feed to our layer program. You see here that while we support a very large number of layers, the actual number of layers, on average is not to too great. While the amount of data we store per vertex can be between 4-200 bytes, in reality it is often much more towards the lower end. The data is also very compressible.
- In short the takeaway from this table is that pre-shading allows us to store less data during ray traversal, which in reality means we can render bigger scenes.

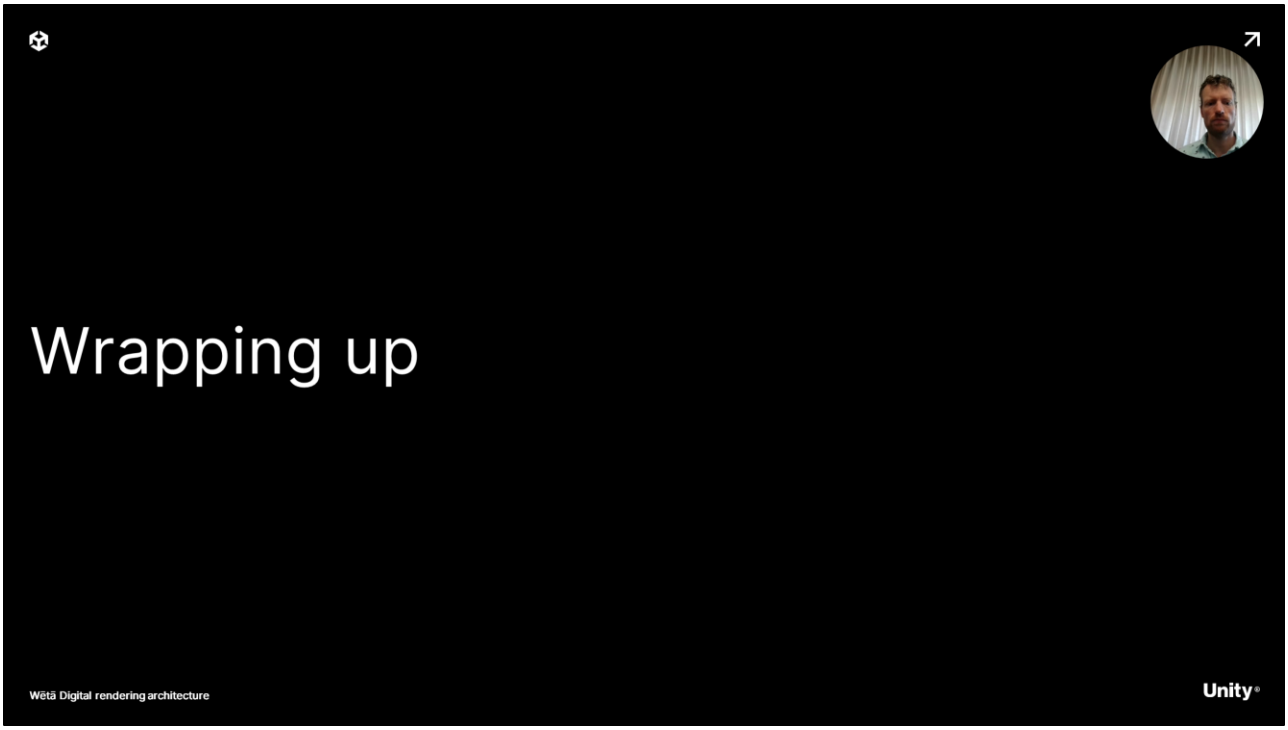
Shade before hit limitations

→ Longer time to first pixel (slow iteration times)
→ Interaction with instancing sub-optimal
  → Workflow difficulties / more memory

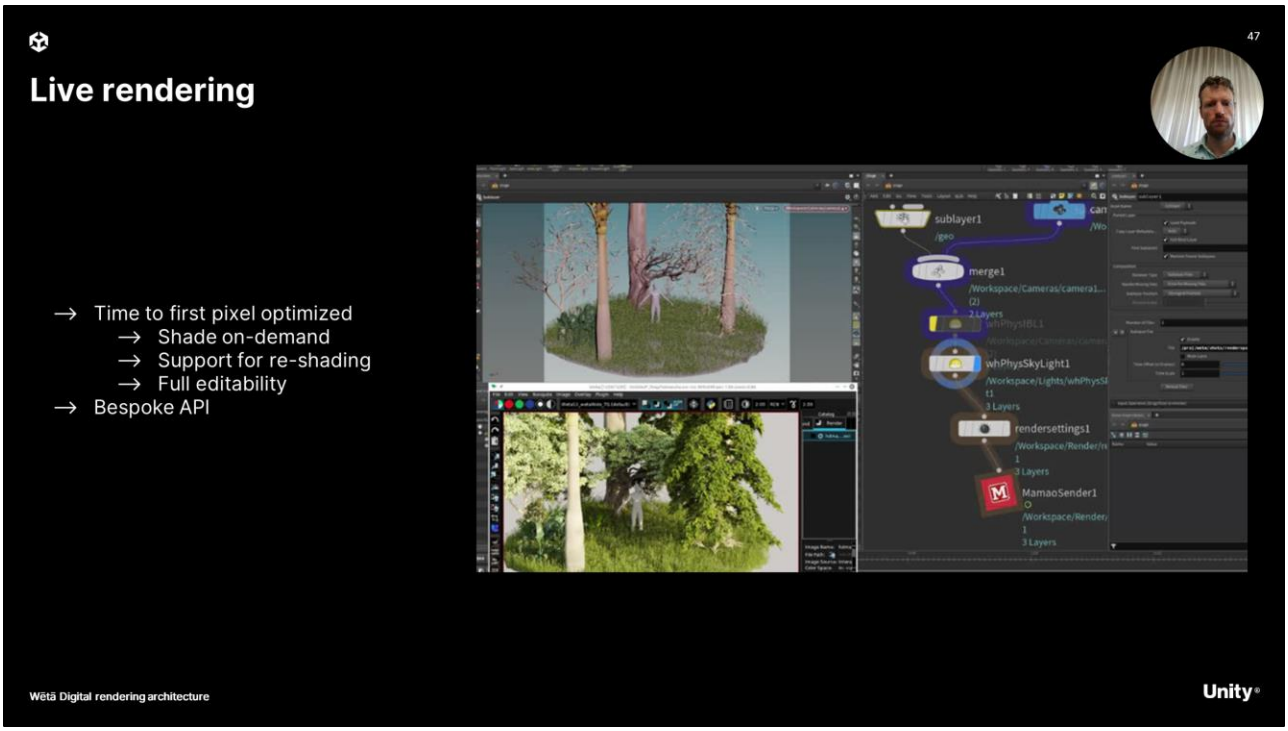
Avatar: The Way of Water -20th century studios

Now it should be clear that shade before hit still is a trade-off. The most obvious trade-off is our longer time to first pixel. This can make Manuka appear “slow” as a shade on hit renderer might already show some (noisy) pixels, Manuka will still be pre-shading. The *time to first pixe*l as it is called is not first in class.

The other drawback is the interaction with instancing. We have mentioned this before and while we can de-instance on scene ingestion, this does consume more memory. This won’t impede artist workflow where the memory is not a concern, but in reality a lot of shots will have memory constraints, and so simply de-instancing everything on ingestion is not always feasible.



(4 mins)
And with this, we have covered the majority of Manuka;s architecture, but we have also glanced over some very important features and aspects. Let’s use this last section of the presentation to cover a few use-cases in Manuka that might be unorthodox.

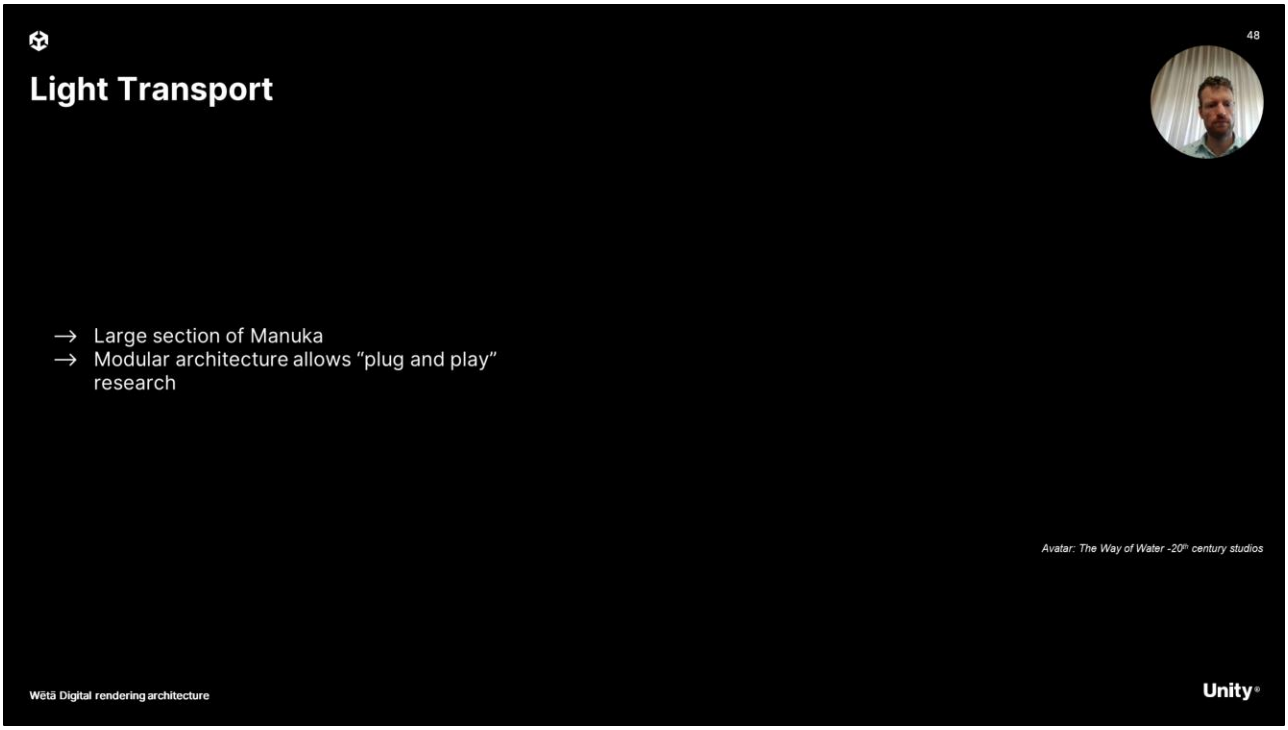


Now you may think that this architecture is not particularly well suited for live or interactive rendering and you'd be right.

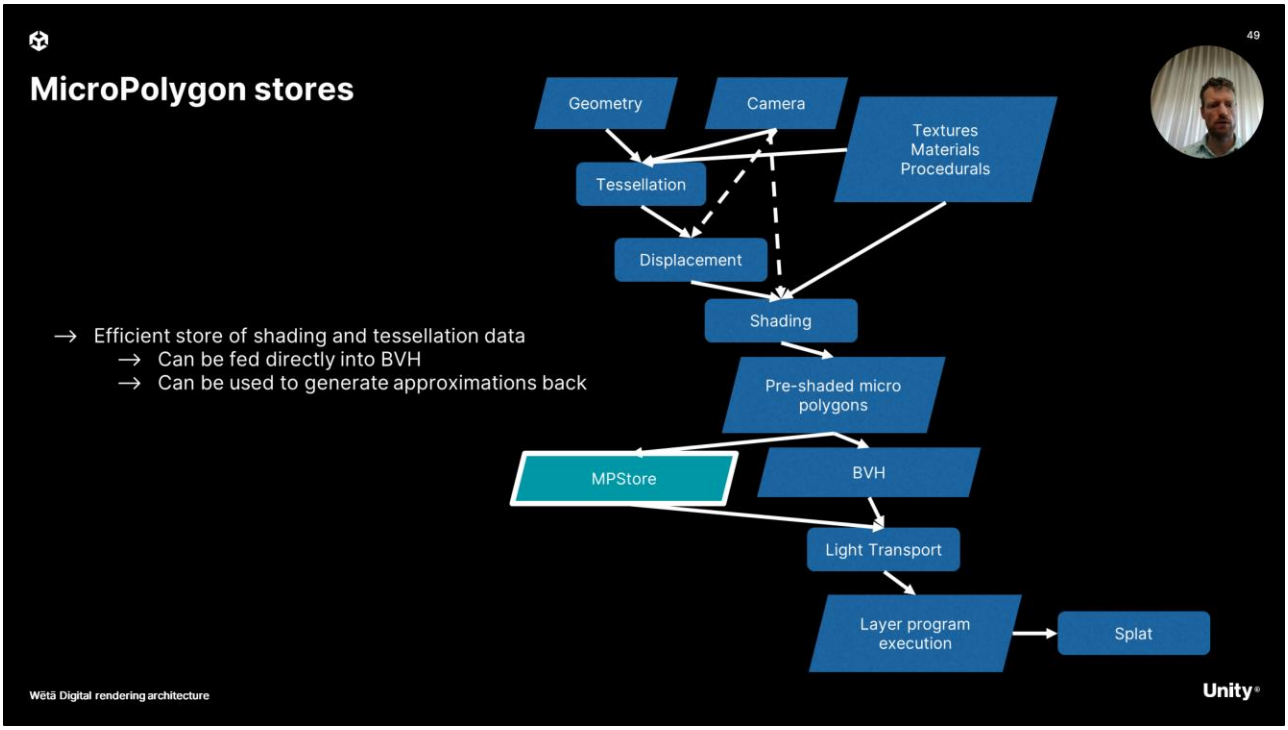
However, Manuka does have a live rendering *mode,* which prioritizes time to first pixel instead of final images. This mode does a few things:

- It switches the shading back-end to be on-demand. This can be best viewed as an intermediate between shade on hit and shade before hit. Rather than shading all geometry up front, we shade the grid of the object only when we first hit it and then store it in a cache. This means our time to first pixel is reduced. We tessellate and shade a shape only when it is hit by a ray. This does produce a problem in that we can only trace rays when the geometry is in the BVH, which means we can't do displacement in this mode.
- Live rendering mode in Manuka also supports re-shading, which allows you to cache the tessellation step so that you don't have to execute it again when you change something "simple" in a shdaer like a colour.
- And finally, this is a fully editable mode of Manuka, where changes can be made to geometry/lights/instances "on the fly". In order to support editing better, the BVH used in this mode is layered, meaning that we don't have to rebuild the full BVH for localized edits.
- The interactive rendering section is a section of Manuka where we can see Manuka deviating from the Renderman API norm. The Renderman API was simply not designed for this flexibility and the APIs we use for editing can be considered more modern. Perhaps in the future our offline rendering will also

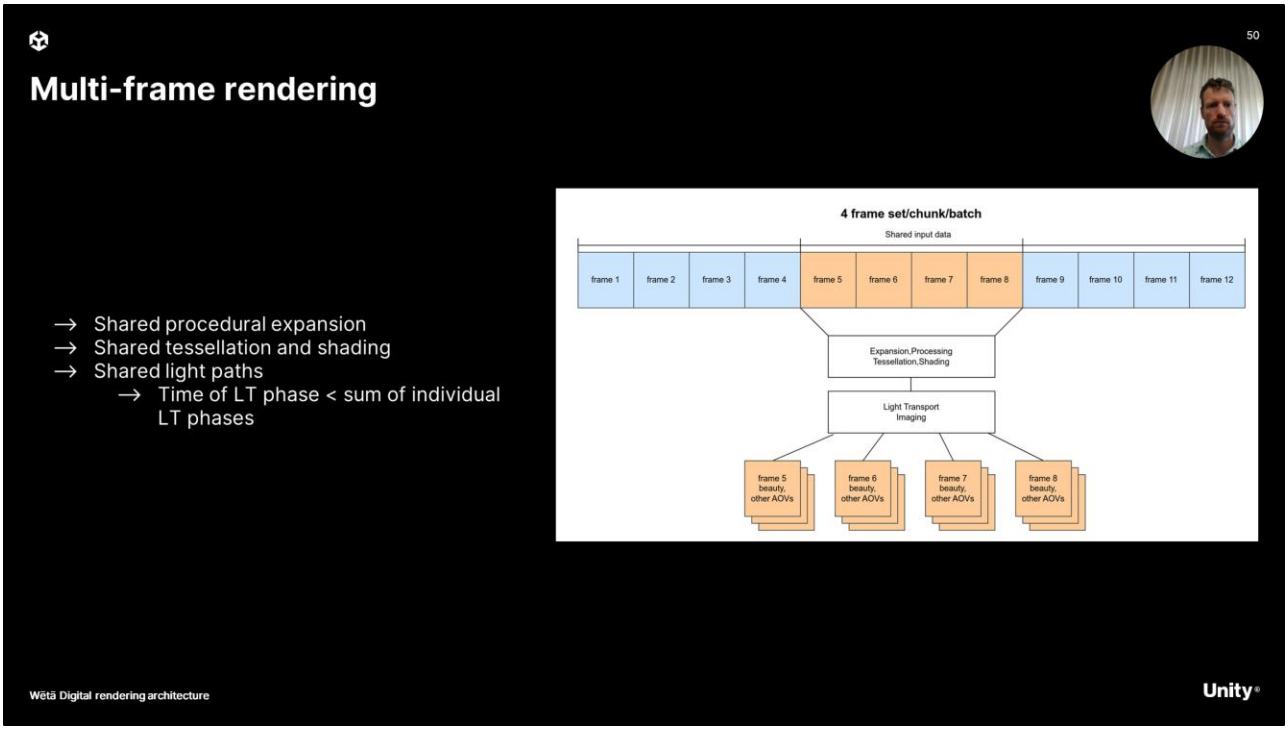
move to a more modern API.



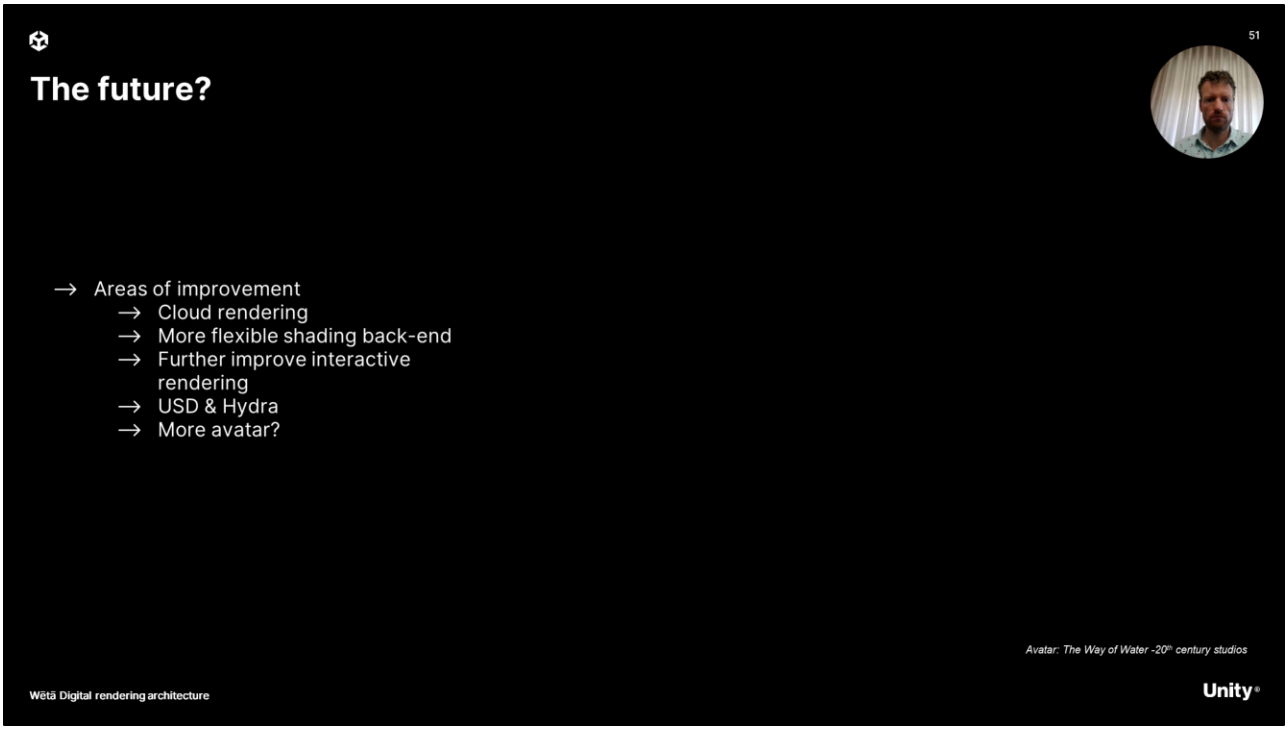
- Now even though we only briefly cover lighttransport and it is conceptually relatively straightforward, it is actually a fairly large section of the Manuka code base.
- Doing Light transport in a performant way is a very active area of research. In a scene like this, where there are many highlights, reflections and refractions, brute force will simply fail to produce a sharp image in a reasonable timeframe.
- In the LightTransport section of Manuka, it more closely resembles a “research renderer” as opposed to a “production renderer”. Many techniques are explored and controllable for the knowledgeable artist to work well in very specific scenarios. This suits us well, different shows will require different techniques as they have different problems. Avatar 2 had a lot of water and caustics for instance, whereas another show such as Alita Battle Angel can really push what we do for rendering eyes.



- The caching of shaded data has been mentioned before, but the usage of it is still expanding and so it is worth mentioning it separately here.
- What we see here is a more detailed pipeline of Manuka and what we have discussed. The shading cache , we internally call this a Micropolygon store, and this data can be used to feed directly into LightTrasport. It should be clear that if we have “hot” MPStore caches, our time to lighttransport, and thereby our time to first pixel, can be greatly improved as we can bypass a large section of work.
- We can also use the mpstore data to generate good approximations for earlier in our pipeline. The geometry includes the output of displacement and can be baked with an albedo color per vert, giving a simple object that can be used as a reference elsewhere.



- Now MPStores can for also be used implement multi-frame rendering with (nearly) static content. We basically store (slightly) more shaded vertices on the geometry to satisfy the camera/object movement for the frames covered.
- Similar to real-time rendering, multi-frame rendering allows us to re-use some of the data form the previous frame. In this case, we can spread the cost of procedural expansion, tessellation and shading for these objects.
- And so while a multi-frame render will take more time compared to a single frame, the extra time is compensated by the extra frames that you save.



- And finally, to conclude, Manuka is still a very active area of development at Unity/Weta Digital. To highlight just a few areas of improvement we are looking into:
    - The experience of cloud rendering for Avatar 2 has given us an excellent view on where pain points are for us and what we can best do to avoid them.
    - Other areas of development is a more flexible shading back-end, so that we can combine the best of both worlds for shade before and shade on hit.
    - Interactive rendering is another key area of improvement. This works together with integrating ever more tightly with USD workflows and USD enabled DCC's.
    - And maybe, we will also render more blue people in the future too.