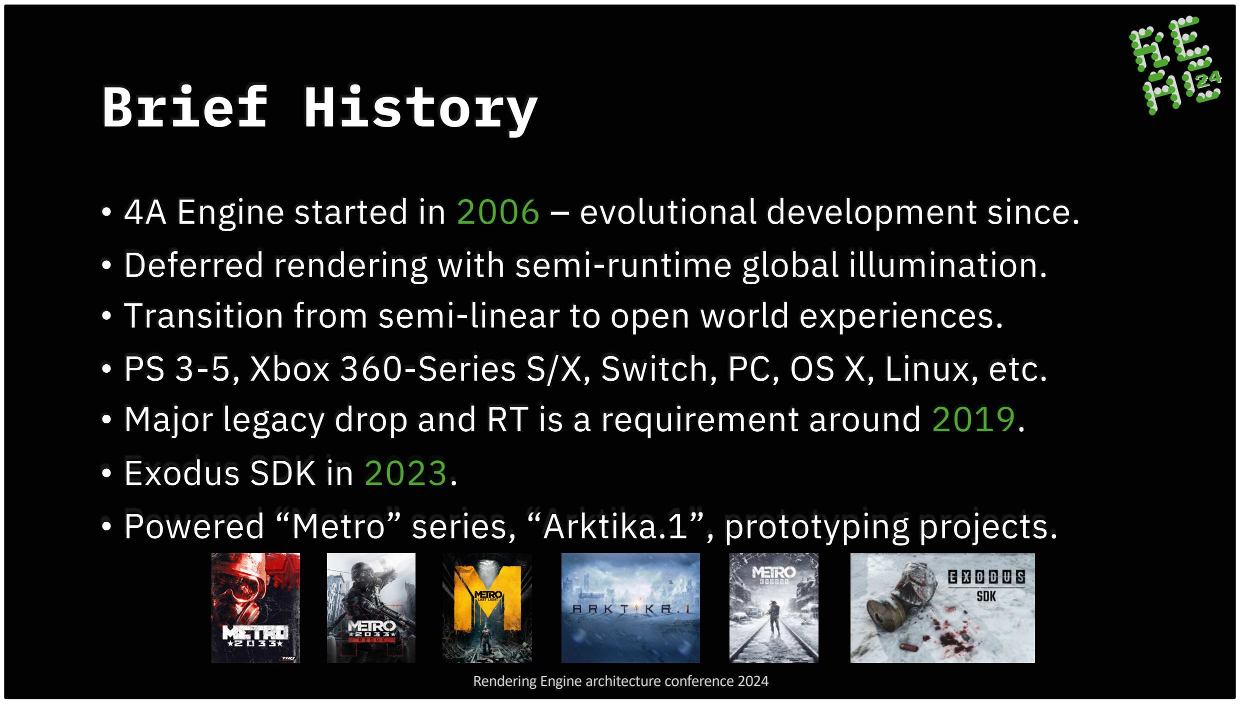
Resources Management Architecture in 4A Engine

RENDERING ENGINE ARCHITECTURE CONFERENCE REAC 24

Hi everyone. My name is Oleksandr Drazhevskyi. I'm a graphics programmer with more than 10 years of experience. I have helped to design and develop several rendering technologies while working as a Technical Architect at Ubisoft.
More recently I was working on improving 4A Engine. And, for the next hour or so, we'll be going through *some* of these improvements focusing on low-level rendering resources representation and management.

Before we get started and to set expectations – this talk is more of a technical report rather than an introduction of something completely new and unheard of. But at the same time, there's a fair amount of novelty and uncommon solutions which, I hope, you'll find interesting and useful.



Brief History

- 4A Engine started in 2006 – evolutional development since.
- Deferred rendering with semi-runtime global illumination.
- Transition from semi-linear to open world experiences.
- PS 3-5, Xbox 360-Series S/X, Switch, PC, OS X, Linux, etc.
- Major legacy drop and RT is a requirement around 2019.
- Exodus SDK in 2023.
- Powered “Metro” series, “Arktika.1”, prototyping projects.

Rendering Engine architecture conference 2024

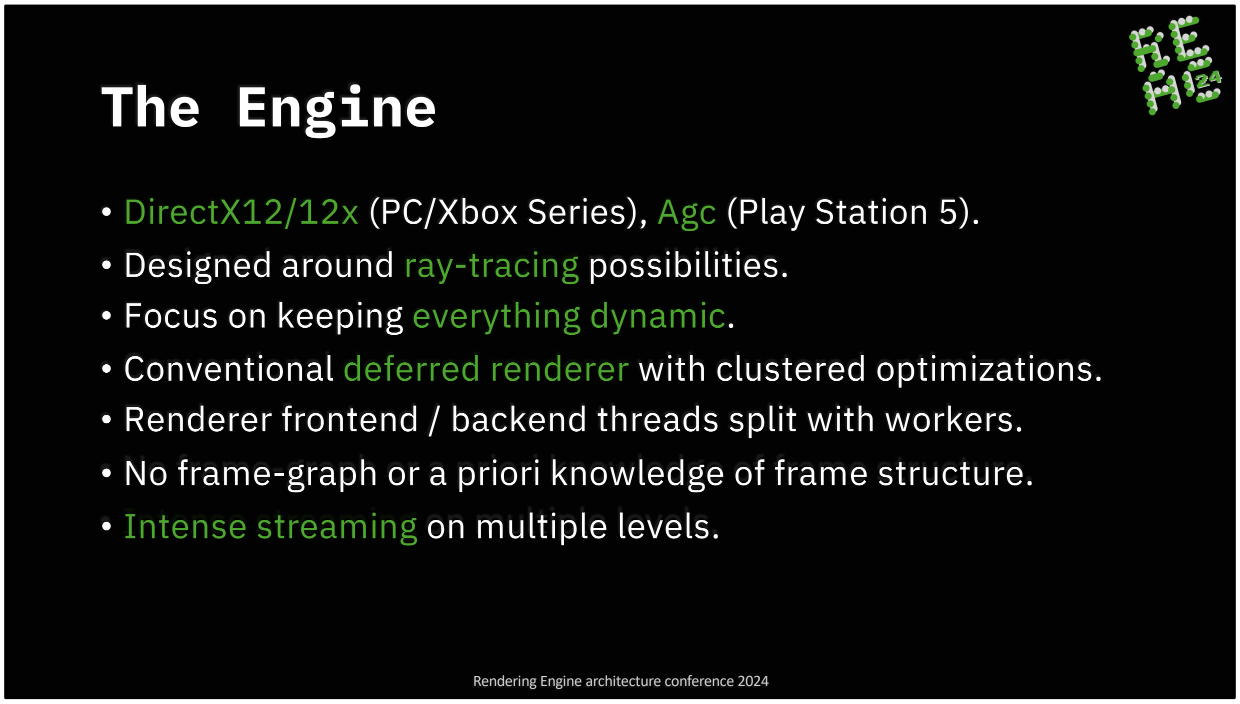
First things first – a short history of the Engine to set the context.

The 4A Engine started around 2006 with an aim to power Metro 2033 (a story-driven first-person survival shooter).
It was initially designed around deferred shading with completely dynamic lighting – including RSM-based global illumination.
In its’ initial shape (although, with significant modifications) it helped to ship multiple projects and delivered them on wide range of platforms. From PC and consoles to VR headsets.

With Metro Exodus the ambition dictated a shift towards more open-world level and game design.
That resulted in streaming, scalability, and adaptability to become a major design decision factors for core Engine systems.

Furthermore - while working on Enhanced Edition of Metro Exodus it has been agreed to drop support of legacy platforms and non-ray-trace-capable hardware.
Enhanced Edition of Metro Exodus was released in 2019 with RT being a requirement.

And, finally, in 2023 a version of the Engine used in Metro Exodus was released for public usage.
It was a long “passion” projects of studio founders and worked out really well with enormous number of Mods being developed in it.



The Engine

- DirectX12/12x (PC/Xbox Series), Agc (Play Station 5).
- Designed around ray-tracing possibilities.
- Focus on keeping everything dynamic.
- Conventional deferred renderer with clustered optimizations.
- Renderer frontend / backend threads split with workers.
- No frame-graph or a priori knowledge of frame structure.
- Intense streaming on multiple levels.

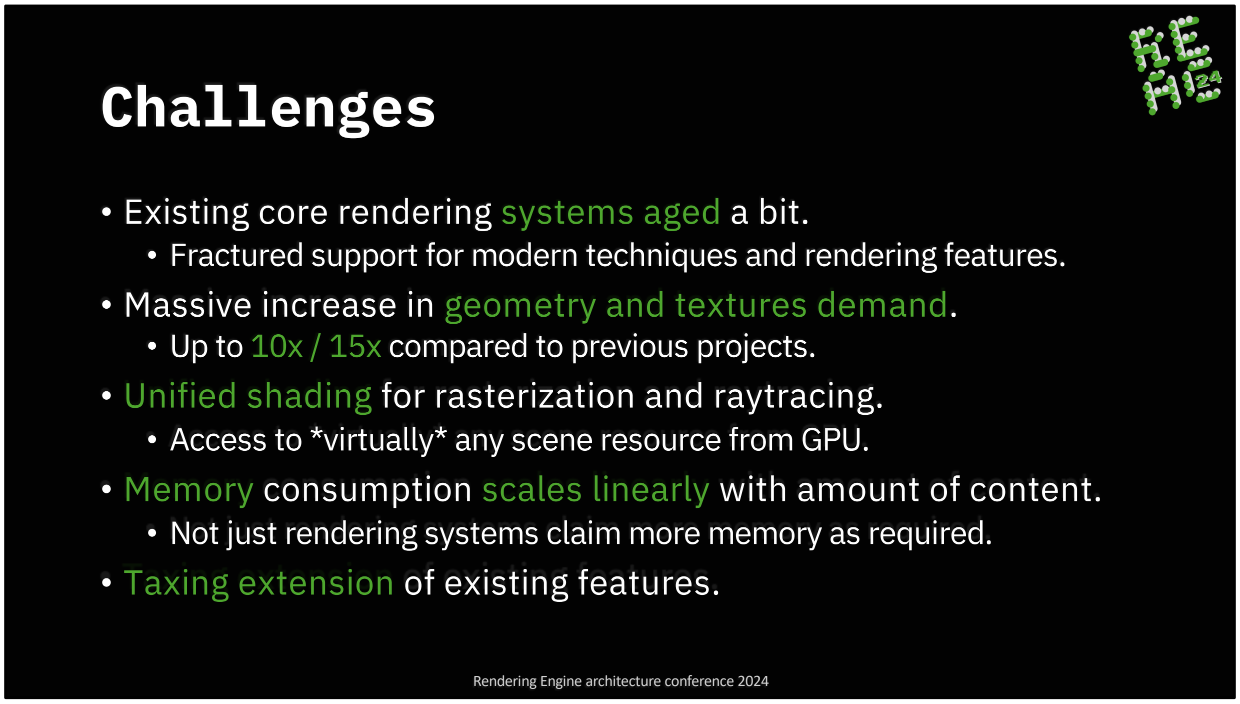
Rendering Engine architecture conference 2024

These days 4A Engine supports PC, Xbox Series, and Play Station 5 only. This is both freeing and challenging at the same time.
On one hand - a small set of high-end target platforms with similar features and capabilities makes it much easier to develop efficient cross-platform solutions.
On the other hand – reliance on high-end features usually leads one into uncharted territory with no established common ways to solve a problem. (We'll be talking more about examples of such cases in few minutes.)

Many aspects of the Engine designed around ray-tracing (including graphics, gameplay, sounds, etc.).
The Enhanced Edition of Metro Exodus was one of the first titles to use fully dynamic run-time global illumination solution based on ray-tracing.
And it's been used and expended since then.

Ray-tracing is a great tool to achieve one of the original design pillars of the Engine – which is to embrace dynamic solutions as much as possible.

As to the graphics technology structure - it is somewhat traditional with dedicated main (aka frontend thread) and rendering (aka backend thread) and extensive worker usage (in a SPU-style).


Challenges

- Existing core rendering systems aged a bit.
  - Fractured support for modern techniques and rendering features.
- Massive increase in geometry and textures demand.
  - Up to 10x / 15x compared to previous projects.
- Unified shading for rasterization and raytracing.
  - Access to *virtually* any scene resource from GPU.
- Memory consumption scales linearly with amount of content.
  - Not just rendering systems claim more memory as required.
- Taxing extension of existing features.

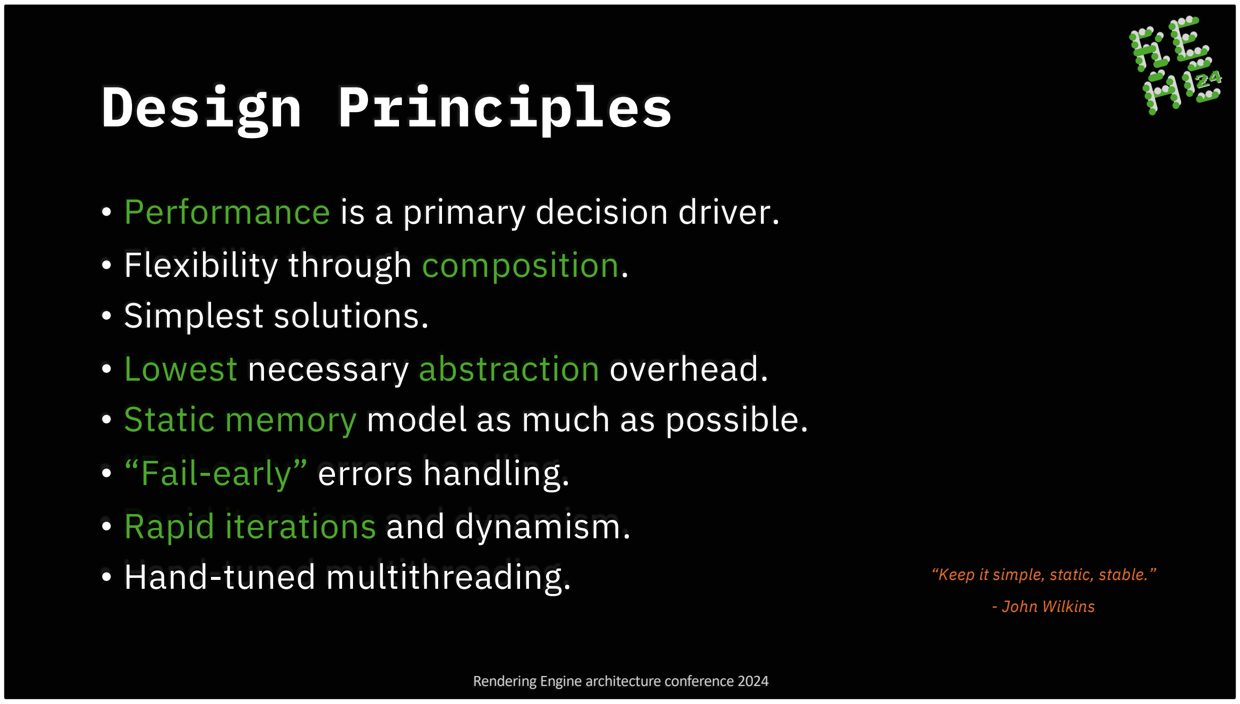
Rendering Engine architecture conference 2024

Reliance on ray-tracing and introduction of more advanced rendering system showed the age of existing low-level graphics code.
Originally, It was designed around DirectX 9 API capabilities and evolved from there. And as it usually happens, with evolution comes extra costs of abstraction leveling.

The biggest shortcoming was on a rendering resources management side as there was a need not only to move towards retain global scene representation but also a necessity to support much more complex and dense scenes.
Which, in turn, added even higher pressure on scene resources streaming.
Furthermore – streaming and resources management parts of the Engine couldn’t rely on commonly-used optimizations opportunities.
An obvious example is that one can’t drop LODs of a geometry behind the camera as it can be touched by ray-tracing. Same goes for lighting, textures, and so on.

To keep memory in check there’s a need to stream rasterization *and* ray-tracing parts of a scene and to do it very fast.

After the thorough review of challenges ahead it became evident that improvements in core resources management and streaming will yield the biggest benefit for the time invested.
The results of these improvements as well as some implementation details will be covered in the next sections of the talk.



Design Principles

- Performance is a primary decision driver.
- Flexibility through composition.
- Simplest solutions.
- Lowest necessary abstraction overhead.
- Static memory model as much as possible.
- “Fail-early” errors handling.
- Rapid iterations and dynamism.
- Hand-tuned multithreading.

“Keep it simple, static, stable.”

- John Wilkins

Rendering Engine architecture conference 2024

While designing these improved tech we relied on a set of core design pillars of the Engine.

High performance has been one of the corner stones of the 4A Engine and there was a desire to expand on it even more because of much higher demands on content side.

To achieve that we rely on close-to-metal abstraction level that still covers all target platforms with least amount of platform-specific code.
As such, there’s no “hardcoded” definition of meshes, materials, even textures in low-level graphics code.
Instead, a set of building blocks exposed to middle-level code which composes them to build what we call a common use-cases.

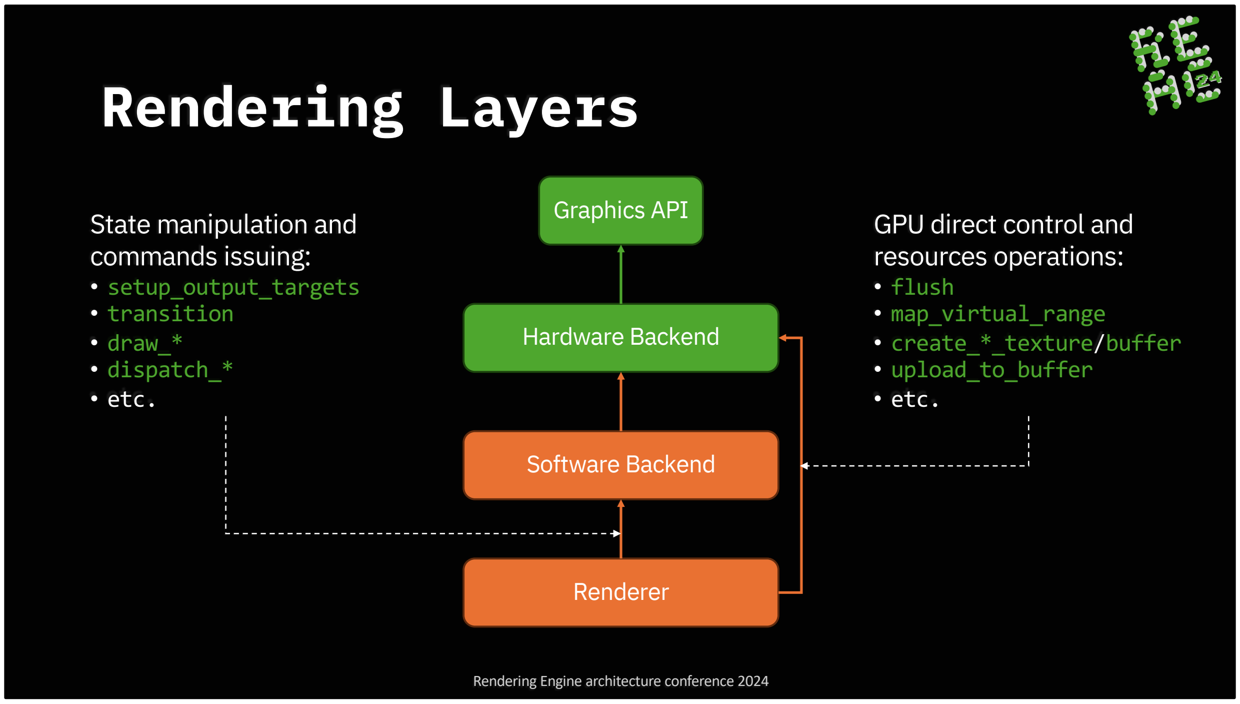
Also, we embrace the “fail-early” approach to error handling. It means that code should report errors as soon as possible.
Where it’s possible, we want errors to show-up during compilation – thus, heavy reliance on static type system.
In cases when correctness can’t be checked at compile-time – validation code

should report issues at the earliest stage of the pipeline.

Finally, if errors still creep-in – we need an efficient instruments to debug and fix them quickly.
So, it’s necessary to ensure that designed solutions are debugging friendly.

Rendering Resources

And with that – let’s move to the first section of the talk. Here I’ll present high level overview of a so-called “bare” approach to low-level rendering resources as well as some implementation details for them.



Rendering Layers

State manipulation and commands issuing:

- `setup_output_targets`
- `transition`
- `draw_*`
- `dispatch_*`
- `etc.`

GPU direct control and resources operations:

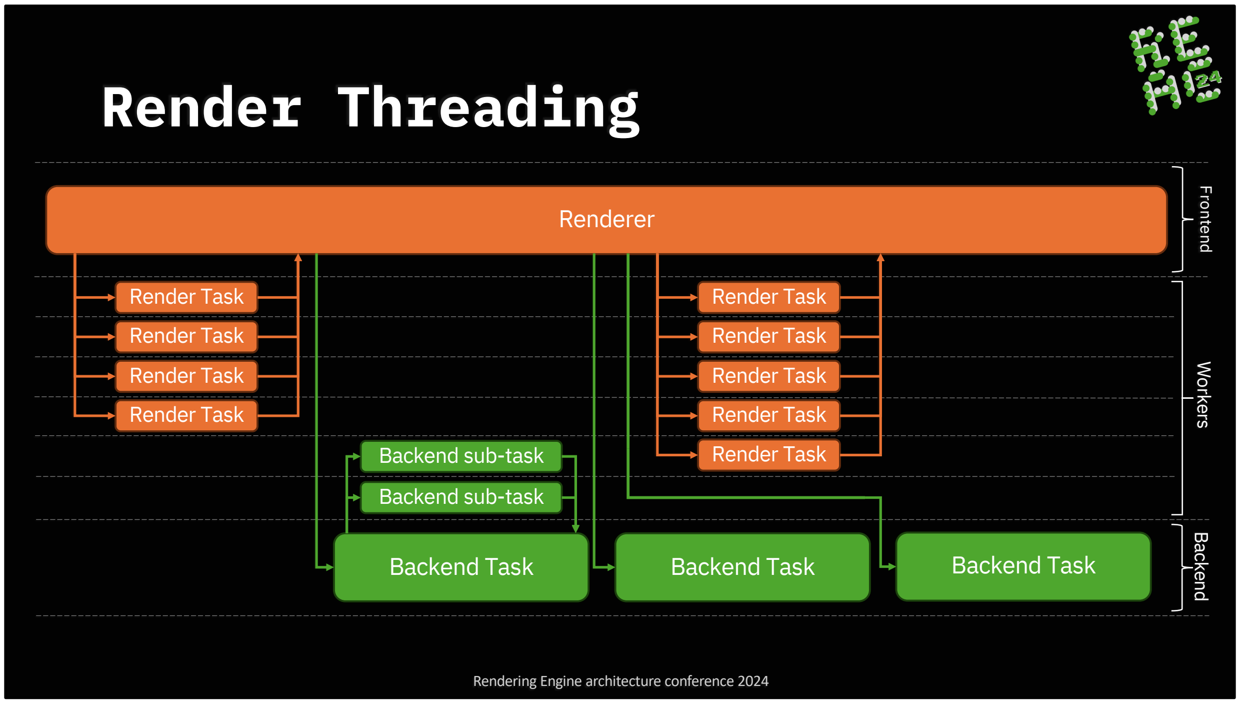
- `flush`
- `map_virtual_range`
- `create_*_texture/buffer`
- `upload_to_buffer`
- `etc.`

Rendering Engine architecture conference 2024

Rendering resources are used on almost every level of the graphics technology.

In 4A Engine graphics code structured as three aggregations operating on multiple levels:

- Renderer itself – which can be logically split in two parts:
    - The middle-level part - it implements resources management and streaming, use-cases definitions, etc.
        - Software Backend is one the middle-level component. It drives command lists recording.
    - The high-level part – it provides a frame loop, rendering passes, and so on.
        - It also contains interface which is used by other Engine parts to provide data for the scene description.
- Hardware Backend – which handles interaction with low-level resources, execution of recorded commands, memory manipulation, etc.

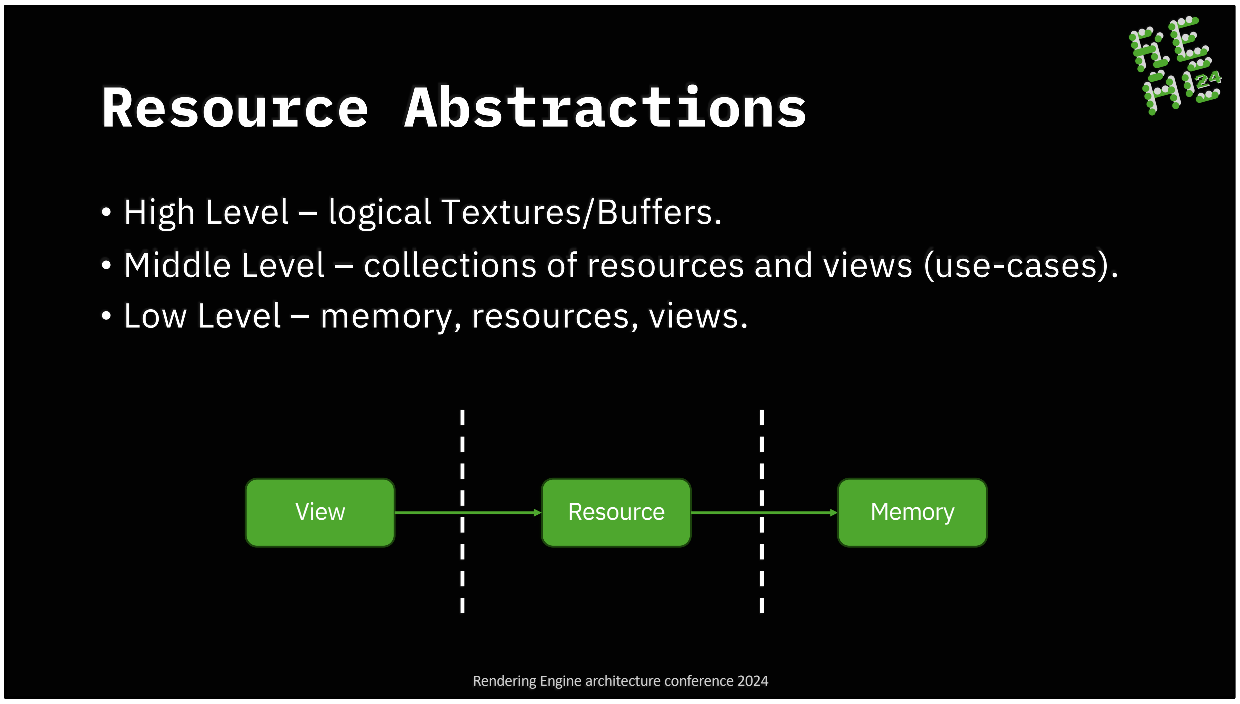

Render Threading

Rendering layers are loosely mapped onto dedicated Frontend and Backend threads. These threads are pipelined and utilize fork and join model quite a lot to distribute work to other threads.

Frontend thread produces tasks for backend to process. It can spawn secondary tasks which don't submit anything to backend directly. (Although, they do access graphics API abstraction if needed.)
Backend thread, primarily, polls on tasks for execution. Upon receiving – processes them with highest priority. If no tasks available – helps other systems with their tasks.

[OPTIONAL]

To give you an example of threaded processing of these layers:
* Renderer can use secondary tasks to pre-sort and fill-in skinning buffer and once it's ready, schedule a backend task to upload delta change by running a compute shader dispatch.
* While backend can spawn additional tasks to cull and select visible TLAS instances for current frame and update the in-place buffer for TLAS building.



Resource Abstractions

- High Level – logical Textures/Buffers.
- Middle Level – collections of resources and views (use-cases).
- Low Level – memory, resources, views.

View
Resource
Memory

Rendering Engine architecture conference 2024

Rendering resources inherit rendering layers abstraction.

At the very core of the graphics technology, we settled with design which is similar in spirit to DirectX 12 and Vulkan.
It defines memory, resources, and views as separate concepts and exposes them. They're implemented as a part of Rendering Backend.

[OPTIONAL]

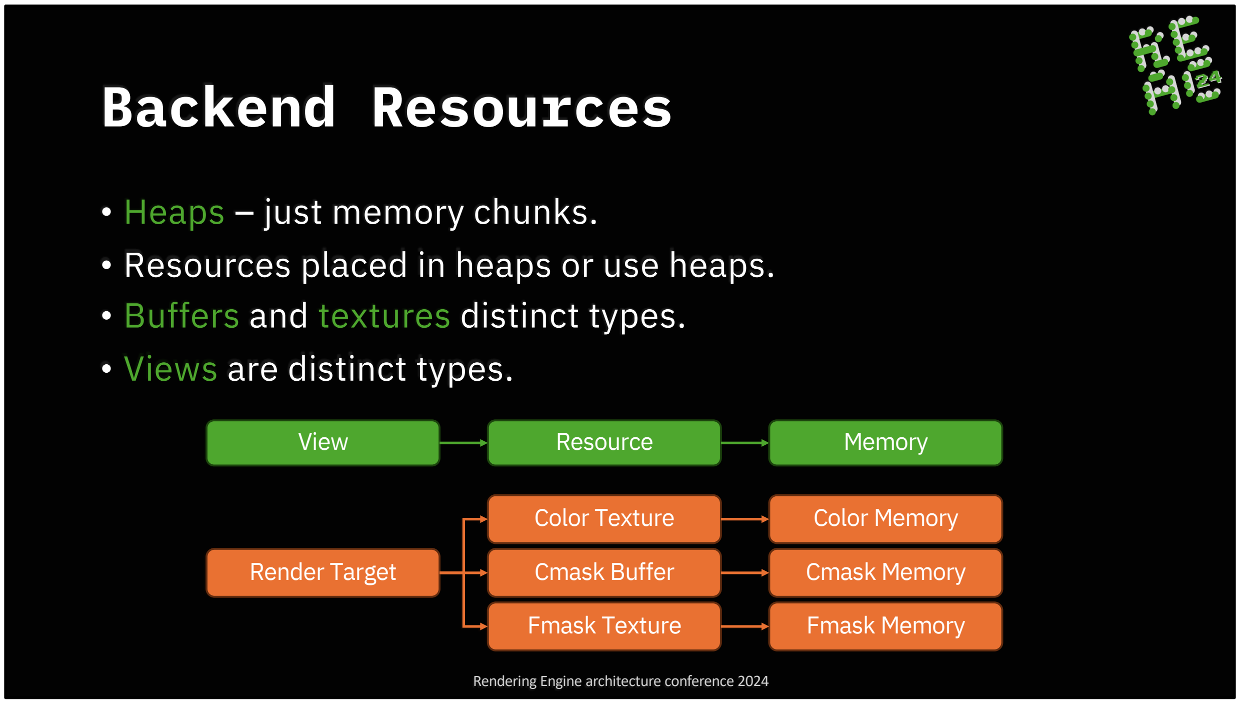
Frankly, we'd prefer to have only memory and resources as views are, largely, just typed pointers to resources with extra metadata.
But, unfortunately, on PC we had to use them to achieve desired level of flexibility. In the end – it proved to be quite efficient, although should there be a possibility to replace views with a direct access to resource descriptors, we'll gladly use it.

[OPTIONAL]

For cross-platform code they appear as opaque data types. One can only hold them to control lifetime, transfer ownership, check for validity, and supply as arguments to higher level graphics code.

These three concepts used by middle-level graphics code to build common “use-cases” – such as: read-only stream-able textures, virtual textures, etc. Those use-cases are just a simple combination of resources and views (sometimes with manual memory handling or extra logic).

Gameplay features, UI elements, and other game-level systems operate with logical textures – which are just an example of a use-case.



Backend Resources

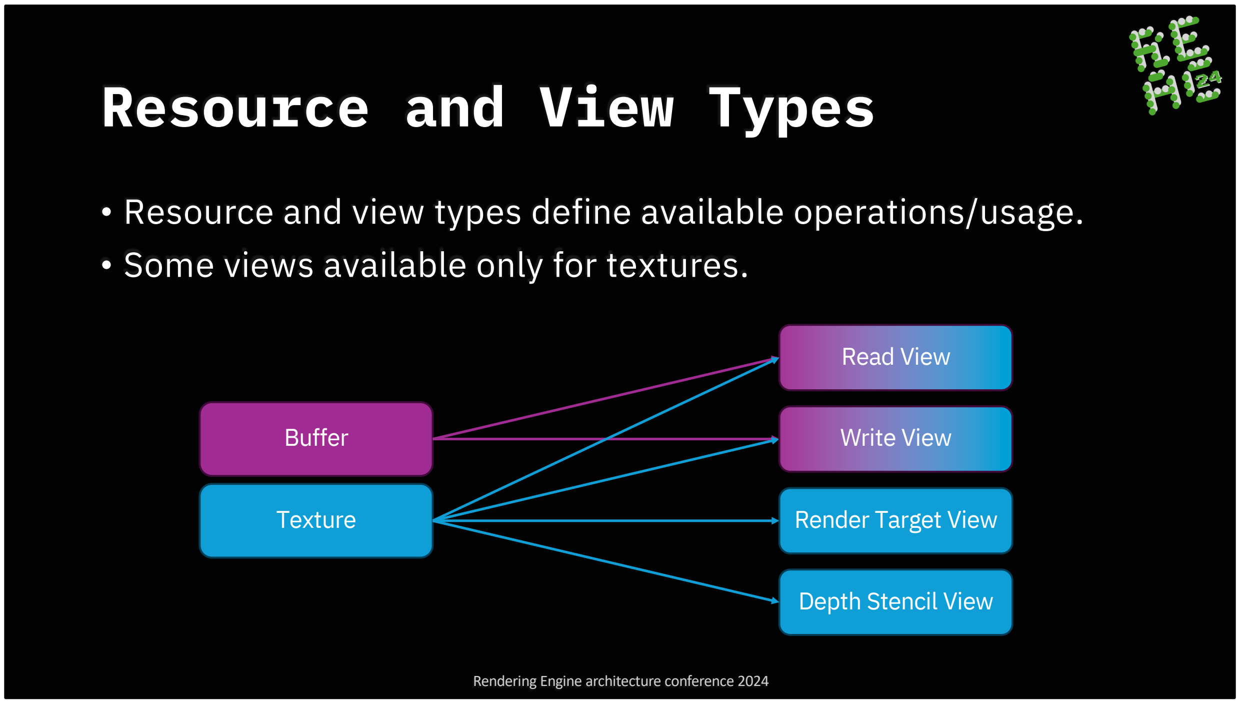
- Heaps – just memory chunks.
- Resources placed in heaps or use heaps.
- Buffers and textures distinct types.
- Views are distinct types.

View
Resource
Memory
Render Target
Color Texture
Cmask Buffer
Fmask Texture
Color Memory
Cmask Memory
Fmask Memory

Rendering Engine architecture conference 2024

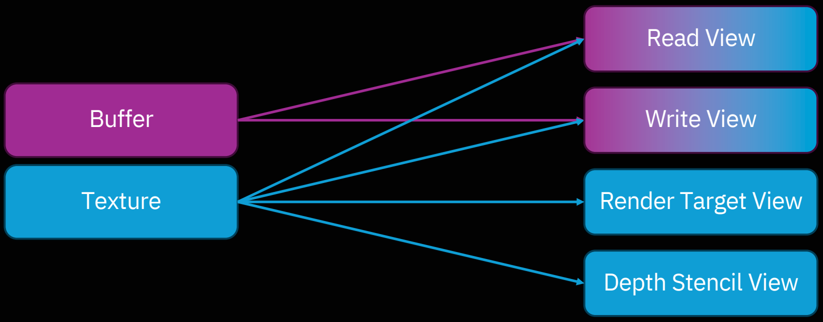
Resource and view types are similar to those defined in Explicit Graphics APIs with several major differences:

- It is possible to access metadata buffers and textures (such as CMask, DCC buffers, etc.) directly from cross-platform code (where supported).
- Resources and views follow strongly-typed semantics. (They are separate types with no implicit conversion between them.)
- Actual memory addressing preferred over intermediate abstractions and used whenever possible. (Acceleration Structure are just virtual addresses – for instance.)
- Also, there's no of sub-resources – views, effectively, represent them implicitly.
- And much more.



Resource and View Types

- Resource and view types define available operations/usage.
- Some views available only for textures.

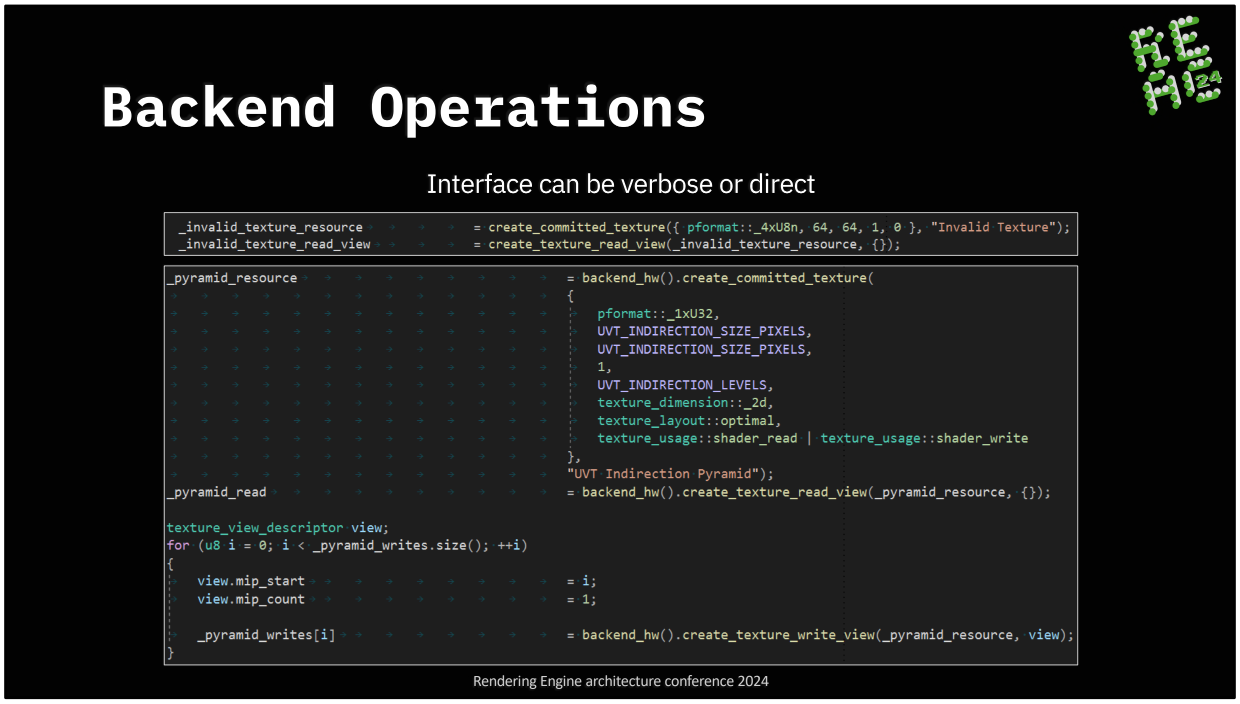


Rendering Engine architecture conference 2024

Having a strong-typed semantics for resources and views makes it possible to explicitly define supported operations and usage based on C++ type alone.
For example:

- One can't set texture resource as render target – it must be a render target view (there's no way to call a function – types don't match).
- One can't transition buffer to a depth-stencil state.
- All of these will be a compile time errors (in most cases).

Many errors and usage violations detected before code gets compiled. (Offline code parsers in IDEs will highlight those).



Backend Operations

Interface can be verbose or direct

```
 _invalid_texture_resource          = create_committed_texture({ pformat::_4xU8n, 64, 64, 1, 0 }, "Invalid Texture");
 _invalid_texture_read_view         = create_texture_read_view(_invalid_texture_resource, {});
```

```
_pyramid_resource                                     = backend_hw().create_committed_texture(
                                                      {
                                                          pformat::_1xU32,
                                                          UVT_INDIRECTION_SIZE_PIXELS,
                                                          UVT_INDIRECTION_SIZE_PIXELS,
                                                          1,
                                                          UVT_INDIRECTION_LEVELS,
                                                          texture_dimension::_2d,
                                                          texture_layout::optimal,
                                                          texture_usage::shader_read | texture_usage::shader_write
                                                      },
                                                      "UVT Indirection Pyramid");
_pyramid_read                                         = backend_hw().create_texture_read_view(_pyramid_resource, {});

texture_view_descriptor view;
for (u8 i = 0; i < _pyramid_writes.size(); ++i)
{
    view.mip_start                                    = i;
    view.mip_count                                    = 1;

    _pyramid_writes[i]                                = backend_hw().create_texture_write_view(_pyramid_resource, view);
}
```

Rendering Engine architecture conference 2024

Here, I'd like to give you an example of code operating with these concepts. Which, hopefully, will give some substance to the wordy explanation.

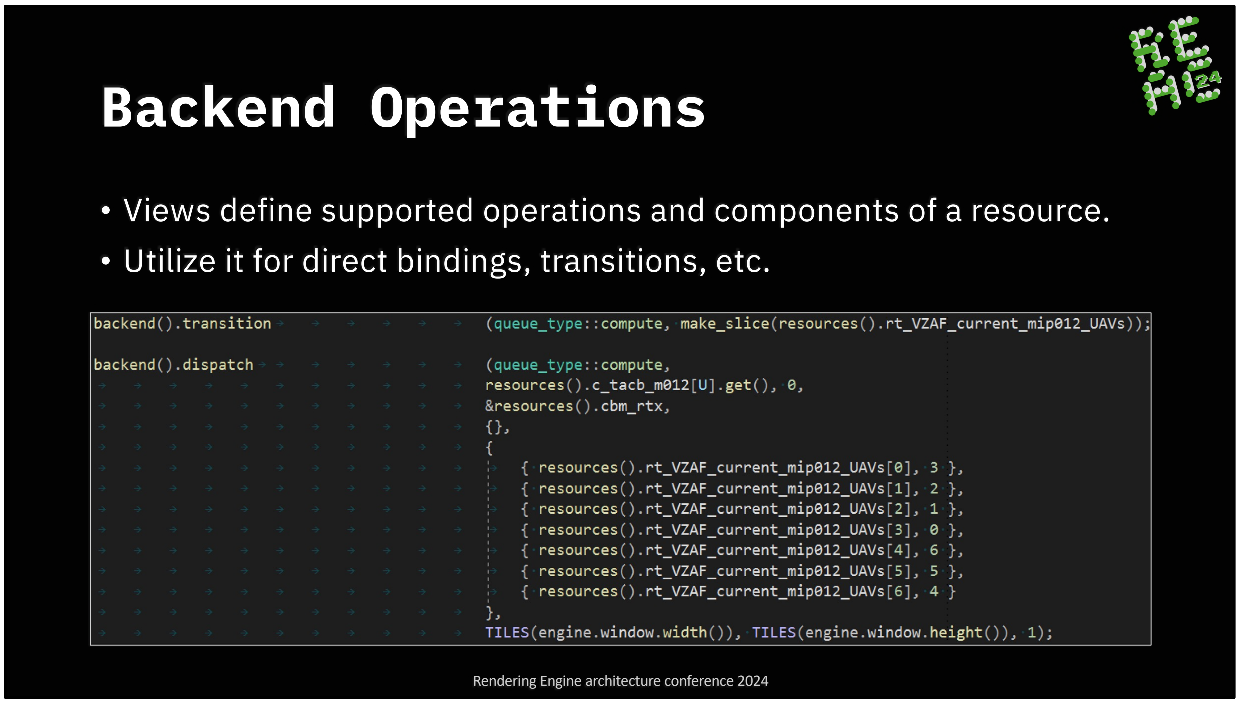
The top one is an example of a very simple "invalid" texture creation which is used to replace a missing resource:

- There's just a resource and single read view.

The bottom one is an example of somewhat advanced manipulation:

- It creates texture optimized for specific layout and usage patters as well as one read view covering entire resource and a list of write views (aka UAVs) per MIP level.

This show a duality of selected interface which can be very verbose when needed, or it can be very brief and simple.



Backend Operations

- Views define supported operations and components of a resource.
- Utilize it for direct bindings, transitions, etc.

```
backend().transition                    (queue_type::compute, make_slice(resources().rt_VZAF_current_mip012_UAVs));

backend().dispatch                      (queue_type::compute,
                                        resources().c_tacb_m012[U].get(), 0,
                                        &resources().cbm_rtx,
                                        {},
                                        {
                                            { resources().rt_VZAF_current_mip012_UAVs[0], 3 },
                                            { resources().rt_VZAF_current_mip012_UAVs[1], 2 },
                                            { resources().rt_VZAF_current_mip012_UAVs[2], 1 },
                                            { resources().rt_VZAF_current_mip012_UAVs[3], 0 },
                                            { resources().rt_VZAF_current_mip012_UAVs[4], 6 },
                                            { resources().rt_VZAF_current_mip012_UAVs[5], 5 },
                                            { resources().rt_VZAF_current_mip012_UAVs[6], 4 }
                                        },
                                        TILES(engine.window.width()), TILES(engine.window.height()), 1);
```

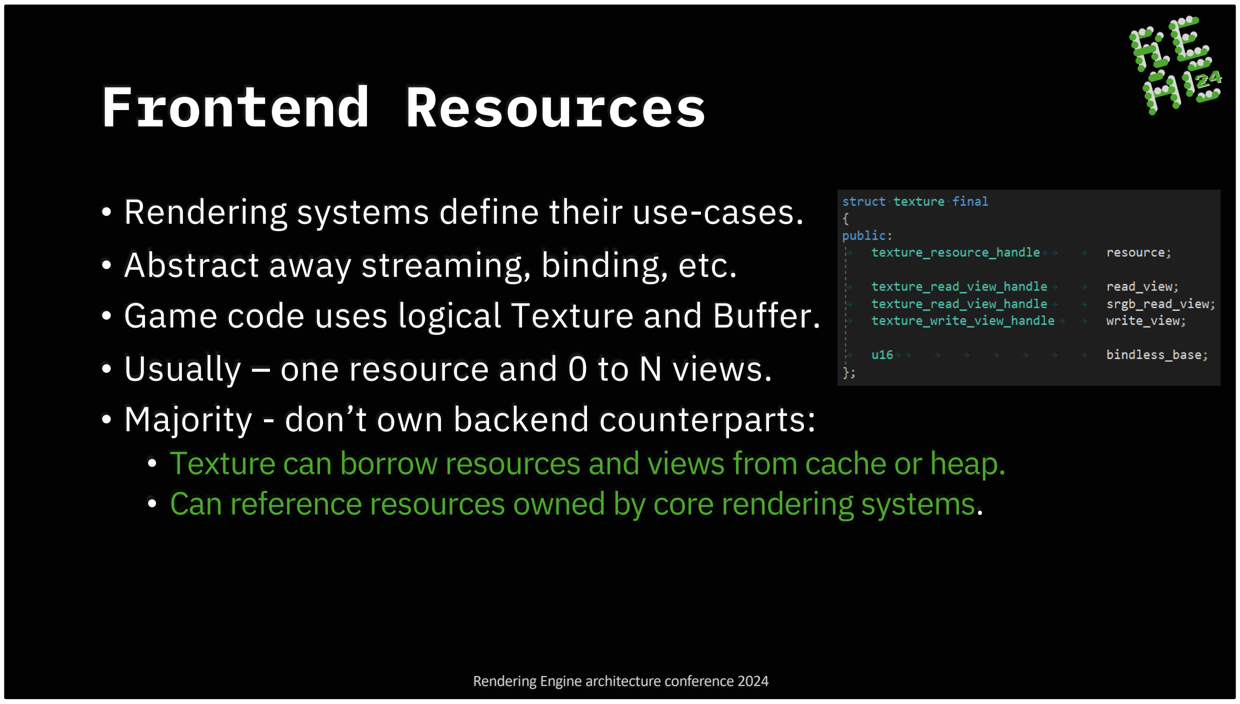
Rendering Engine architecture conference 2024

Since views are available as a first-class objects – they are utilized to simplify code as well.

In this example we schedule a simple dispatch call which writes into multiple MIP levels of a texture.
(Please, don't mind naming here – they are irrelevant and are here for demonstration purposes.)

The first line transitions selected MIP levels into shader write state. One can notice that there's no specification of target state.
The state deduced from view type at compile time so that one can't make a mistake of using incorrect one.
(` make_slice` here just creates a C++ array_view.)

The next statement schedules a dispatch specifying which MIP levels and where they'll be bound.



Frontend Resources

- Rendering systems define their use-cases.
- Abstract away streaming, binding, etc.
- Game code uses logical Texture and Buffer.
- Usually – one resource and 0 to N views.
- Majority - don't own backend counterparts:
  - Texture can borrow resources and views from cache or heap.
  - Can reference resources owned by core rendering systems.

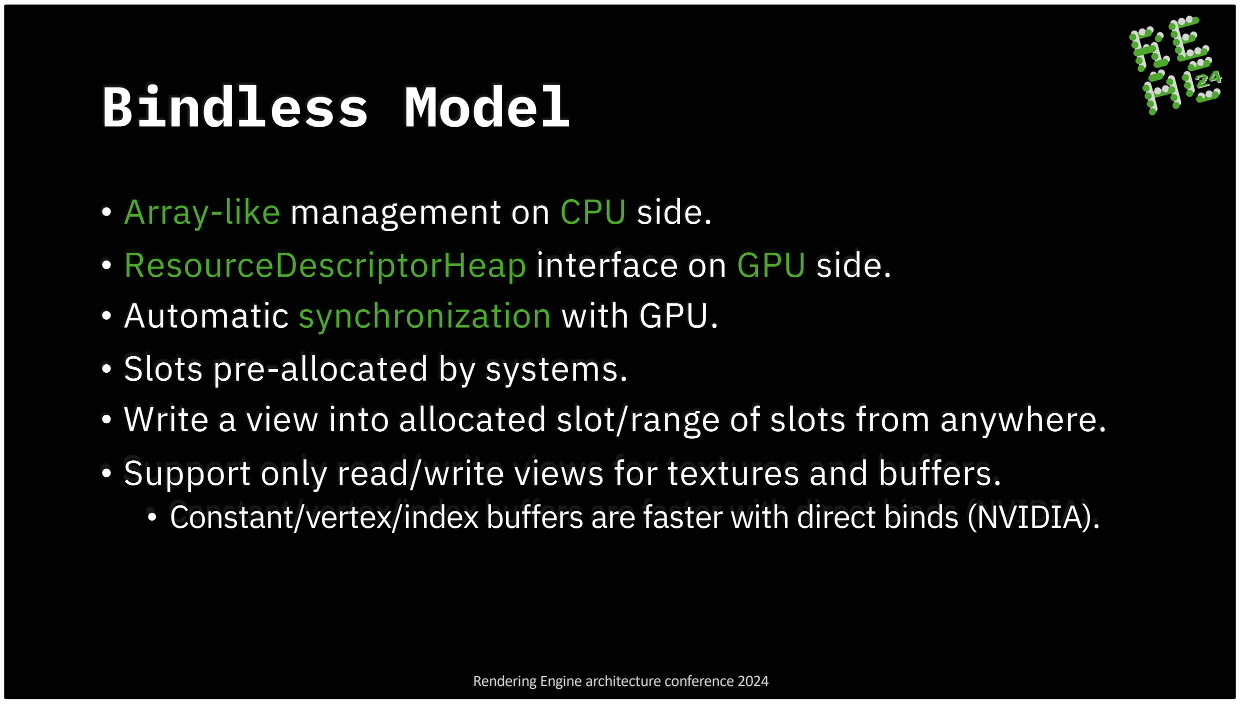
```
struct texture final
{
public:
    texture_resource_handle         resource;

    texture_read_view_handle        read_view;
    texture_read_view_handle        srgb_read_view;
    texture_write_view_handle       write_view;

    u16                             bindless_base;
};
```

Rendering Engine architecture conference 2024

Higher-level graphics code is free to decide what composition of heaps, resources, and views will represent their use-case or work-unit.

For example, if system provides a functionality to read-back data produced on GPU – there's no need for any views. It can manipulate with texture/buffer resources directly.
Or, if a system implements a texture streaming – there's no need to manage write views, or render target views, etc. It can be just a memory heap and a bunch of texture resources and read views which are moved around from one streaming unit to another.
Finally, if system relies on bindless model completely, it can be just a resource with bindless slot allocated for it.

This way, there's no overly-complicated "Uber Texture" class that supports all types of usage patters from read only textures to depth stencil targets.



Bindless Model

- Array-like management on CPU side.
- ResourceDescriptorHeap interface on GPU side.
- Automatic synchronization with GPU.
- Slots pre-allocated by systems.
- Write a view into allocated slot/range of slots from anywhere.
- Support only read/write views for textures and buffers.
  - Constant/vertex/index buffers are faster with direct binds (NVIDIA).

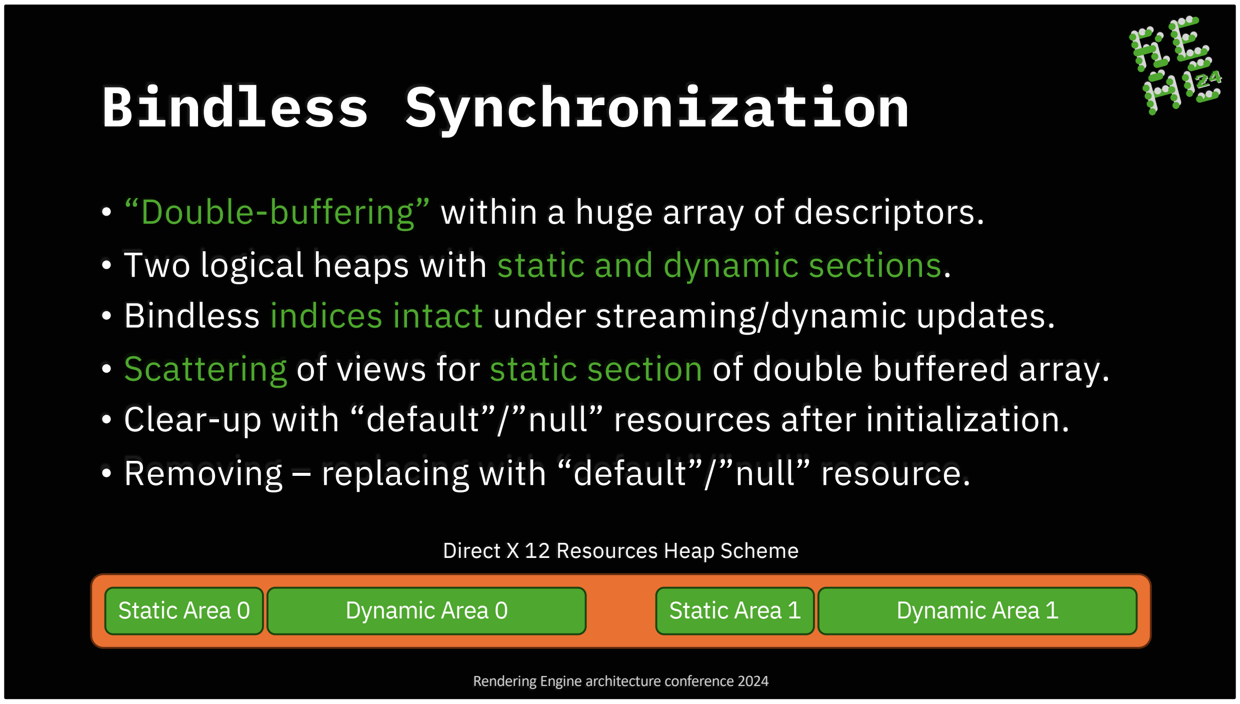
Rendering Engine architecture conference 2024

To provide global access to texture data in ray-tracing passes 4A Engine relies on bindless resources model.

Following the lowest possible abstraction principle – backend implements common interface for it, but it doesn't enforce any management specifics and doesn't impose constraints beside number of addressable slots.

From the "user-land" code perspective it's just a static array of views. There's a single-entry interface which is to write resource view to a specific location within it.

While all view types supported - in practice, bindless model utilized for texture read and writes views mostly.
Because of the way how descriptor indexing implemented on IHV side, bindless is not a great suit for buffers and one can be better off with buffers suballocation to achieve global indexing.



Bindless Synchronization

- “Double-buffering” within a huge array of descriptors.
- Two logical heaps with static and dynamic sections.
- Bindless indices intact under streaming/dynamic updates.
- Scattering of views for static section of double buffered array.
- Clear-up with “default”/”null” resources after initialization.
- Removing – replacing with “default”/”null” resource.

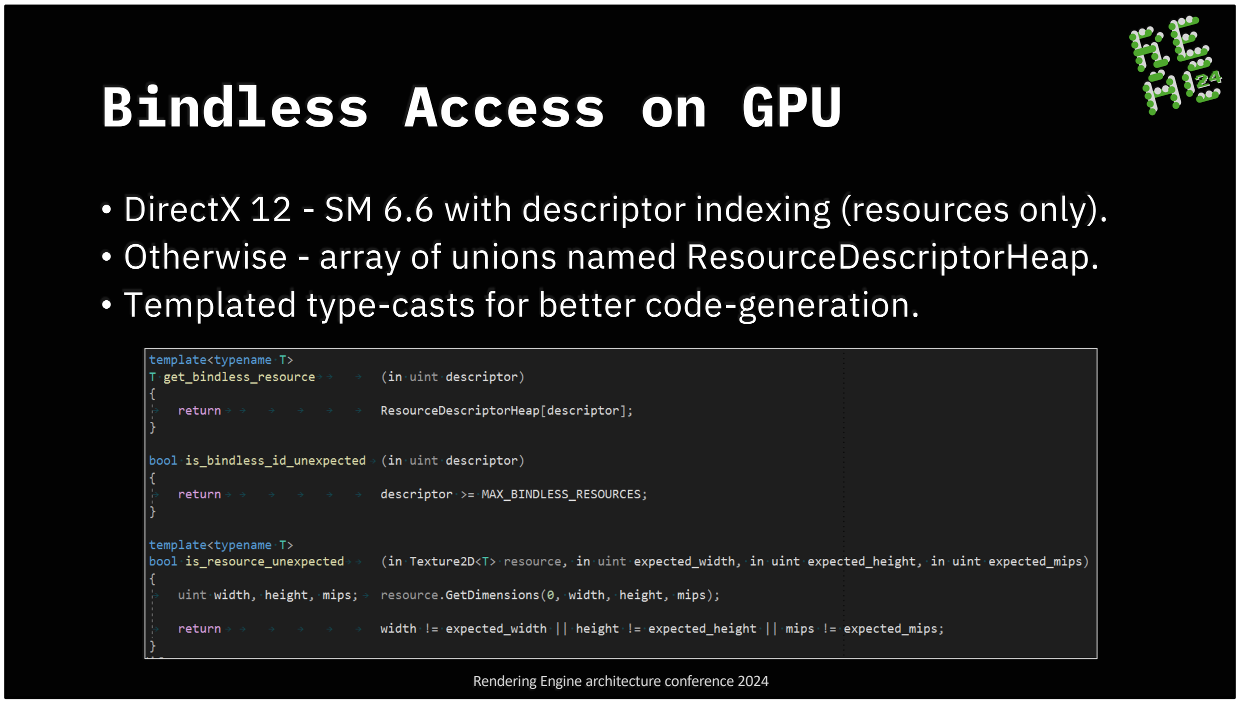
Rendering Engine architecture conference 2024

Using a static allocation strategy ensures that resource persists in same bindless slot for its’ entire lifetime.
This is very powerful as it allows to cache bindless IDs in memory (either CPU or GPU).

But then there’s a question of dynamic resources and stream-able resources. Usually, streaming means that actual resource will change upon quality transition.

To keep logical resource in same slot and to avoid indices “patching” in cached location – instead, we implemented double-buffering of the bindless array in backend.

Views are flip between back and front versions. When resource content changes, we just overwrite the view in back-array. (GPU still uses front array and not affected).
Then, when buffers flipped, write operation repeated on the new back array.



Bindless Access on GPU

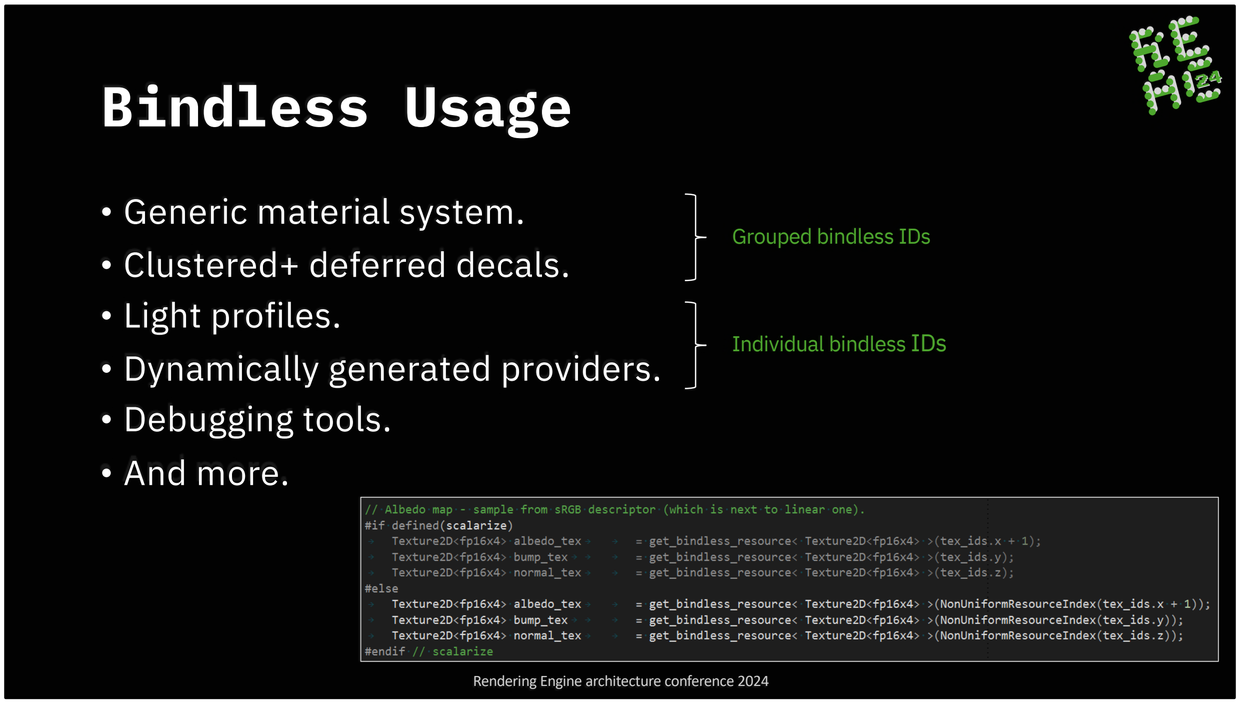
- DirectX 12 - SM 6.6 with descriptor indexing (resources only).
- Otherwise - array of unions named ResourceDescriptorHeap.
- Templated type-casts for better code-generation.

```
template<typename T>
T get_bindless_resource             (in uint descriptor)
{
    return                          ResourceDescriptorHeap[descriptor];
}

bool is_bindless_id_unexpected      (in uint descriptor)
{
    return                          descriptor >= MAX_BINDLESS_RESOURCES;
}

template<typename T>
bool is_resource_unexpected         (in Texture2D<T> resource, in uint expected_width, in uint expected_height, in uint expected_mips)
{
    uint width, height, mips;       resource.GetDimensions(0, width, height, mips);

    return                          width != expected_width || height != expected_height || mips != expected_mips;
}
```

Rendering Engine architecture conference 2024

On GPU side, bindless model follows CPU representation. Resources exposed as a simple array with a bunch of casting functions.
For DirectX 12 it's implemented with dynamic descriptor indexing in ResourceDescriptorHeap.
On consoles it's just a globally visible buffer named ResourceDescriptorHeap that contains union of buffer or texture.

Templated accessors help to ensure that code generation won't issue codependent reads on Vector Pipeline.
(Wave PSSL Compiler tend to do it with in-place resources creation.)
They're also used to inject validation code if needed for debugging.



Bindless Usage

- Generic material system.
- Clustered+ deferred decals.
- Light profiles.
- Dynamically generated providers.
- Debugging tools.
- And more.

Grouped bindless IDs
Individual bindless IDs

```
// Albedo map - sample from sRGB descriptor (which is next to linear one).
#if defined(scalarize)
    Texture2D<fp16x4> albedo_tex        = get_bindless_resource< Texture2D<fp16x4> >(tex_ids.x + 1);
    Texture2D<fp16x4> bump_tex          = get_bindless_resource< Texture2D<fp16x4> >(tex_ids.y);
    Texture2D<fp16x4> normal_tex        = get_bindless_resource< Texture2D<fp16x4> >(tex_ids.z);
#else
    Texture2D<fp16x4> albedo_tex        = get_bindless_resource< Texture2D<fp16x4> >(NonUniformResourceIndex(tex_ids.x + 1));
    Texture2D<fp16x4> bump_tex          = get_bindless_resource< Texture2D<fp16x4> >(NonUniformResourceIndex(tex_ids.y));
    Texture2D<fp16x4> normal_tex        = get_bindless_resource< Texture2D<fp16x4> >(NonUniformResourceIndex(tex_ids.z));
#endif // scalarize
```

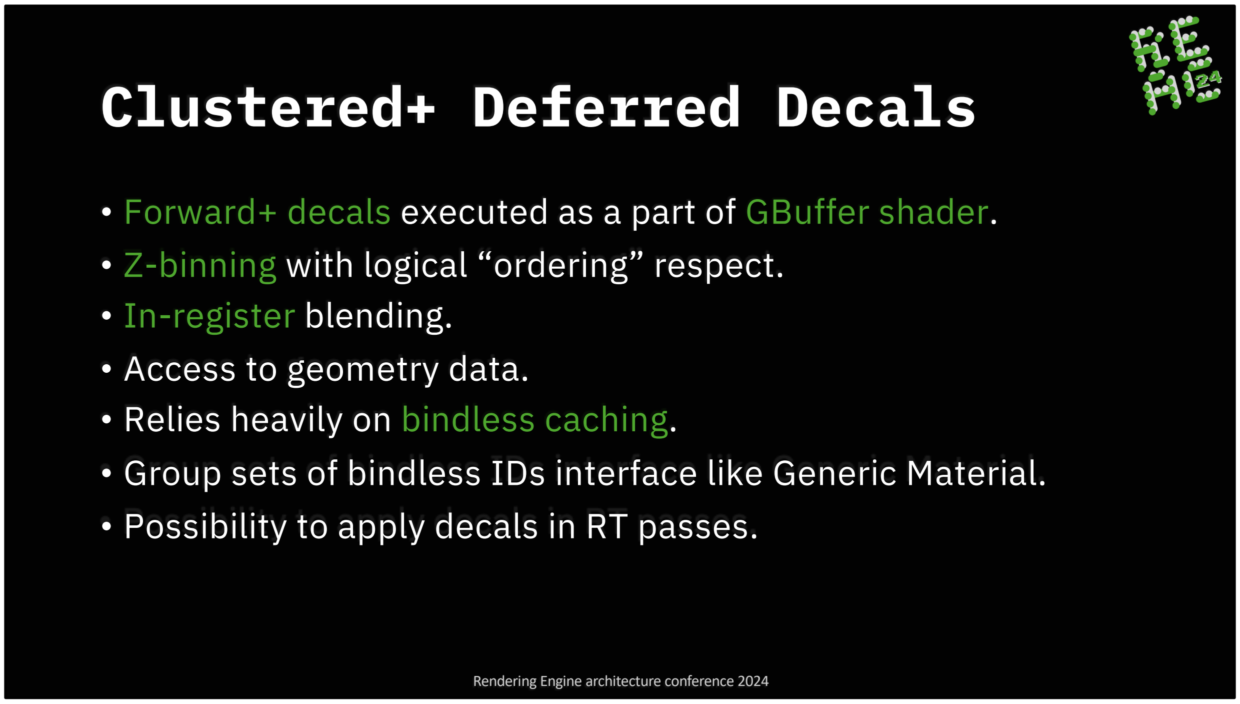
Rendering Engine architecture conference 2024

[OPTIONAL]

In 4A Engie bindless usage patters can be split in two categories:
- Accessing groups of bindless resources with a group slot ID.
    - Simple allocation strategy that puts related bindless resources in adjacent slots (for example – PBR set of textures, like on the figure at the bottom of the slide).
    - It's useful because it can be used as an implicit material ID. It's utilized for GPU access statistics writing for textures streaming.
- Accessing resources using individual slot ID.
    - A bit more taxing on tracking side but has finer granularity of access.

[OPTIONAL]

There's a fair number of use-cases for bindless model in 4A Engine and we'll use just two of them to give you a hint of the model exposure and level of integration: Those being:
- Clustered+ deferred decals as an example of grouped bindless IDs.
- And Analytical light profiles as an example of individual bindless IDs.

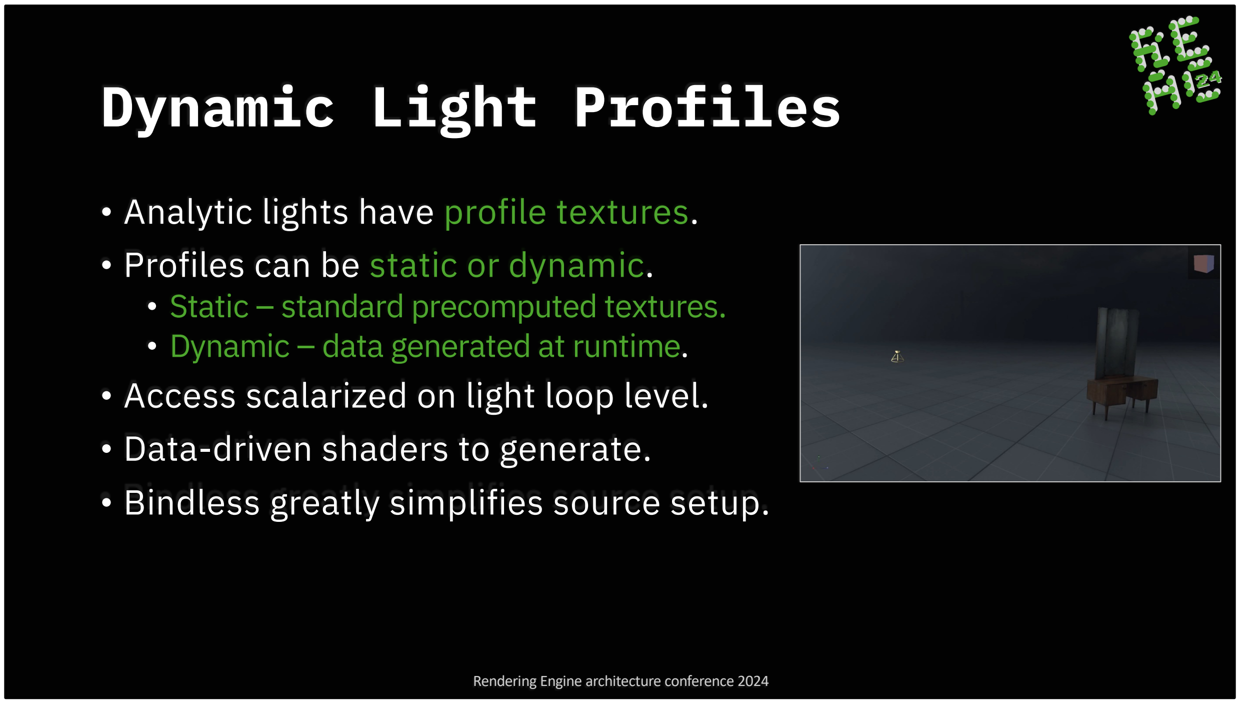

Clustered+ Deferred Decals

- Forward+ decals executed as a part of GBuffer shader.
- Z-binning with logical “ordering” respect.
- In-register blending.
- Access to geometry data.
- Relies heavily on bindless caching.
- Group sets of bindless IDs interface like Generic Material.
- Possibility to apply decals in RT passes.

Rendering Engine architecture conference 2024

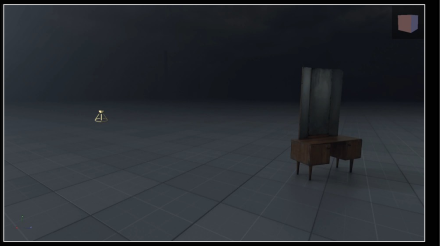
Clustered+ Deferred Decals is a primary decaling solution of the engine – designed for high performance with thousands of items per view.
It’s a bit uncommon approach as it’s, essentially, a Forward+-style decals embedded in Gbuffer shader and executed at the very end of it.
This approach has multiple benefits:
- Trivial occlusion culling.
- Fast blending.
- Minimal overdraw.
- Perfect derivative.
- Access to instance parameters.

It’s an efficient tool which shines in combination with bindless resources caching.
We can use unified access to generic material data to turn any set of textures into a decal with no material duplication, memory waste or extra code.



Dynamic Light Profiles

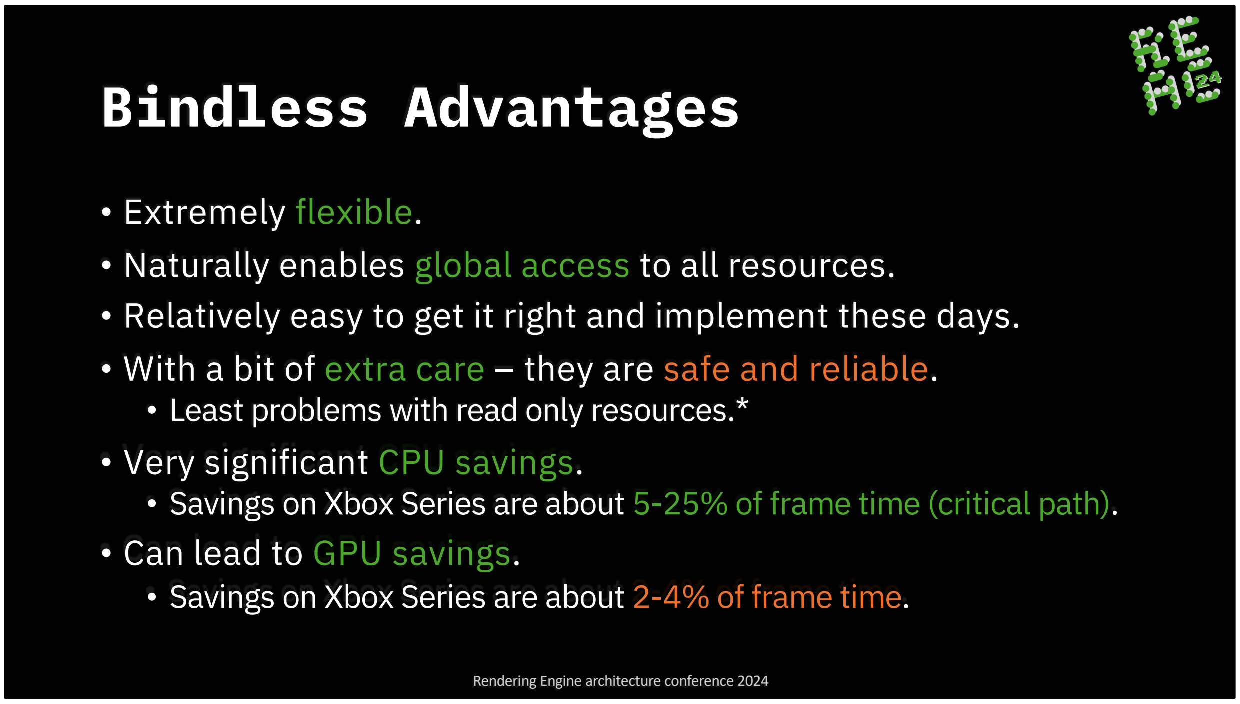
- Analytic lights have profile textures.
- Profiles can be static or dynamic.
  - Static – standard precomputed textures.
  - Dynamic – data generated at runtime.
- Access scalarized on light loop level.
- Data-driven shaders to generate.
- Bindless greatly simplifies source setup.



Rendering Engine architecture conference 2024

For light profiles, this model unlocks several opportunities:

- First one being freedom of source specification. It's trivial to use transient resources as a light profiles (Render Targets, per-frame generated data, etc.).
  - For example, video to the right shows an example of run-time decoded video supplied as a light profile.
  - Same mechanism utilized to generate dynamic light profiles from data driven shaders and much more.
- Another one – consistency. Once bindless ID is it's applied consistently in all lighting passes: from direct lighting, to volumetrics and ray-traced lighting (which can be seen in the video clip).
  - There's no need to bind these textures in multiple rendering passes, no need to adjust shader code for it. "It just works."
- Finally - memory optimization, as it's not necessary to keep all profiles in texture array for dynamic indexing.



Bindless Advantages

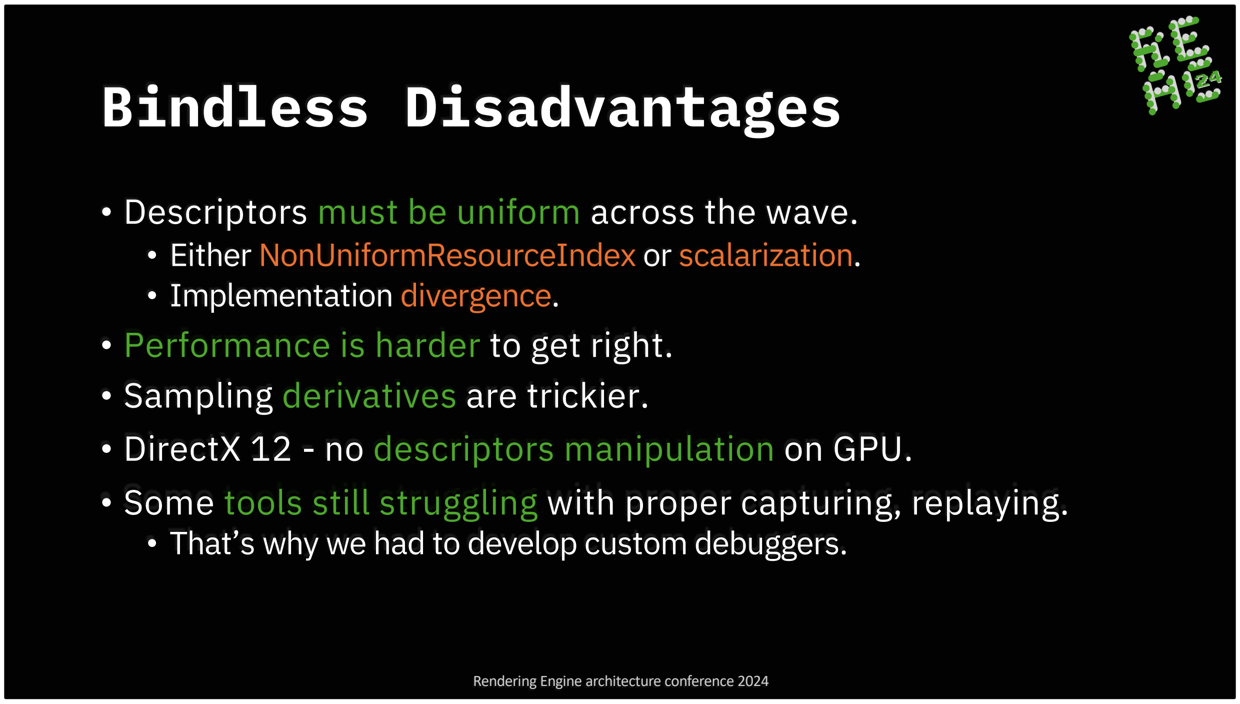
- Extremely flexible.
- Naturally enables global access to all resources.
- Relatively easy to get it right and implement these days.
- With a bit of extra care – they are safe and reliable.
  - Least problems with read only resources.*
- Very significant CPU savings.
  - Savings on Xbox Series are about 5-25% of frame time (critical path).
- Can lead to GPU savings.
  - Savings on Xbox Series are about 2-4% of frame time.

Rendering Engine architecture conference 2024

One can't stress enough how flexible bindless model is. Although, this should be no news for the audience.
It unlocks global access to all resources which is a must for ray-tracing and very useful in many other cases as we, hopefully, just showed.

It has been around for a while and, while it still has drawbacks and some differences in across IHVs – it's ready to be used as primary resource addressing mechanism on GPU.
In fact, with a bit of effort it can outperform traditional direct binding not only on CPU but on GPU as well.

We've seen a measurable performance saving on GPU after rearranging code to hide latency from extra indirection.


Bindless Disadvantages

- Descriptors must be uniform across the wave.
  - Either NonUniformResourceIndex or scalarization.
  - Implementation divergence.
- Performance is harder to get right.
- Sampling derivatives are trickier.
- DirectX 12 - no descriptors manipulation on GPU.
- Some tools still struggling with proper capturing, replaying.
  - That's why we had to develop custom debuggers.

Rendering Engine architecture conference 2024

But, as with any solution, there're some issues and potential improvements which, hopefully, can get more attention with wider adoption of bindless model.

On AMD hardware one must place resource descriptors in SGPR. Modern NVIDIA hardware these days has no such requirement.
This makes it so optimization for one sometimes means pressurization for other.

To keep code somewhat shared, one must make descriptors uniform across the wave.
So, it's either manual scalarization, or decoration with `NonUniformResourceIndex`.

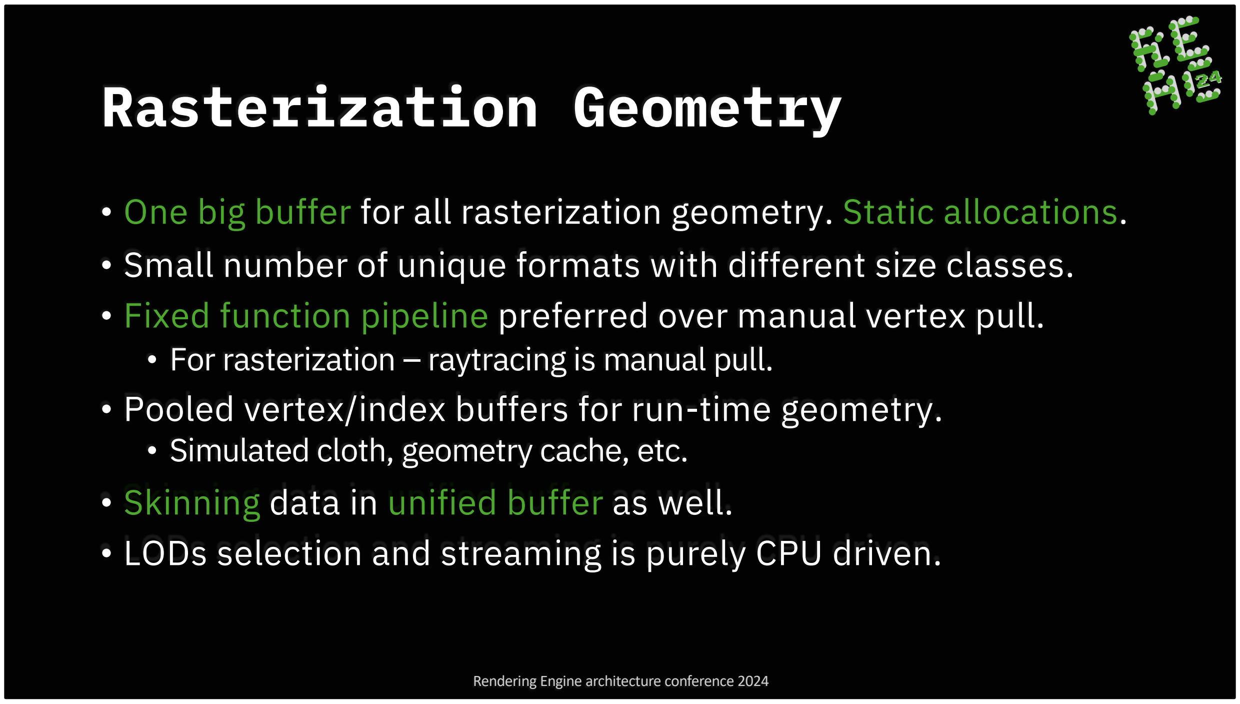
Both options are scalarization - which usually implies that helper threads in pixel shaders can't provide reliable derivatives for texture sampling unless helper lanes masked out.
But by doing so on PC breaks invalidates scalarization. Thus, it's up to a programmer to either support multiple sampling implementations, or compute derivatives manually, or fallback to a single mip-level.
(Although, it must be noted than on PC AMD hardware helper lanes do have well-

defined derivatives *if* there’s no dependent memory reads).

Finally, some GPU debugging tools still have issues with bindless resources capturing and debugging. Biggest of them is non-deterministic software-replay and very slow replays.

Management and Streaming

Now, having a way to put resources in memory and to address them is just a first step.
Next up, we need to manage these resources and to stream them in and out.
This section will go into details of different resource types and their management.



Rasterization Geometry

- One big buffer for all rasterization geometry. Static allocations.
- Small number of unique formats with different size classes.
- Fixed function pipeline preferred over manual vertex pull.
  - For rasterization – raytracing is manual pull.
- Pooled vertex/index buffers for run-time geometry.
  - Simulated cloth, geometry cache, etc.
- Skinning data in unified buffer as well.
- LODs selection and streaming is purely CPU driven.

Rendering Engine architecture conference 2024

For rasterization geometry - index and vertex data stored in one big buffer with access.

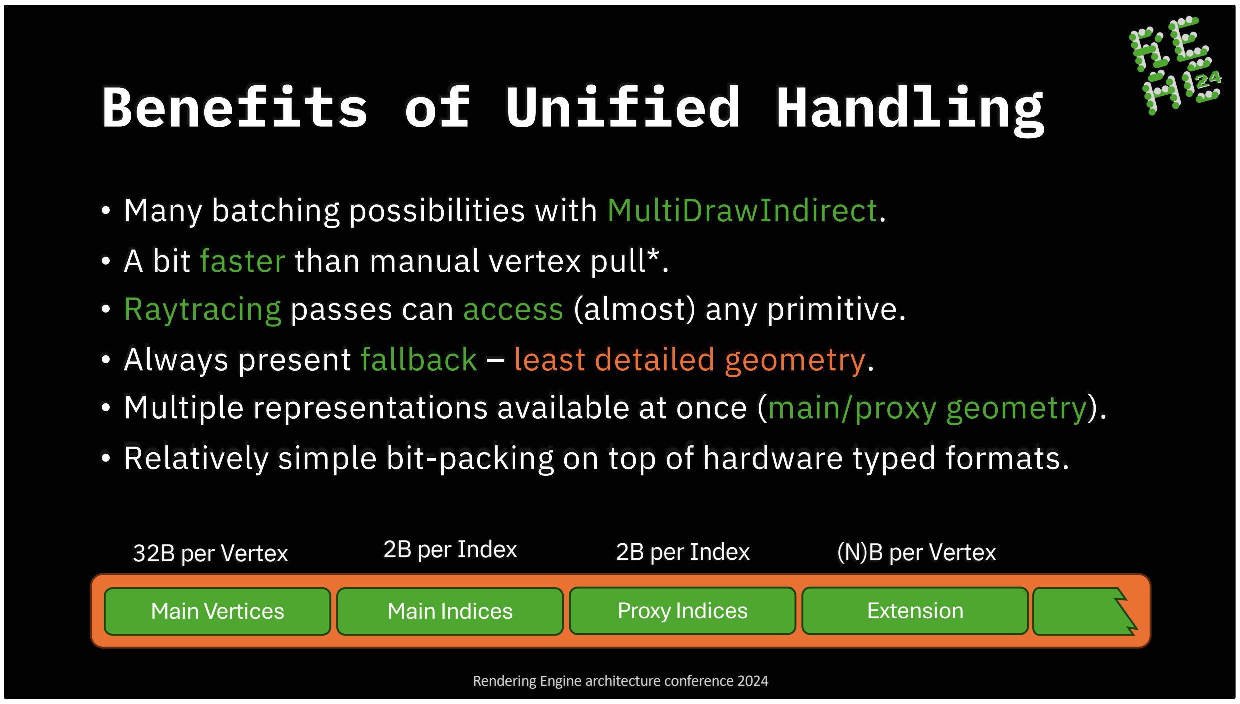
Historically, Engine always assumed small set of vertex formats with very few size classes.
Fixed set of formats sounds limiting – but it's rarely brough-up as a significant constraint for a real-life content, though.
And, when exotic format needed, it usually composed from existing format and "extension" (suppled as second stream).

[OPTIONAL]

All platforms use fixed-function geometry pulling as there's no cluster-culling or triangles culling.
In such cases, fixed function pipeline has measurable performance benefits over manual vertex pull on target hardware (measure to be about 2% faster vertex processing).

[OPTIONAL]

While we sill rely on CPU to select resident geometries and their LOD levels – it's not a preferable option.
The goal is to move most parts of scene representation and traversal to GPU in not-so-distant future.



Benefits of Unified Handling

- Many batching possibilities with MultiDrawIndirect.
- A bit faster than manual vertex pull*.
- Raytracing passes can access (almost) any primitive.
- Always present fallback – least detailed geometry.
- Multiple representations available at once (main/proxy geometry).
- Relatively simple bit-packing on top of hardware typed formats.

32B per Vertex
2B per Index
2B per Index
(N)B per Vertex
Main Vertices
Main Indices
Proxy Indices
Extension

Rendering Engine architecture conference 2024

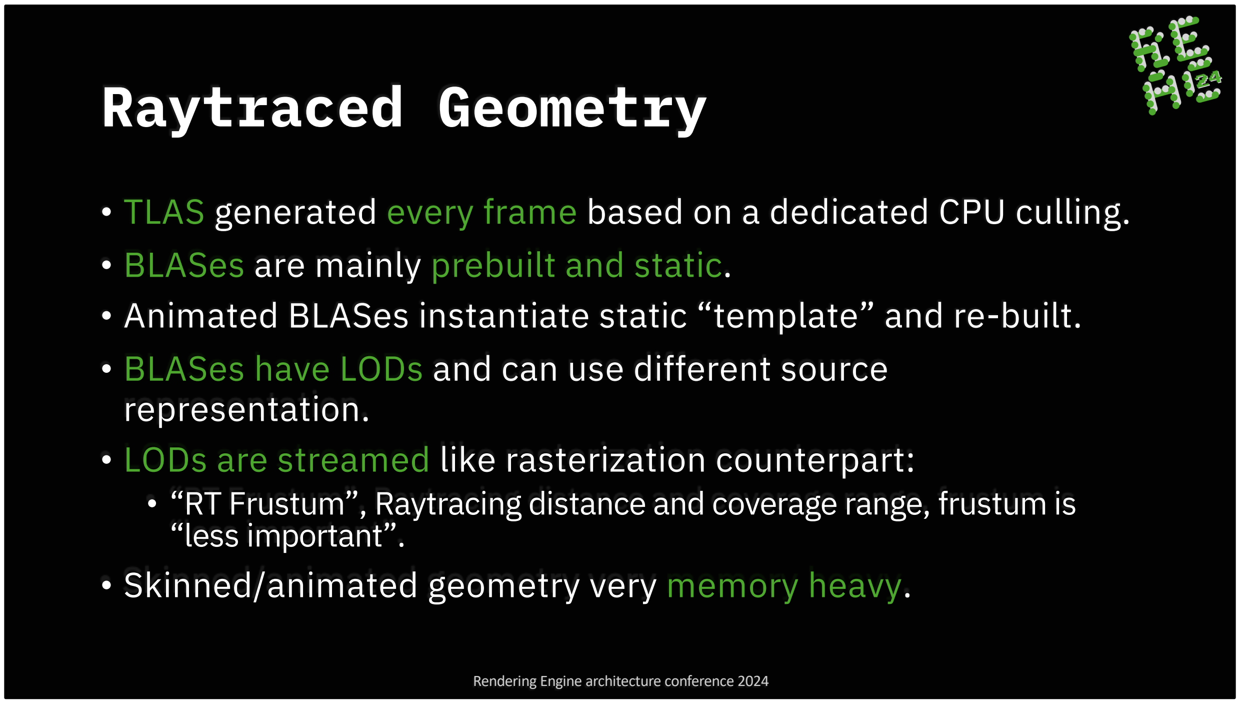
Rasterization geometry can have multiple LODs streamed in with least detailed LOD being resident and present all the time.

Each LOD contains two set of indices – main and proxy.
Main representation is there for primary view rasterization.
While proxy representation employed by shadows rasterization and/or BLAS building.

- Proxy is a simplified version of the main representation that references same vertices.

Also, as I just mentioned, some vertex formats can have custom extensions to with uncommon strides.

Global geometry buffer is there for direct rasterization and vertices attributes pulling when ray-tracing.



Raytraced Geometry

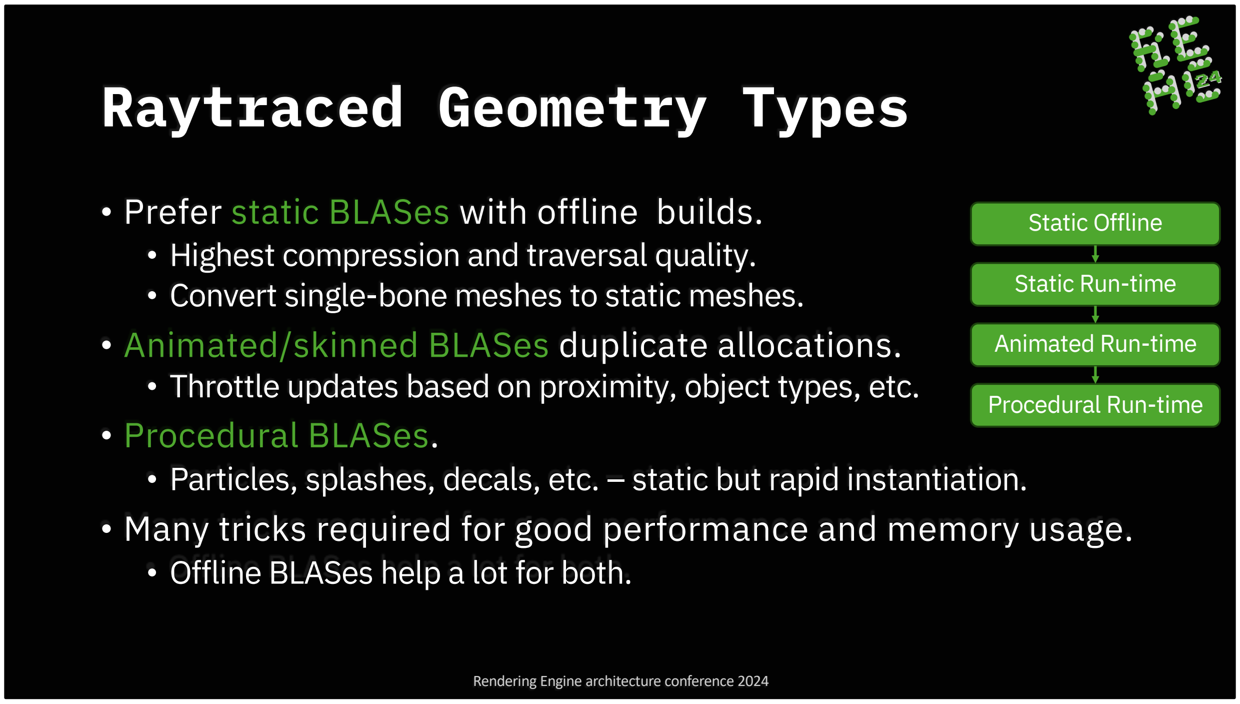
- TLAS generated every frame based on a dedicated CPU culling.
- BLASes are mainly prebuilt and static.
- Animated BLASes instantiate static “template” and re-built.
- BLASes have LODs and can use different source representation.
- LODs are streamed like rasterization counterpart:
  - “RT Frustum”, Raytracing distance and coverage range, frustum is “less important”.
- Skinned/animated geometry very memory heavy.

Rendering Engine architecture conference 2024

For ray-traced geometry – we rebuild TLAS from scratch every frame as it’s highly volatile data and with instances being selected on CPU.

Ray-tracing representation of geometries (aka. BLASes), unlike rasterization counterparts, can vary greatly in size and their memory consumption doesn’t depend linearly on vertices count.
That makes fragmentation one of the biggest concerns when managing memory for them.

To address them and to achieve faster tracing – most BLASes are prebuilt offline and streamed in/out alongside rasterization geometry.
Although, streaming prioritization is a bit different as it takes into account ray-tracing distance, etc.


Raytraced Geometry Types

- Prefer static BLASes with offline builds.
  - Highest compression and traversal quality.
  - Convert single-bone meshes to static meshes.
- Animated/skinned BLASes duplicate allocations.
  - Throttle updates based on proximity, object types, etc.
- Procedural BLASes.
  - Particles, splashes, decals, etc. – static but rapid instantiation.
- Many tricks required for good performance and memory usage.
  - Offline BLASes help a lot for both.

$$\begin{tikzcd} \text{Static Offline} \arrow[d] \\ \text{Static Run-time} \arrow[d] \\ \text{Animated Run-time} \arrow[d] \\ \text{Procedural Run-time} \end{tikzcd}$$

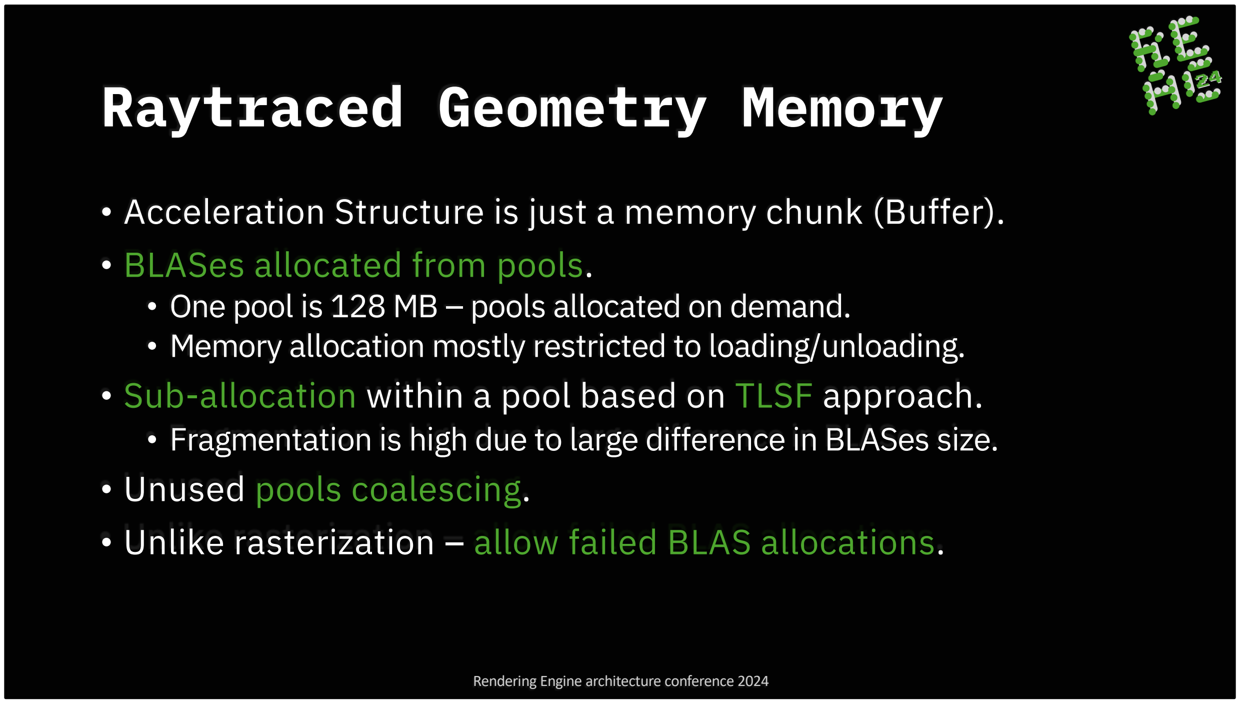
Rendering Engine architecture conference 2024

The Engine splits ray-tracing geometries in four different types:
- Static offline geometry.
- Static run-time geometry.
- Animated run-time geometry.
- And procedural run-time geometry.

These types represent different ways of handling and supported features.
They're mentioned in order of their complexity and performance/memory cost.

It's always preferred to use simpler types whenever possible with automatic conversion to simpler types.
For instance, skinned geometry with a single bone can be converted to static model in almost every case.
Doing so – enabled offline BVH building for them which comes with mentioned benefits.

Unfortunately, offline BVH building now available on PC and, thus, static offline type is not used.



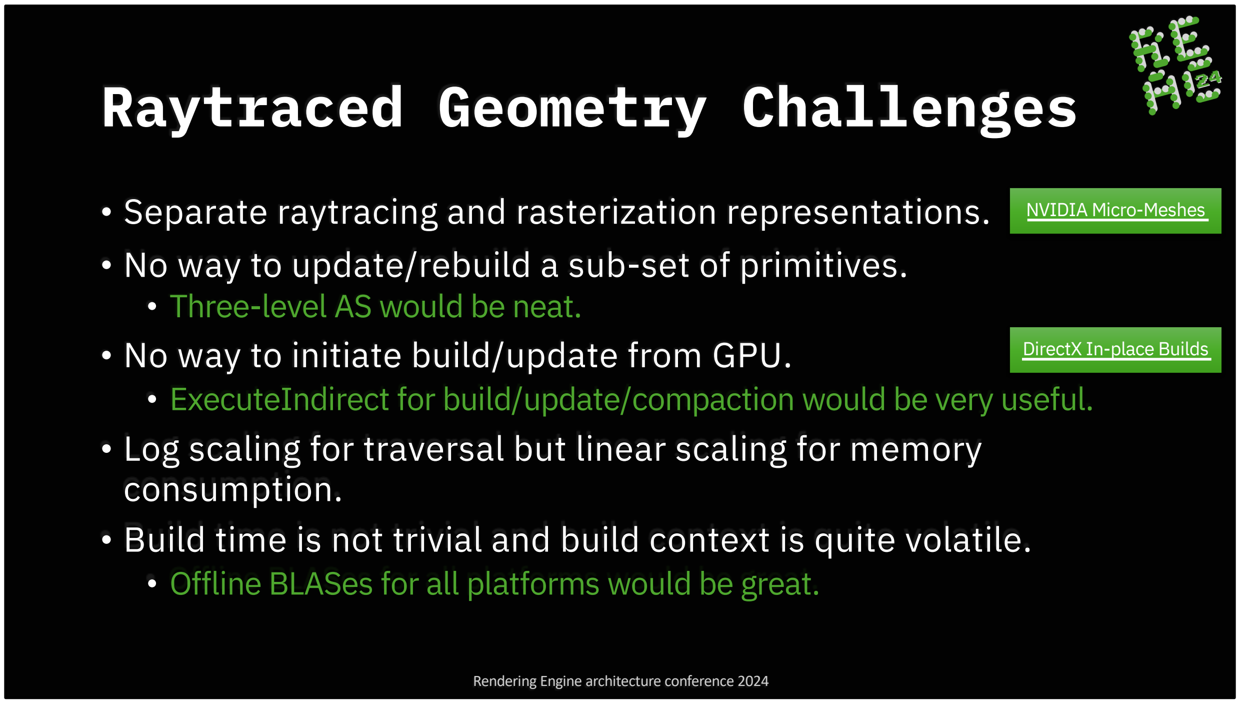
Raytraced Geometry Memory

- Acceleration Structure is just a memory chunk (Buffer).
- BLASes allocated from pools.
  - One pool is 128 MB – pools allocated on demand.
  - Memory allocation mostly restricted to loading/unloading.
- Sub-allocation within a pool based on TLSF approach.
  - Fragmentation is high due to large difference in BLASes size.
- Unused pools coalescing.
- Unlike rasterization – allow failed BLAS allocations.

Rendering Engine architecture conference 2024

Memory management for ray-traced geometry is very different when compared to rasterization geometry because of significant fragmentation and much higher memory consumption.
All BLASes allocated from pools where each pool is a 128 MB buffer. BLASes suballocated from pools using simplified and streamlined version of two-level segregated storage allocator.

Pools are created on-demand and have loose defragmentation running on them. There's an upper limit on pools to prevent unrestricted growth. When limit is reached – allocation can fail and calling code handles denial of allocation.
In case of failed allocation, it's possible to fallback to least detailed BLAS - which is always present similar to least detailed LOD in rasterization geometry.



Raytraced Geometry Challenges

- Separate raytracing and rasterization representations.
- No way to update/rebuild a sub-set of primitives.
  - Three-level AS would be neat.
- No way to initiate build/update from GPU.
  - ExecuteIndirect for build/update/compaction would be very useful.
- Log scaling for traversal but linear scaling for memory consumption.
- Build time is not trivial and build context is quite volatile.
  - Offline BLASes for all platforms would be great.

NVIDIA Micro-Meshes

DirectX In-place Builds

Rendering Engine architecture conference 2024

While DXR-based ray-tracing scene representation gets the job done – it also imposes quite a few limitations which make ray-traced geometry handling more complex and rigid that it could have been.

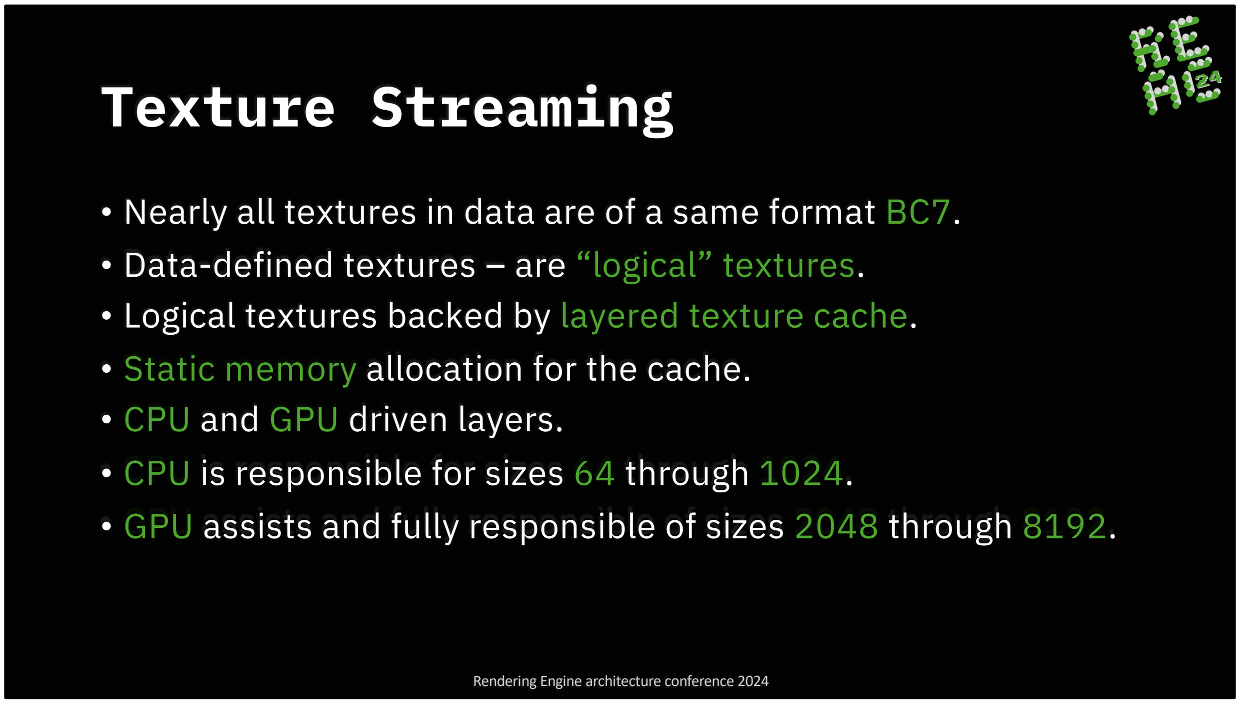
One of the biggest issues is two separate representations for rasterization and ray-tracing. It goes with high cost on memory and complexity.
Nvidia has a promising solution that allows to have unified representation. It's called Micro Meshes. It'd be great to have wider availability of this solution or alternatives.

Another big obstacle is a lack of functionality to update TLAS/BLAS on sub-level which would greatly simplify animated geometry handling and dynamic level of details.
Personally, I'd like to see something like three-level acceleration structure – which should be easier to do since it's primarily in software stack.

Having a possibility to feed BLAS builds with in-place produced data can make vertex processing handling so much easier with no need to stream out processed vertices to memory.

There's a research within DXR team at Microsoft to implement similar solution which is not part of the specification yet. It would be really great to have it (or something similar) in not-so-distant future.

Finally, it would be beneficial to have ways of offline BLASes building on PC – as it offers significant performance and memory savings.
We've seen up to 30-40% reduction in memory consumption and about 5-15% boost in tracing speed.



Texture Streaming

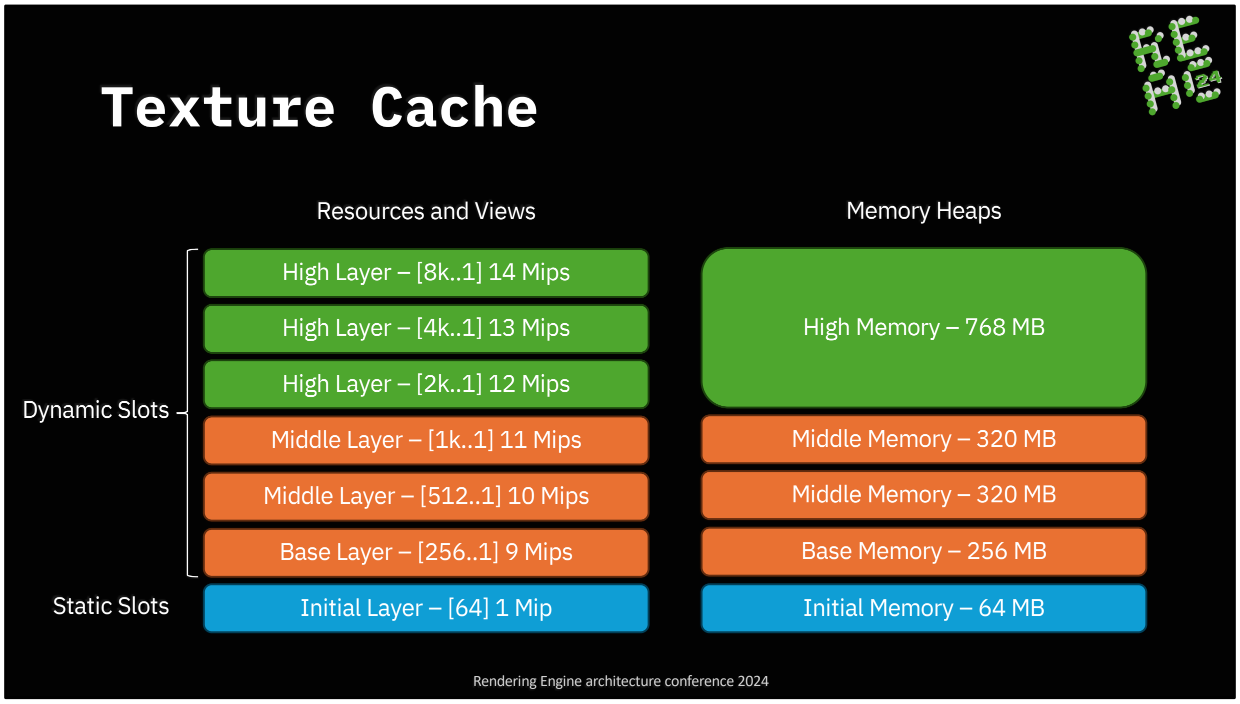
- Nearly all textures in data are of a same format BC7.
- Data-defined textures – are “logical” textures.
- Logical textures backed by layered texture cache.
- Static memory allocation for the cache.
- CPU and GPU driven layers.
- CPU is responsible for sizes 64 through 1024.
- GPU assists and fully responsible of sizes 2048 through 8192.

Rendering Engine architecture conference 2024

For texture streaming we exploit the fact of almost all textures being stored as power of two BC7 compressed data.
(While it’s not strictly necessary for the solution discussed – it makes it a bit simpler.)

Using this assumption, we pre-allocate a set of resources with different number of MIP levels and then swap them around when texture changes quality level.

This way, logical concept of a “texture” and actual resources with texture data are kind of separated and linked to one-another when certain quality level streamed in.
Low-level rendering resources and views statically allocated in texture cache and then borrowed by logical “textures”.



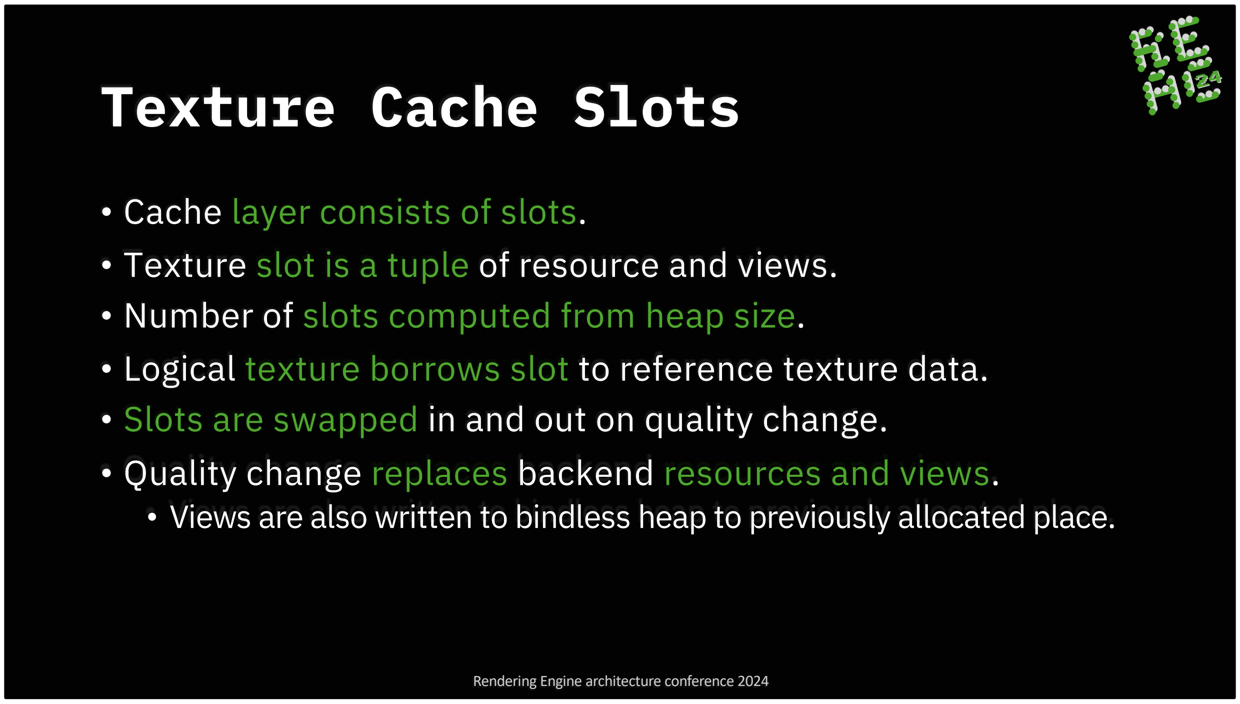
Texture Cache

Resources and Views
Memory Heaps
Dynamic Slots
Static Slots
High Layer – [8k..1] 14 Mips
High Layer – [4k..1] 13 Mips
High Layer – [2k..1] 12 Mips
Middle Layer – [1k..1] 11 Mips
Middle Layer – [512..1] 10 Mips
Base Layer – [256..1] 9 Mips
Initial Layer – [64] 1 Mip
High Memory – 768 MB
Middle Memory – 320 MB
Middle Memory – 320 MB
Base Memory – 256 MB
Initial Memory – 64 MB

Rendering Engine architecture conference 2024

Texture cache consists of seven cache layers:

- Initial layer is static and contains all the textures available in content. It's never streamed in or out and acts as a fallback.
- Layers 256 though 1k represent base and middle layer types - resources here implemented as placed textures.
- Layers 2k though 8k represent high layer type - here resources implemented as virtual textures.



Texture Cache Slots

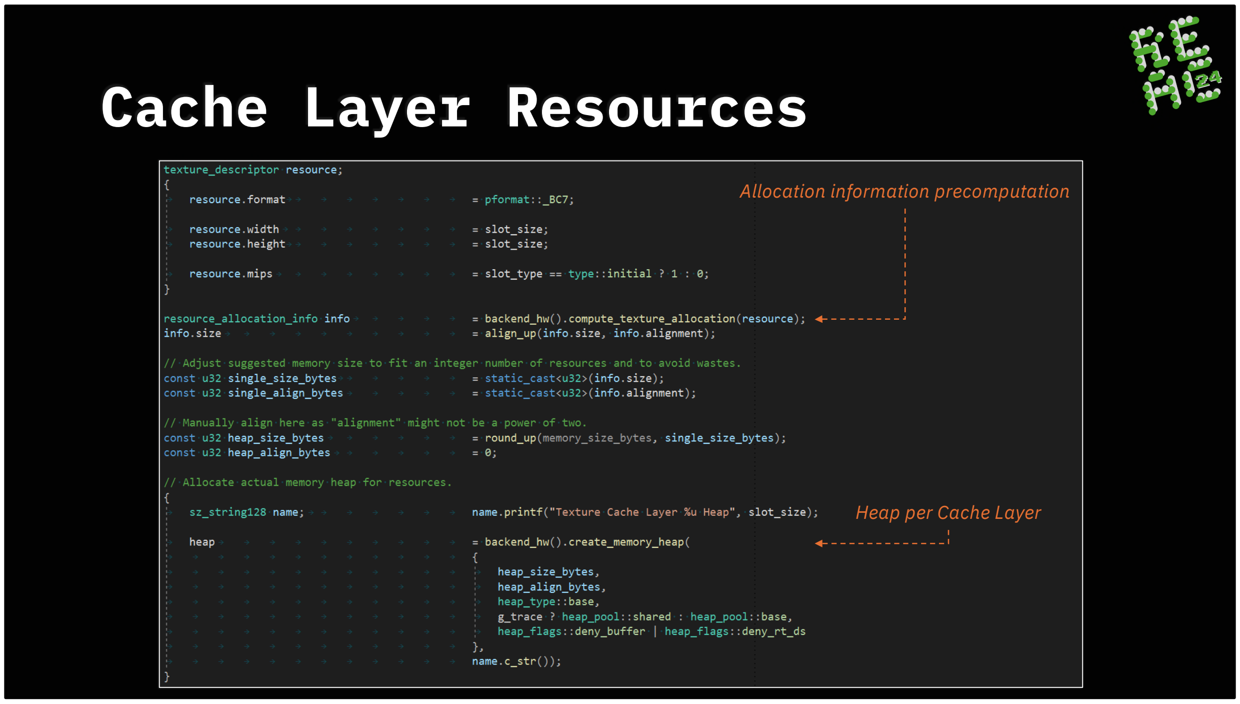
- Cache layer consists of slots.
- Texture slot is a tuple of resource and views.
- Number of slots computed from heap size.
- Logical texture borrows slot to reference texture data.
- Slots are swapped in and out on quality change.
- Quality change replaces backend resources and views.
  - Views are also written to bindless heap to previously allocated place.

Rendering Engine architecture conference 2024

Each layer of the cache is an array of slots where each slot is just a texture resource with two read views.
Each slot has linear and sRGB read views – one of them used in shaders based on implicit knowledge of expected content of a texture.

Number of slots in a layer computed based on layer memory budget and allocation size needed for a single resource.

When logical "texture" gets linked to a slot, it borrows texture and read views. So that any access to It will go directly to layer slot that is active right now.
As a part of linking process, read views are written into bindless array as well. (This ensures synchronized access to the texture views regardless of the binding model used.)



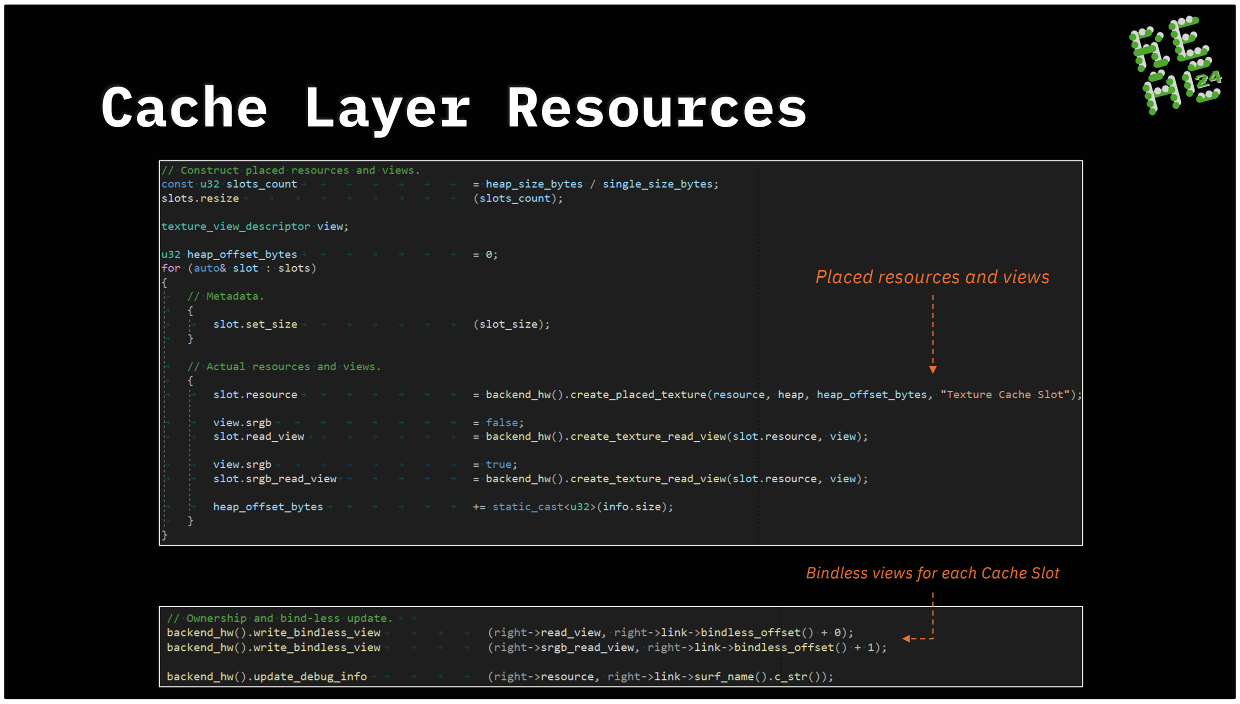
Cache Layer Resources

```
texture_descriptor resource;
{
    resource.format                     = pformat::_BC7;

    resource.width                      = slot_size;
    resource.height                     = slot_size;

    resource.mips                       = slot_type == type::initial ? 1 : 0;
}

resource_allocation_info info           = backend_hw().compute_texture_allocation(resource);
info.size                               = align_up(info.size, info.alignment);

// Adjust suggested memory size to fit an integer number of resources and to avoid wastes.
const u32 single_size_bytes             = static_cast<u32>(info.size);
const u32 single_align_bytes            = static_cast<u32>(info.alignment);

// Manually align here as "alignment" might not be a power of two.
const u32 heap_size_bytes               = round_up(memory_size_bytes, single_size_bytes);
const u32 heap_align_bytes              = 0;

// Allocate actual memory heap for resources.
{
    sz_string128 name;                  name.printf("Texture Cache Layer %u Heap", slot_size);

    heap                                = backend_hw().create_memory_heap(
                                        {
                                            heap_size_bytes,
                                            heap_align_bytes,
                                            heap_type::base,
                                            g_trace ? heap_pool::shared : heap_pool::base,
                                            heap_flags::deny_buffer | heap_flags::deny_rt_ds
                                        },
                                        name.c_str());
}
```

Here’s how cache layer construction looks like for middle layer types.

It starts with computation of a single slot size based supported MIP levels.
This information then utlilized to arrange resources placement within a heap with respect to their size and alignment.

Next, memory heap is created to host these placed resources.



Cache Layer Resources

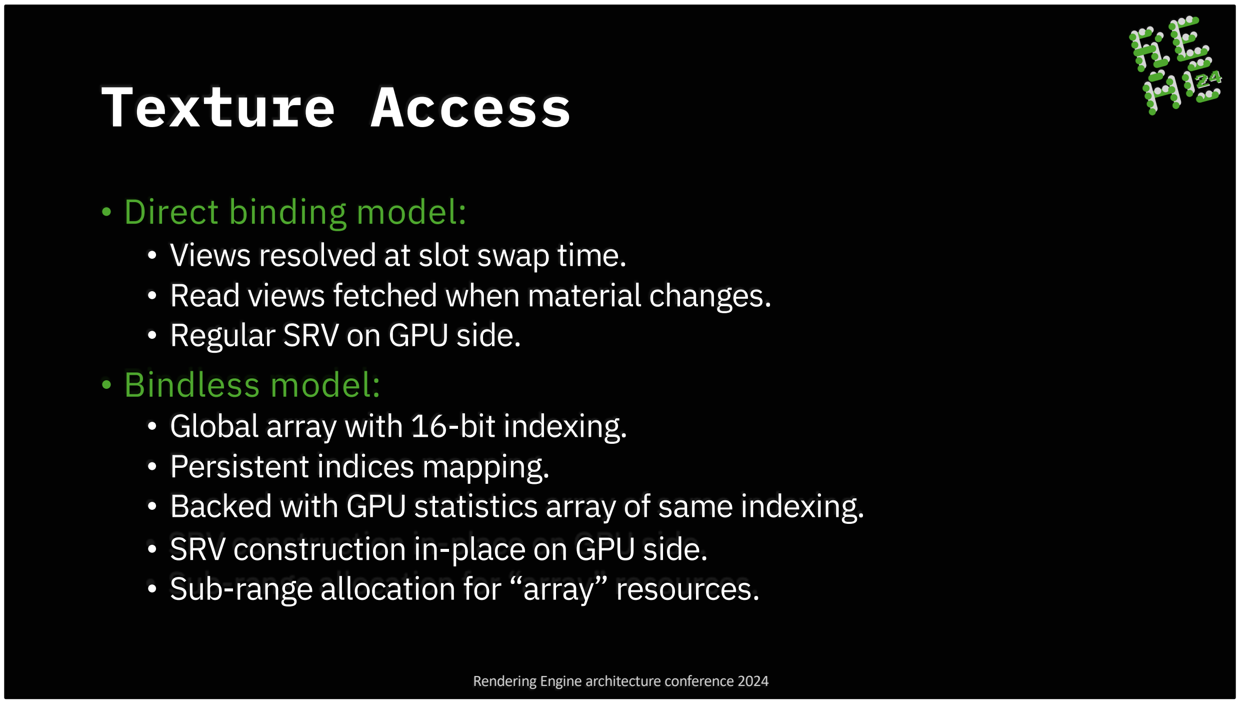
```
// Construct placed resources and views.
const u32 slots_count                      = heap_size_bytes / single_size_bytes;
slots.resize                               (slots_count);

texture_view_descriptor view;

u32 heap_offset_bytes                      = 0;
for (auto& slot : slots)
{
    // Metadata.
    {
        slot.set_size                      (slot_size);
    }

    // Actual resources and views.
    {
        slot.resource                      = backend_hw().create_placed_texture(resource, heap, heap_offset_bytes, "Texture Cache Slot");

        view.srgb                          = false;
        slot.read_view                     = backend_hw().create_texture_read_view(slot.resource, view);

        view.srgb                          = true;
        slot.srgb_read_view                = backend_hw().create_texture_read_view(slot.resource, view);

        heap_offset_bytes                  += static_cast<u32>(info.size);
    }
}
```

Placed resources and views
Bindless views for each Cache Slot

```
// Ownership and bind-less update.
backend_hw().write_bindless_view           (right->read_view, right->link->bindless_offset() + 0);
backend_hw().write_bindless_view           (right->srgb_read_view, right->link->bindless_offset() + 1);

backend_hw().update_debug_info             (right->resource, right->link->surf_name().c_str());
```

At this point it’s just a matter of populating texture slots with placed resources and respective views.

Layer slots don’t allocate bindless space as they can migrate from one logical “texture” to another while we want to maintain persistent indices on logical level. Thus, bindless allocation deferred to logical “textures”.



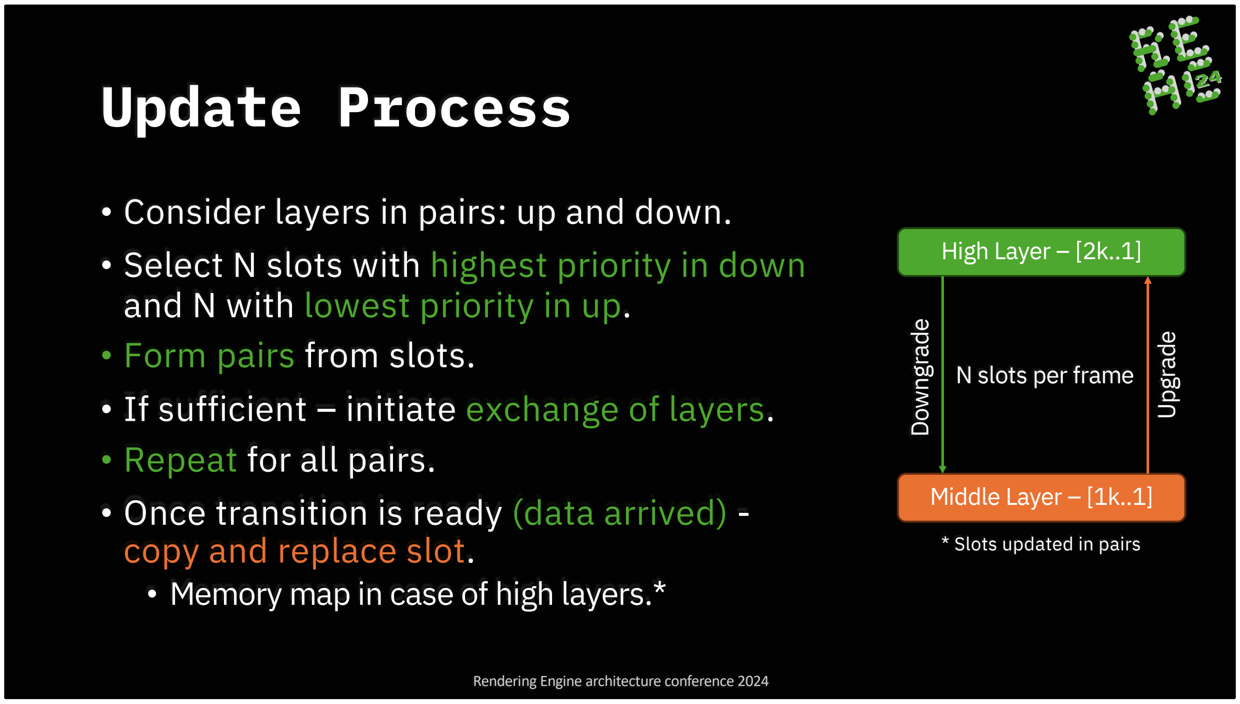
Texture Access

- Direct binding model:
  - Views resolved at slot swap time.
  - Read views fetched when material changes.
  - Regular SRV on GPU side.
- Bindless model:
  - Global array with 16-bit indexing.
  - Persistent indices mapping.
  - Backed with GPU statistics array of same indexing.
  - SRV construction in-place on GPU side.
  - Sub-range allocation for “array” resources.

Rendering Engine architecture conference 2024

In case of direct binding model (which is still utilized by many systems), texture_read_view passed in low-level draw call/dispatch issuing code.
From there it is transformed directly to platform specific view representation (which is no-op, just a cast) and scheduled for reading.

In case of bindless model all IDs are allocated from a single array.
The bindless ID is just a simple 16-bit index sufficient to fetch a and to address access statistics for each logical “texture”.
These statistics contain accessed MIP index, number of samples made, etc. and populated by a hardware VM unit.
Statistics drive GPU-assisted part of streaming prioritization.



Update Process

- Consider layers in pairs: up and down.
- Select N slots with highest priority in down and N with lowest priority in up.
- Form pairs from slots.
- If sufficient – initiate exchange of layers.
- Repeat for all pairs.
- Once transition is ready (data arrived) - copy and replace slot.
  - Memory map in case of high layers.*

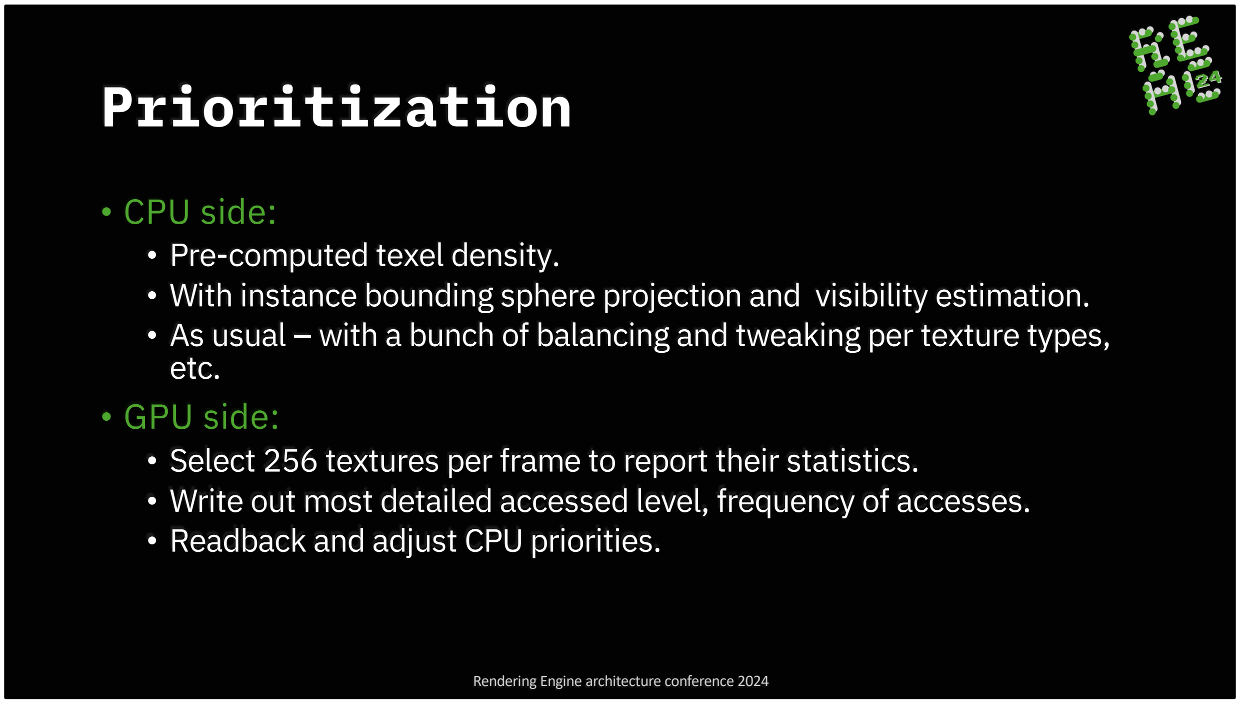
High Layer – [2k..1]
Downgrade
N slots per frame
Upgrade
Middle Layer – [1k..1]
* Slots updated in pairs

Rendering Engine architecture conference 2024

Streamer relies on CPU and GPU priorities to decide quality level for each active texture.
This process can be described as a downgrade/upgrade algorithm executed on pairs of layers adjacent to each other:

1. First, we select several slots with highest priority in smaller layer and same number of slots with lowest priority in higher layer.
2. Next, if priority delta is high enough – we initiate loading of a MIP level matching higher layer size.
   1. While waiting for data – slots are set in pending state and can't be elected for transition until in this state.
   2. And only a certain number of pending slots per layer allowed to control IO throughput.
3. Next, when the new MIP data is ready – we exchange existing MIP levels between layers and upload new MIP level to higher layer.
   1. In case of higher layers – instead of copying data around, we map entire MIP level to virtual address space of the upgraded slot.

And then - steps two and three repeated for all selected pairs of slots.


Prioritization

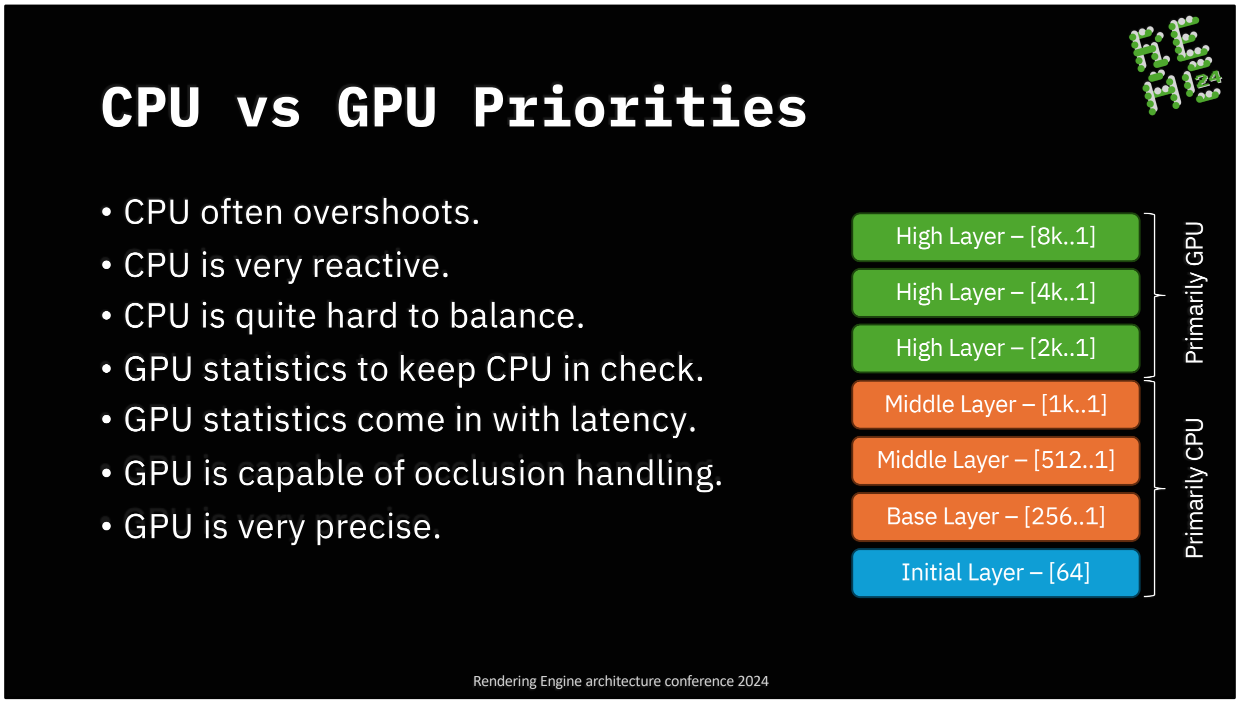
- CPU side:
  - Pre-computed texel density.
  - With instance bounding sphere projection and visibility estimation.
  - As usual – with a bunch of balancing and tweaking per texture types, etc.
- GPU side:
  - Select 256 textures per frame to report their statistics.
  - Write out most detailed accessed level, frequency of accesses.
  - Readback and adjust CPU priorities.

Rendering Engine architecture conference 2024

Described process has a very good property of self balancing under heavy overcommitment.
To get the best of this property it's crucial to have priority estimation as precise as possible.

To achieve that, we use different prioritization strategies for different cache layers.
Cache layers rely on a mixture of CPU and GPU based estimations.

CPU estimation based on pre-computed texel density and bounding sphere projection for each instance using a texture.
GPU estimation, on the other hand, is a follow-up/assisting estimation based on direct memory access statistics.



CPU vs GPU Priorities

- CPU often overshoots.
- CPU is very reactive.
- CPU is quite hard to balance.
- GPU statistics to keep CPU in check.
- GPU statistics come in with latency.
- GPU is capable of occlusion handling.
- GPU is very precise.

High Layer – [8k..1]
High Layer – [4k..1]
High Layer – [2k..1]
Middle Layer – [1k..1]
Middle Layer – [512..1]
Base Layer – [256..1]
Initial Layer – [64]
Primarily GPU
Primarily CPU

Rendering Engine architecture conference 2024

Both sources of priority have their own advantages and flaws which fit great some layers and harm other.

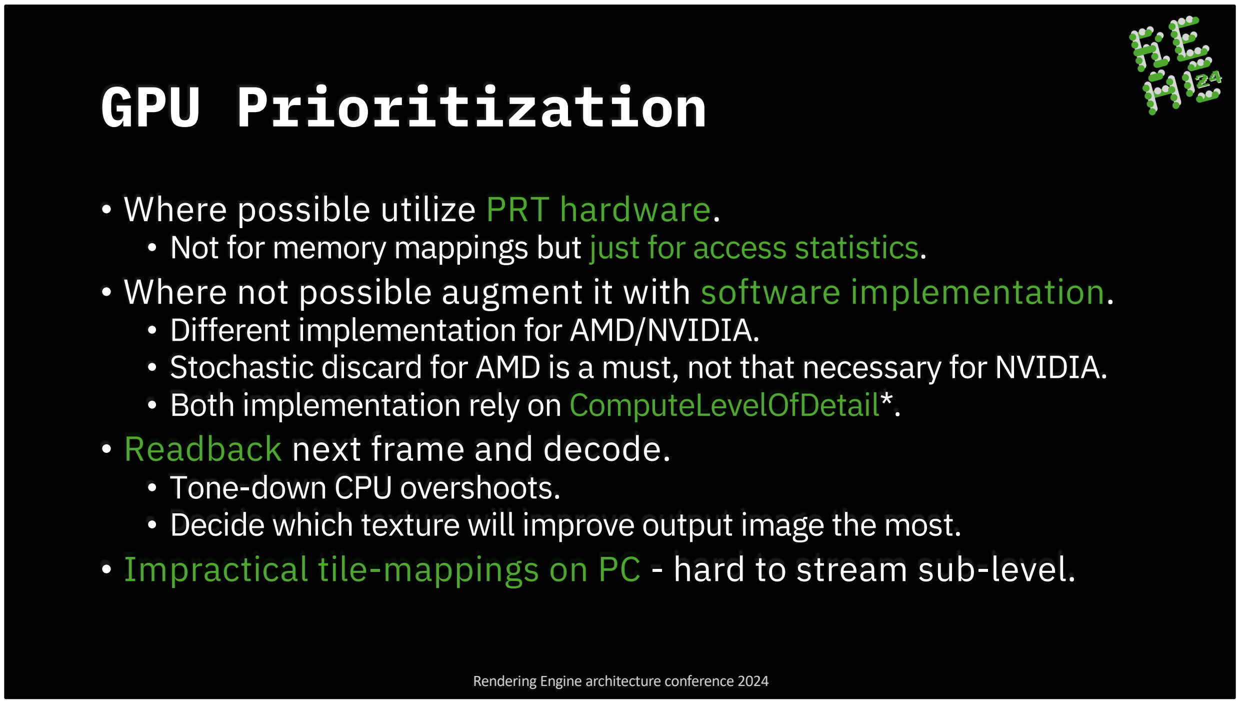
For CPU estimation:
- It’s very conservative and often overshoots.
- But it’s reactive and makes texture requests before they’re seen on screen – which, effectively, shortens latency.

(While it’s fine for layers with smaller memory footprint it quickly becomes a bottleneck for high layers where cost of memory movement is significant.)

For GPU estimation:
- It’s exact – we know a specific level of detail that must be present to satisfy all requested reads on GPU.
- But it comes with significant latency.

(While it can be fine for higher layers it is very noticeable when happens at middle layers (as quality difference between these layers is perceptually higher).

To get best of both estimates, we combine them, making CPU in charge of initial, base, and middle layers and leaving high layers primarily for GPU.



GPU Prioritization

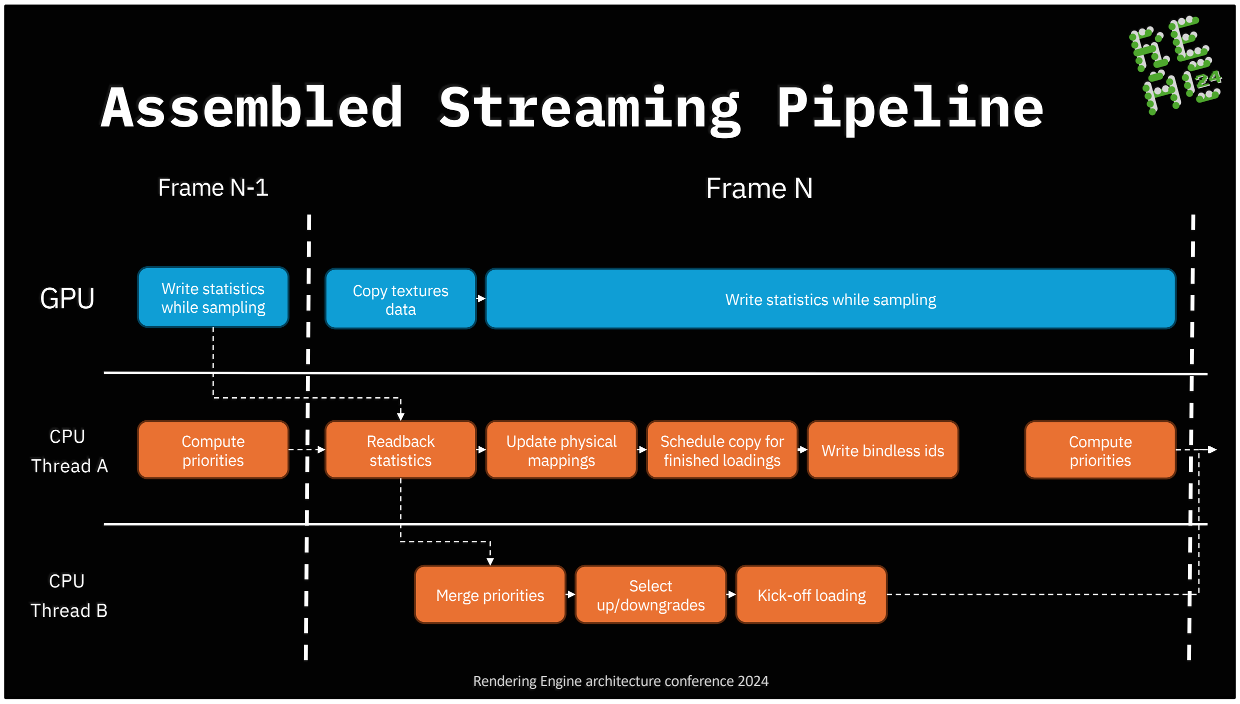
- Where possible utilize PRT hardware.
  - Not for memory mappings but just for access statistics.
- Where not possible augment it with software implementation.
  - Different implementation for AMD/NVIDIA.
  - Stochastic discard for AMD is a must, not that necessary for NVIDIA.
  - Both implementation rely on ComputeLevelOfDetail*.
- Readback next frame and decode.
  - Tone-down CPU overshoots.
  - Decide which texture will improve output image the most.
- Impractical tile-mappings on PC - hard to stream sub-level.

Rendering Engine architecture conference 2024

For GPU estimation Engine selects 256 textures each frame to report their usage statistics.
On platforms with access to PRT/VM hardware this hardware utilized to write them out.
Where it's not available – it's substituted by software implementation:

- In its' core it's a `ComputeLevelOfDetail` call with some post processing.
- Details are different for AMD and NVIDIA as they tend to tolerate many atomic writes to same memory location differently.

We wanted to use GPU estimation to control streaming decisions on sub-MIP level (something like a Sampler Feedback).
But, while it's viable option on consoles, it's still impractically slow to map tiles on PC when they are not a consecutive range of tiles, have gaps, etc.
(It'd be really great to have it addressed finally by Microsoft on OS level as it will unlock so many possibilities to unify code and achieve more optimal memory consumptions.)


Assembled Streaming Pipeline

This figure represents an entire texture streaming pipeline assembled from the components I just described.

It has three parallel execution timelines synchronized on frame boundary.
At the beginning of a frame, sampling statistics from GPU copied to CPU for analysis.

CPU and GPU estimate of priorities kicked off for processing in separate thread:

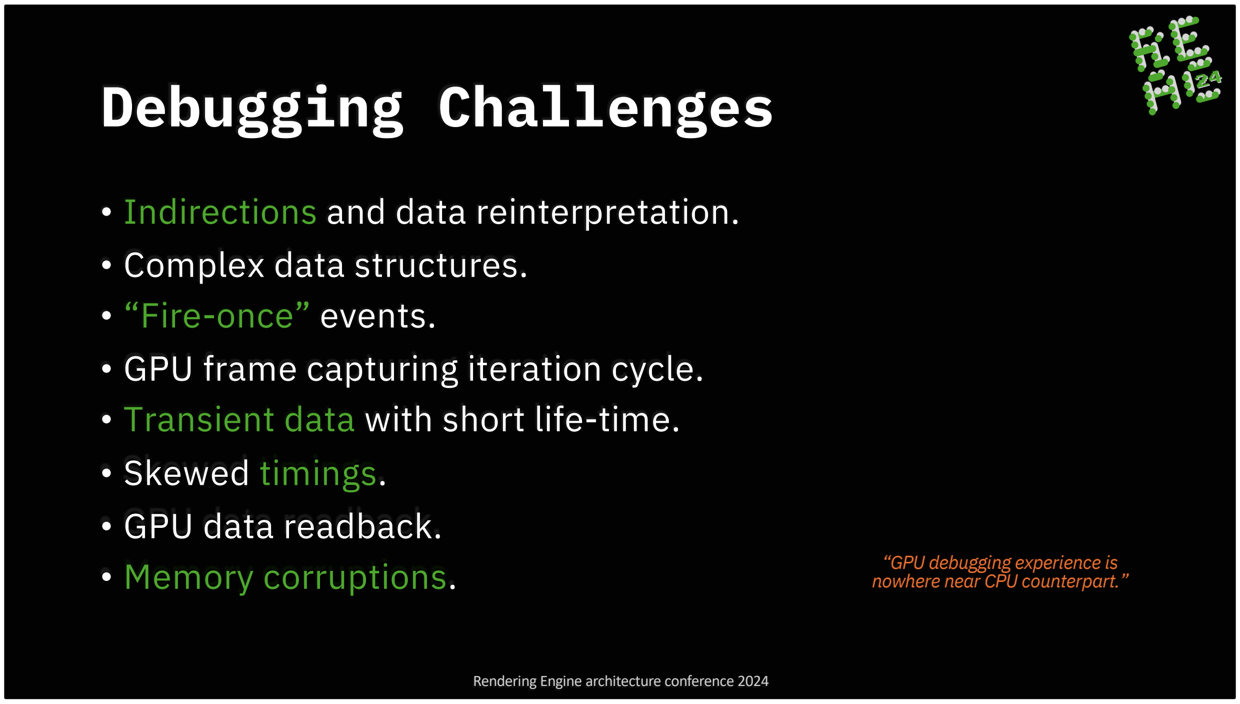
- There, they are merged to produce single priority value.
- Then, they are fed into update process (which we described few slides ago).
- Based on decision made in update process - data loading for top MIPs can be initiated.

The original CPU thread, in parallel, processes completed loading requests from previous frames, updates tile mappings, writes bindless views, and schedules MIP copy operations for GPU.

Meanwhile, GPU copies texture data between cache layers and starts scene rendering which produces new set of access statistics for the next frame.

Tools

Now, moving to the next section of the talk which covers custom GPU debugging, inspection, and supporting tools developed while working on resources model and streaming solutions.
They were instrumental to achieve correctness and performance goals quickly.


Debugging Challenges

- Indirections and data reinterpretation.
- Complex data structures.
- "Fire-once" events.
- GPU frame capturing iteration cycle.
- Transient data with short life-time.
- Skewed timings.
- GPU data readback.
- Memory corruptions.

"GPU debugging experience is nowhere near CPU counterpart."

Rendering Engine architecture conference 2024

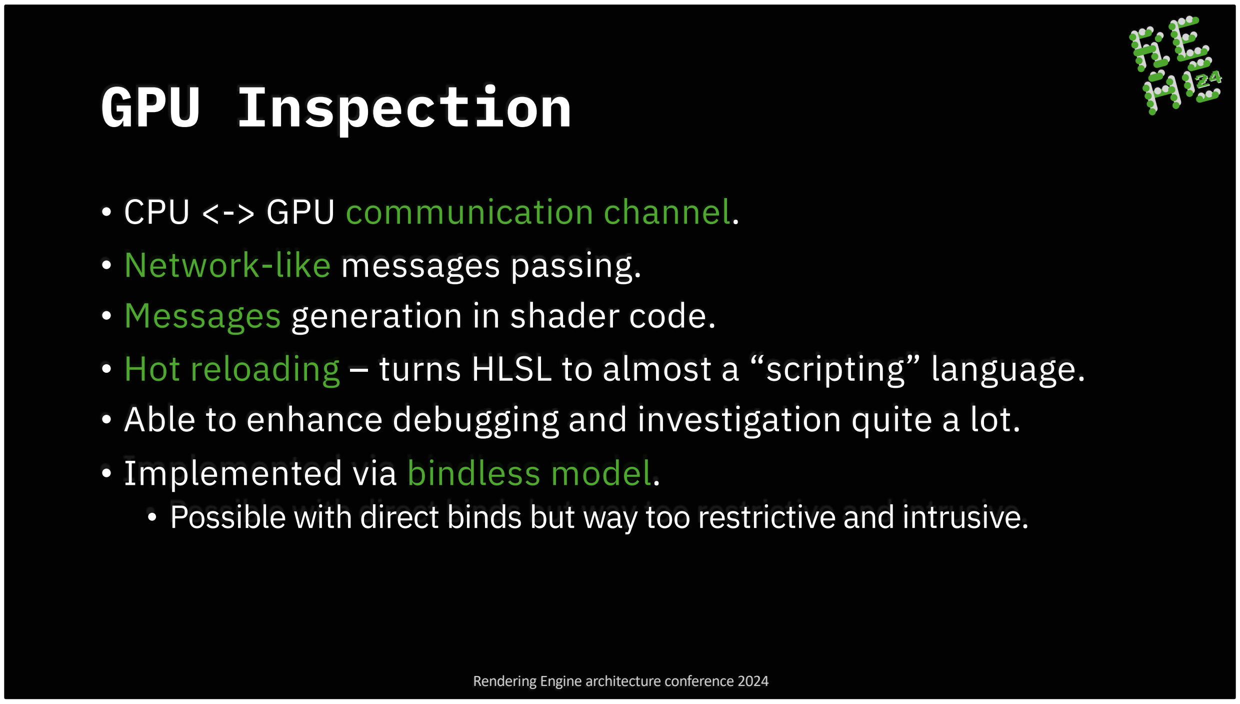
Generally speaking - GPU-side debugging experience was and still is challenging at times. But with increasing complexity of GPU data structures and wider adoption of flexible data models the struggling is even high.

It's widely known that GPU debugging is lacking behind that of a CPU - where you can pause execution at any moment of time, inspect entire memory, and so on. On consoles disparity is lower with unified memory and the ability to halt GPU and attached a debugger.

But even there it's still a challenge to "catch" a state of a system "then and there". In most cases it boils down to a GPU capture triggering – which is relatively time consuming and disruptive process.
Furthermore, to inspect data over time or just see it in real-time – we tend to modify shader code to output "debug values" to textures and then blit them to screen.

While working on bindless resources model and GPU-assisted texture streaming, the iteration time of debugging and testing was high enough for us to consider building simpler custom debugging and inspection tools to expedite the

processes.
That resulted in two sets of tools called GPU Inspection and GPU Extraction.

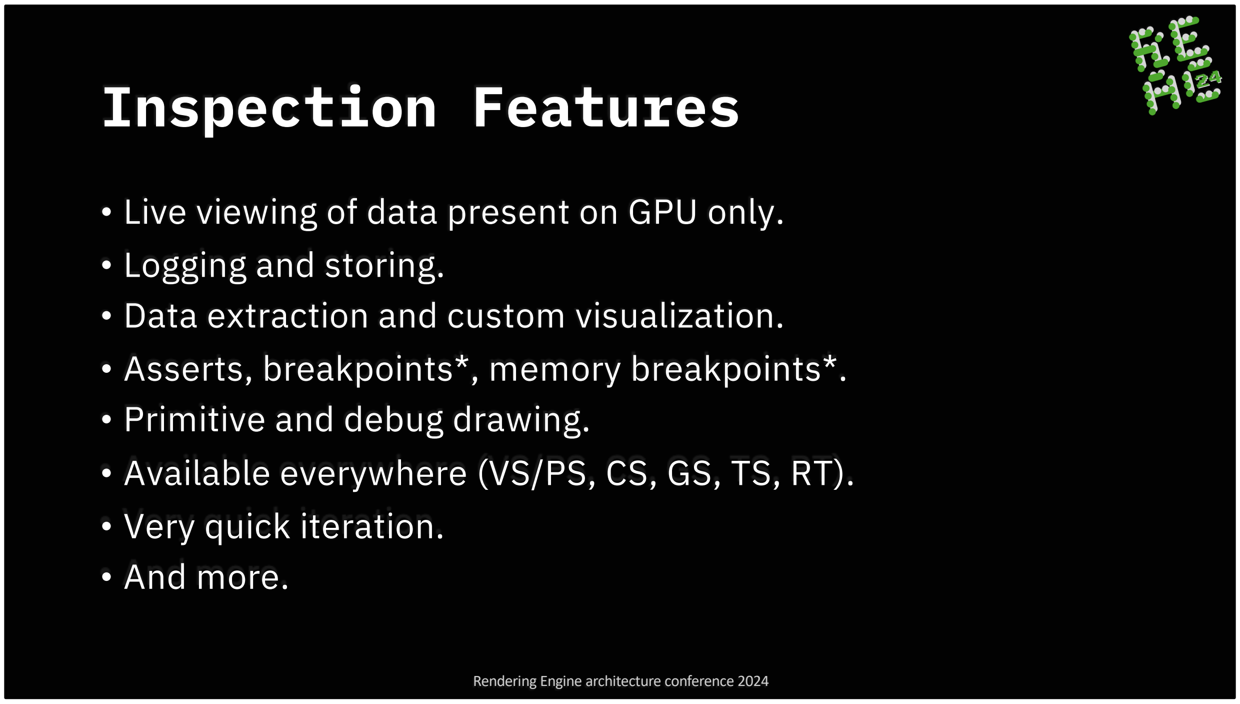


GPU Inspection

- CPU <-> GPU communication channel.
- Network-like messages passing.
- Messages generation in shader code.
- Hot reloading – turns HLSL to almost a “scripting” language.
- Able to enhance debugging and investigation quite a lot.
- Implemented via bindless model.
  - Possible with direct binds but way too restrictive and intrusive.

Rendering Engine architecture conference 2024

The first one is GPU Inspection. It’s a very simple communication channel between CPU and GPU that allows network-like messages passing.
GPU generate message can invoke execution of CPU code based on decision made in shaders.
Which is a powerful tools as it supports _any_ code.
Using shader code to control messages generation allows for very quick iterations as shaders have support for hot reloading with run-time re-compilation.



Inspection Features

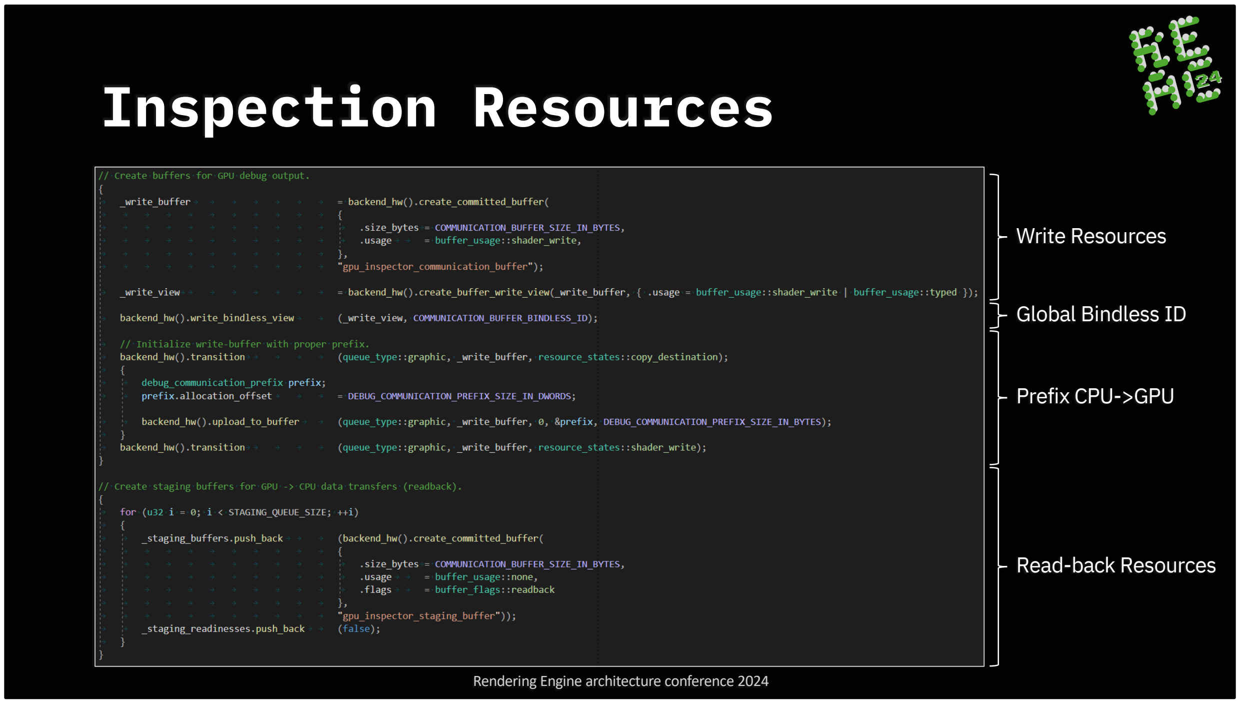
- Live viewing of data present on GPU only.
- Logging and storing.
- Data extraction and custom visualization.
- Asserts, breakpoints*, memory breakpoints*.
- Primitive and debug drawing.
- Available everywhere (VS/PS, CS, GS, TS, RT).
- Very quick iteration.
- And more.

Rendering Engine architecture conference 2024

GPU-generated messages are composed with one another to implement next inspection and debugging features:

- Run-time (or live) data inspection, logging, and capturing.
- Breakpoints and asserts based on condition defined in shader code.
- Primitive and debug drawing for any information available on GPU.
- And much more.

Since GPU Inspection implemented on top of bindless resources model (communication channel is a bindless buffer resource) – it’s available in any shader stage and almost any queue type.
(One can live-inspect data in Tessellation shader or Ray-tracing shader, etc.)



Inspection Resources

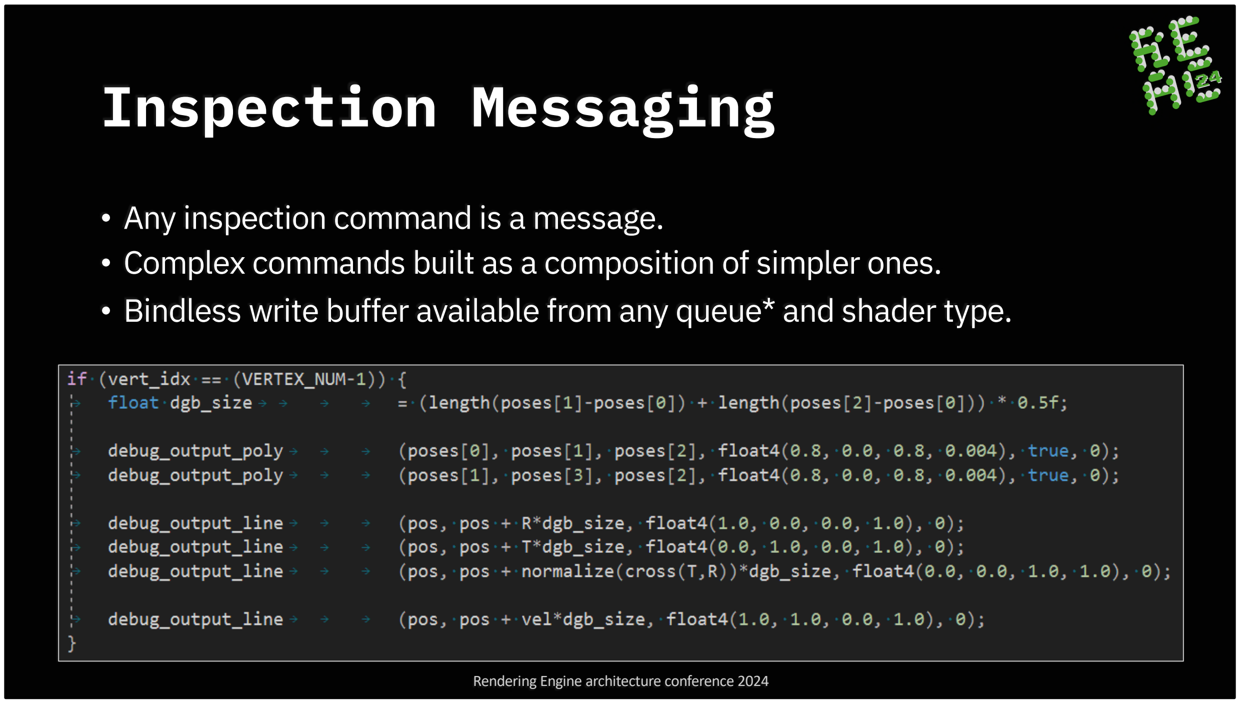
```
// Create buffers for GPU debug output.
{
    _write_buffer                               = backend_hw().create_committed_buffer(
                                                {
                                                    .size_bytes = COMMUNICATION_BUFFER_SIZE_IN_BYTES,
                                                    .usage      = buffer_usage::shader_write,
                                                },
                                                "gpu_inspector_communication_buffer");

    _write_view                                 = backend_hw().create_buffer_write_view(_write_buffer, { .usage = buffer_usage::shader_write | buffer_usage::typed });

    backend_hw().write_bindless_view              (_write_view, COMMUNICATION_BUFFER_BINDLESS_ID);

    // Initialize write-buffer with proper prefix.
    backend_hw().transition                       (queue_type::graphic, _write_buffer, resource_states::copy_destination);
    {
        debug_communication_prefix prefix;
        prefix.allocation_offset                = DEBUG_COMMUNICATION_PREFIX_SIZE_IN_DWORDS;

        backend_hw().upload_to_buffer             (queue_type::graphic, _write_buffer, 0, &prefix, DEBUG_COMMUNICATION_PREFIX_SIZE_IN_BYTES);
    }
    backend_hw().transition                       (queue_type::graphic, _write_buffer, resource_states::shader_write);
}

// Create staging buffers for GPU -> CPU data transfers (readback).
{
    for (u32 i = 0; i < STAGING_QUEUE_SIZE; ++i)
    {
        _staging_buffers.push_back                (backend_hw().create_committed_buffer(
                                                {
                                                    .size_bytes = COMMUNICATION_BUFFER_SIZE_IN_BYTES,
                                                    .usage      = buffer_usage::none,
                                                    .flags      = buffer_flags::readback
                                                },
                                                "gpu_inspector_staging_buffer"));
        _staging_readinesses.push_back            (false);
    }
}
```

This code shows how GPU Inspection uses low-level resources and bindless model to establish a communication channel.

First, the buffer and view created to provide a message queue functionality on GPU.
The view is written to predefined slot in bindless area so that any shader knows where to find it with no need to bind communication channel to every shader using it.

Next, the prefix data written at the beginning of the buffer. It's a messages allocation context.

Finally, a collection of read-back buffers created to download GPU produced messages to CPU.
Some message types can be processed on GPU directly, though.
They are just marked as processed and skipped over on CPU.



Inspection Messaging

- Any inspection command is a message.
- Complex commands built as a composition of simpler ones.
- Bindless write buffer available from any queue* and shader type.

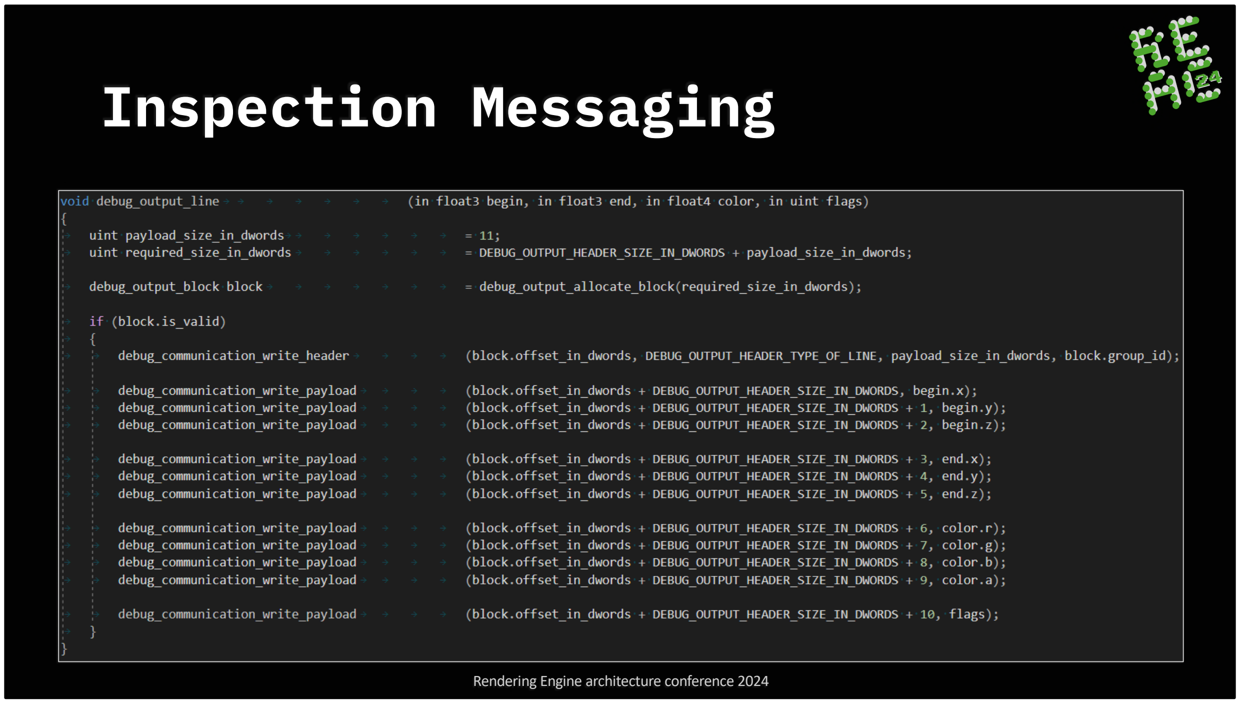
```
if (vert_idx == (VERTEX_NUM-1)) {
    float dgb_size              = (length(poses[1]-poses[0]) + length(poses[2]-poses[0])) * 0.5f;

    debug_output_poly           (poses[0], poses[1], poses[2], float4(0.8, 0.0, 0.8, 0.004), true, 0);
    debug_output_poly           (poses[1], poses[3], poses[2], float4(0.8, 0.0, 0.8, 0.004), true, 0);

    debug_output_line           (pos, pos + R*dgb_size, float4(1.0, 0.0, 0.0, 1.0), 0);
    debug_output_line           (pos, pos + T*dgb_size, float4(0.0, 1.0, 0.0, 1.0), 0);
    debug_output_line           (pos, pos + normalize(cross(T,R))*dgb_size, float4(0.0, 0.0, 1.0, 1.0), 0);

    debug_output_line           (pos, pos + vel*dgb_size, float4(1.0, 1.0, 0.0, 1.0), 0);
}
```

Rendering Engine architecture conference 2024

On GPU side, a list of predefined and commonly used commands provided alongside a possibility to build completely new commands and messages.

I'll use this simple example to walk through the messages generation process and how they are serialized to communication channel.

Here, we record command from Vertex Shader code to draw some primitives and lines.
The debug_output_line function generates a message to draw a simple line from point A to point B with specific color.



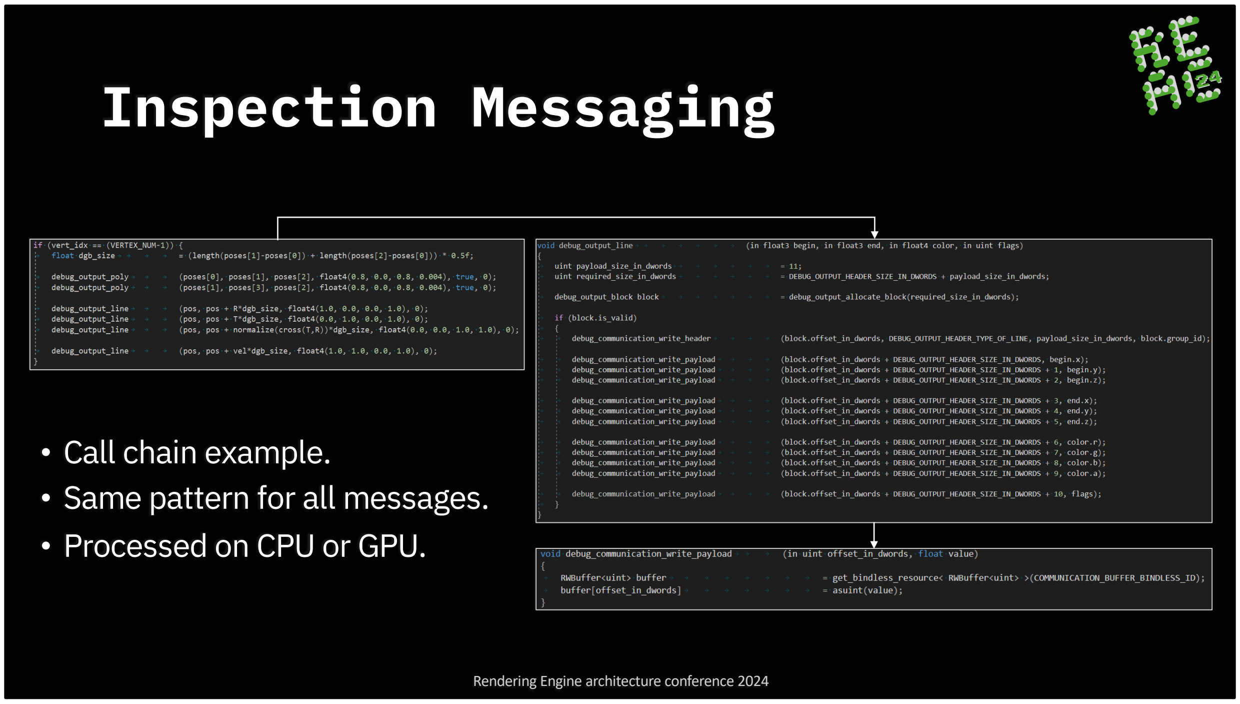
Inspection Messaging

```
void debug_output_line                         (in float3 begin, in float3 end, in float4 color, in uint flags)
{
    uint payload_size_in_dwords                    = 11;
    uint required_size_in_dwords                   = DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + payload_size_in_dwords;

    debug_output_block block                       = debug_output_allocate_block(required_size_in_dwords);

    if (block.is_valid)
    {
        debug_communication_write_header           (block.offset_in_dwords, DEBUG_OUTPUT_HEADER_TYPE_OF_LINE, payload_size_in_dwords, block.group_id);

        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS, begin.x);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 1, begin.y);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 2, begin.z);

        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 3, end.x);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 4, end.y);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 5, end.z);

        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 6, color.r);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 7, color.g);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 8, color.b);
        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 9, color.a);

        debug_communication_write_payload          (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 10, flags);
    }
}
```

It’s expanded into a series of allocation and buffer writes to build a line draw message.

It starts by atomic allocation of a message space in communication buffer.
If allocation succeeded – message header is written followed by payload data.

In this case, payload is a few floats representing line ends, line color, and line visualization flags (control space in which points specified, drawing options, etc.).



Inspection Messaging

```
if (vert_idx == (VERTEX_NUM-1)) {
    float dgb_size              = (length(poses[1]-poses[0]) + length(poses[2]-poses[0])) * 0.5f;

    debug_output_poly           (poses[0], poses[1], poses[2], float4(0.8, 0.0, 0.8, 0.004), true, 0);
    debug_output_poly           (poses[1], poses[3], poses[2], float4(0.8, 0.0, 0.8, 0.004), true, 0);

    debug_output_line           (pos, pos + R*dgb_size, float4(1.0, 0.0, 0.0, 1.0), 0);
    debug_output_line           (pos, pos + T*dgb_size, float4(0.0, 1.0, 0.0, 1.0), 0);
    debug_output_line           (pos, pos + normalize(cross(T,R))*dgb_size, float4(0.0, 0.0, 1.0, 1.0), 0);

    debug_output_line           (pos, pos + vel*dgb_size, float4(1.0, 1.0, 0.0, 1.0), 0);
}
```

```
void debug_output_line                          (in float3 begin, in float3 end, in float4 color, in uint flags)
{
    uint payload_size_in_dwords                     = 11;
    uint required_size_in_dwords                    = DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + payload_size_in_dwords;

    debug_output_block block                        = debug_output_allocate_block(required_size_in_dwords);

    if (block.is_valid)
    {
        debug_communication_write_header            (block.offset_in_dwords, DEBUG_OUTPUT_HEADER_TYPE_OF_LINE, payload_size_in_dwords, block.group_id);

        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS, begin.x);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 1, begin.y);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 2, begin.z);

        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 3, end.x);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 4, end.y);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 5, end.z);

        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 6, color.r);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 7, color.g);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 8, color.b);
        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 9, color.a);

        debug_communication_write_payload           (block.offset_in_dwords + DEBUG_OUTPUT_HEADER_SIZE_IN_DWORDS + 10, flags);
    }
}
```

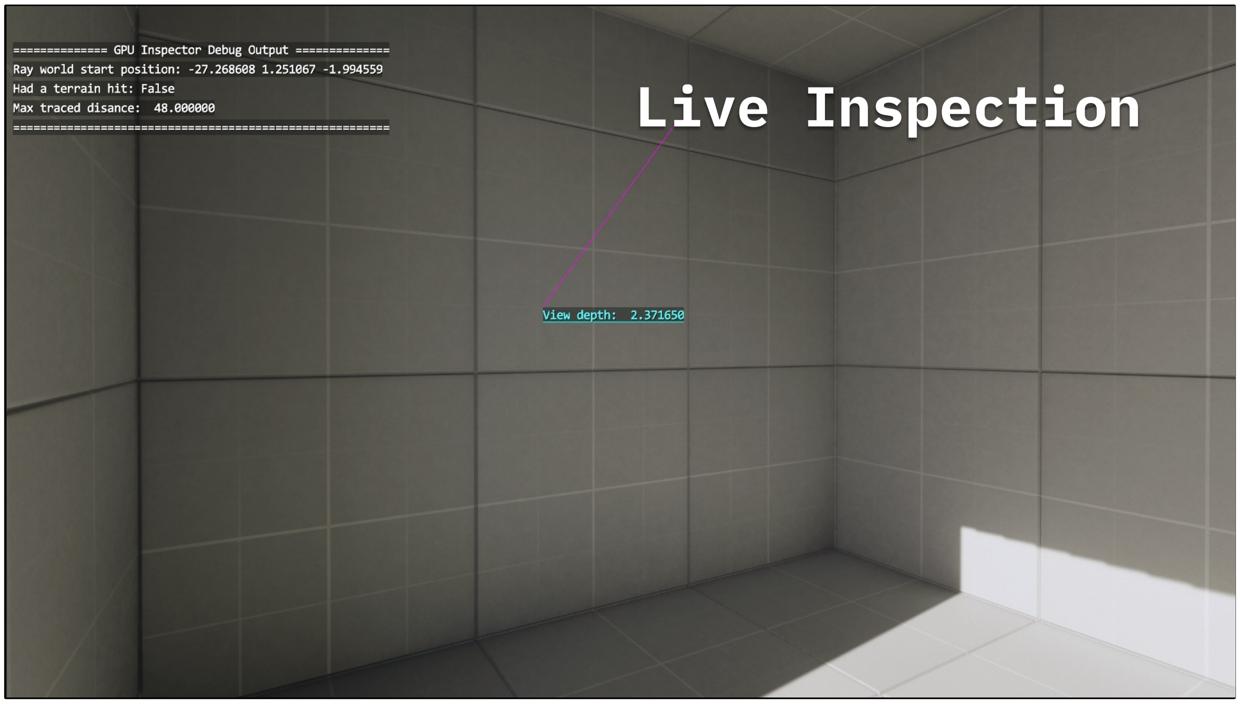
```
void debug_communication_write_payload          (in uint offset_in_dwords, float value)
{
    RWBuffer<uint> buffer                           = get_bindless_resource< RWBuffer<uint> >(COMMUNICATION_BUFFER_BINDLESS_ID);
    buffer[offset_in_dwords]                        = asuint(value);
}
```

- Call chain example.
- Same pattern for all messages.
- Processed on CPU or GPU.

Rendering Engine architecture conference 2024

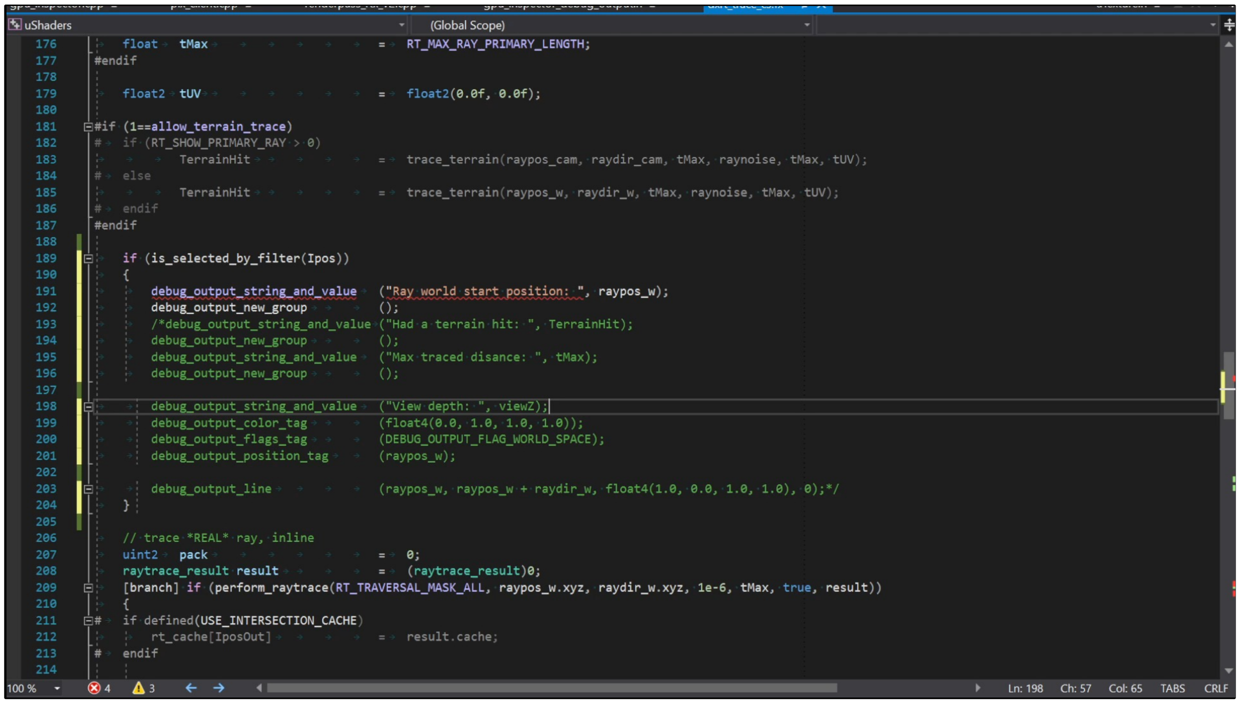
And here's the entire call chain for one message recording.
It goes from draw_output_line invocation to message generation and serialization in communication buffer.
The serialization itself is just grabbing a bindless write view putting a bunch of DWORDs into it.

And that's pretty much it. The processing side knows how to interpret payload based on message type.



```
============= GPU Inspector Debug Output =============
Ray world start position: -27.268608 1.251067 -1.994559
Had a terrain hit: False
Max traced disance:  48.000000
======================================================
```

Following video clip will show an example of live data inspection and debug drawing to visualize ray generation process for Global Illumination tracing.



```
    float   tMax                          =   RT_MAX_RAY_PRIMARY_LENGTH;
#endif

    float2  tUV                           =   float2(0.0f, 0.0f);

#if (1==allow_terrain_trace)
#   if (RT_SHOW_PRIMARY_RAY > 0)
            TerrainHit                    =   trace_terrain(raypos_cam, raydir_cam, tMax, raynoise, tMax, tUV);
#   else
            TerrainHit                    =   trace_terrain(raypos_w, raydir_w, tMax, raynoise, tMax, tUV);
#   endif
#endif

    if (is_selected_by_filter(Ipos))
    {
        debug_output_string_and_value   ("Ray world start position: ", raypos_w);
        debug_output_new_group          ();
        /*debug_output_string_and_value ("Had a terrain hit: ", TerrainHit);
        debug_output_new_group          ();
        debug_output_string_and_value   ("Max traced disance: ", tMax);
        debug_output_new_group          ();

        debug_output_string_and_value   ("View depth: ", viewZ);
        debug_output_color_tag          (float4(0.0, 1.0, 1.0, 1.0));
        debug_output_flags_tag          (DEBUG_OUTPUT_FLAG_WORLD_SPACE);
        debug_output_position_tag       (raypos_w);

        debug_output_line               (raypos_w, raypos_w + raydir_w, float4(1.0, 0.0, 1.0, 1.0), 0);*/
    }

    // trace *REAL* ray, inline
    uint2   pack                          =   0;
    raytrace_result result                =   (raytrace_result)0;
    [branch] if (perform_raytrace(RT_TRAVERSAL_MASK_ALL, raypos_w.xyz, raydir_w.xyz, 1e-6, tMax, true, result))
    {
#   if defined(USE_INTERSECTION_CACHE)
        rt_cache[IposOut]                 =   result.cache;
#   endif

```

Here, we’re in ray-tracing compute shader and we want to inspect ray-generation algorithm.

Since it’s hard to follow this process for every execution thread – we’ll filter it to include only those threads that are under mouse pointer (or virtual cursor on consoles).
That’s exactly what happens in is_selected_by_filter function. One can simply move mouse around to inspect individual threads.

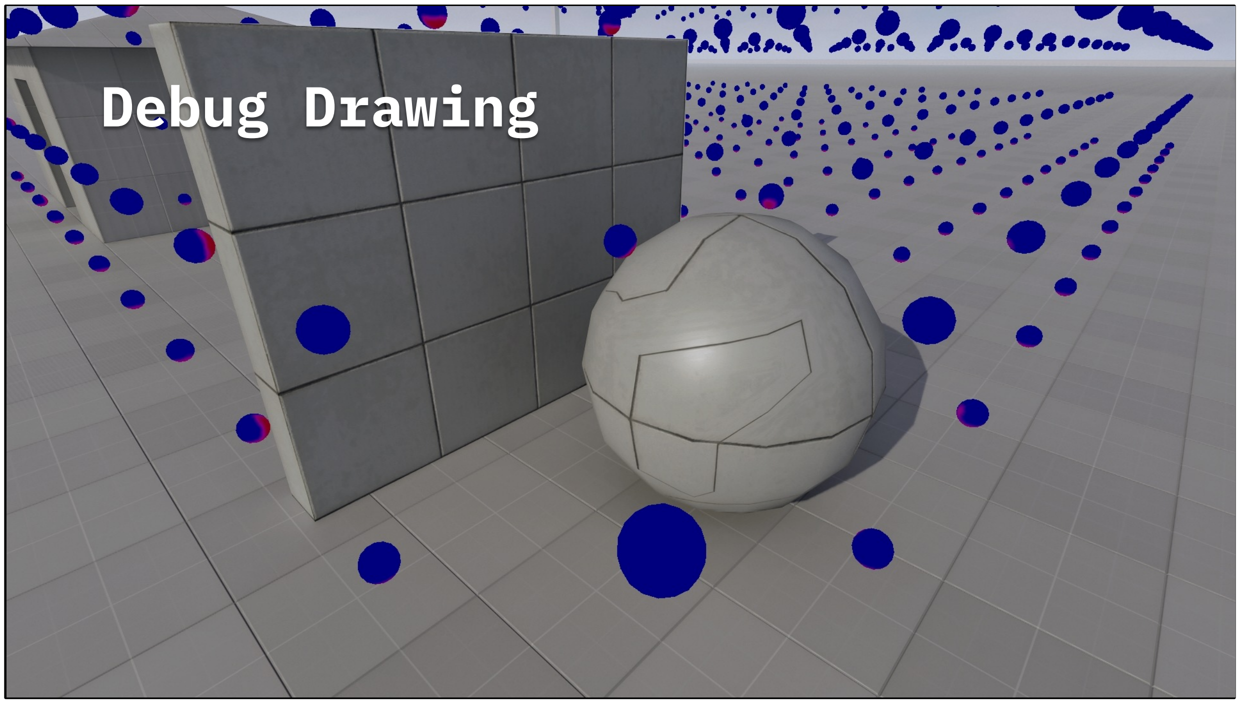
[CLICK]

Once shader is compiled and reloaded – it immediately starts to output inspection data.
Which you can see in the top left corner.

Then, we can add a few more commands to output other ray-generation parameters and to actually draw a line that represents the ray.
Again, shader compilation and reloading – and newly requested data available for inspection.

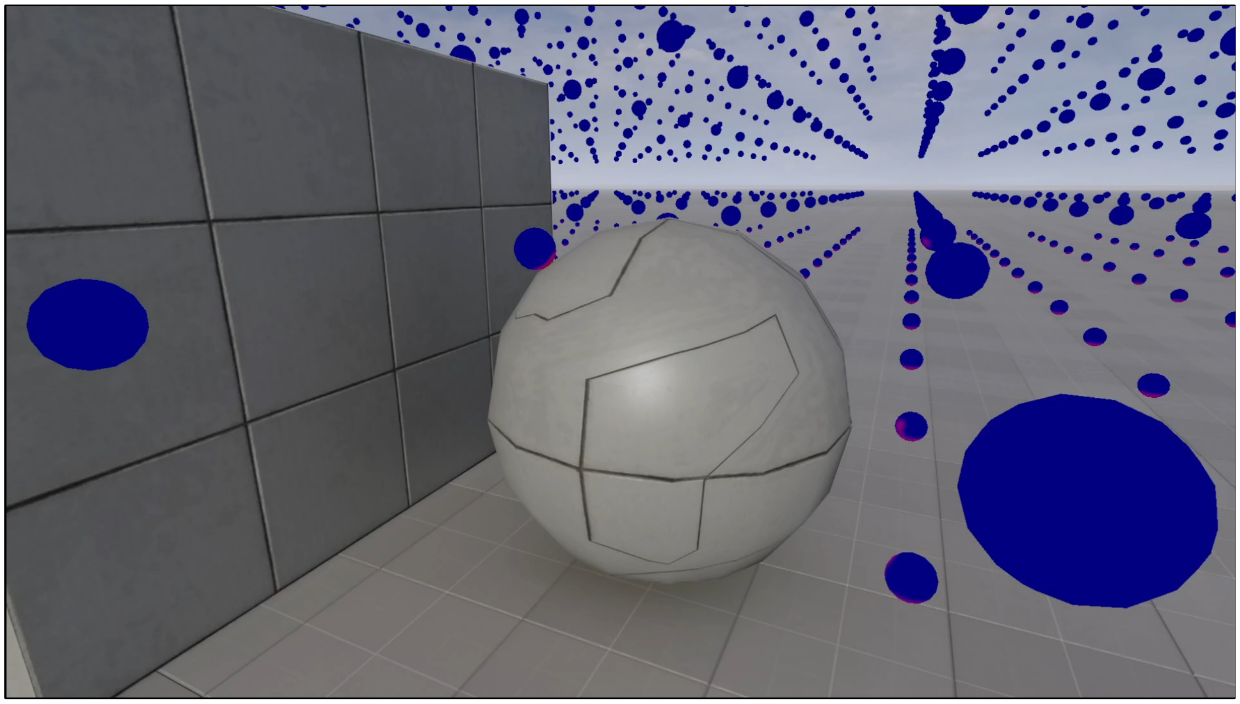
And we can see how ray direction changes every frame (the pink line) while being oriented to upper hemisphere on a surface.

(Hopefully, quick moving ray-visualization survives video encoding and streaming. If not – please, check the video once slides are online.)



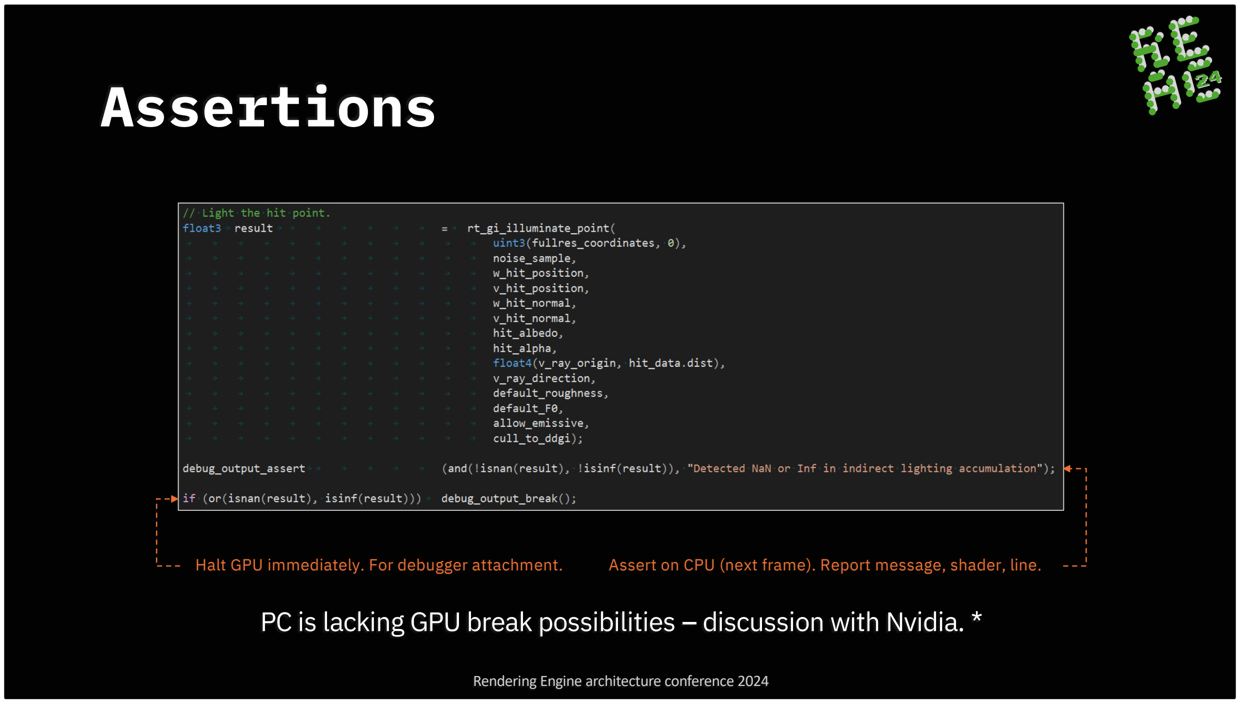
Debug Drawing

Next example shows polygons rendering to visualize data that has geometrical meaning but not stored as traditional explicit geometry.



Here, we use GPU inspection to draw distance map data. The map itself is a result of probes tracing and utilized to prevent lighting leaking, compute sound reverberation, etc.
These maps tend to have precision problems in complex geometry scenarios and it’s always useful to be able to compare expected coverage of a probe with actual coverage.

In this example, we use inspection features to draw sphere geometry where each vertex displaced from probe’s center by the distance stored in the map.
It makes it possible to inspect distance map correctness, reaction to dynamic change of probe location and surrounding geometry.



Assertions

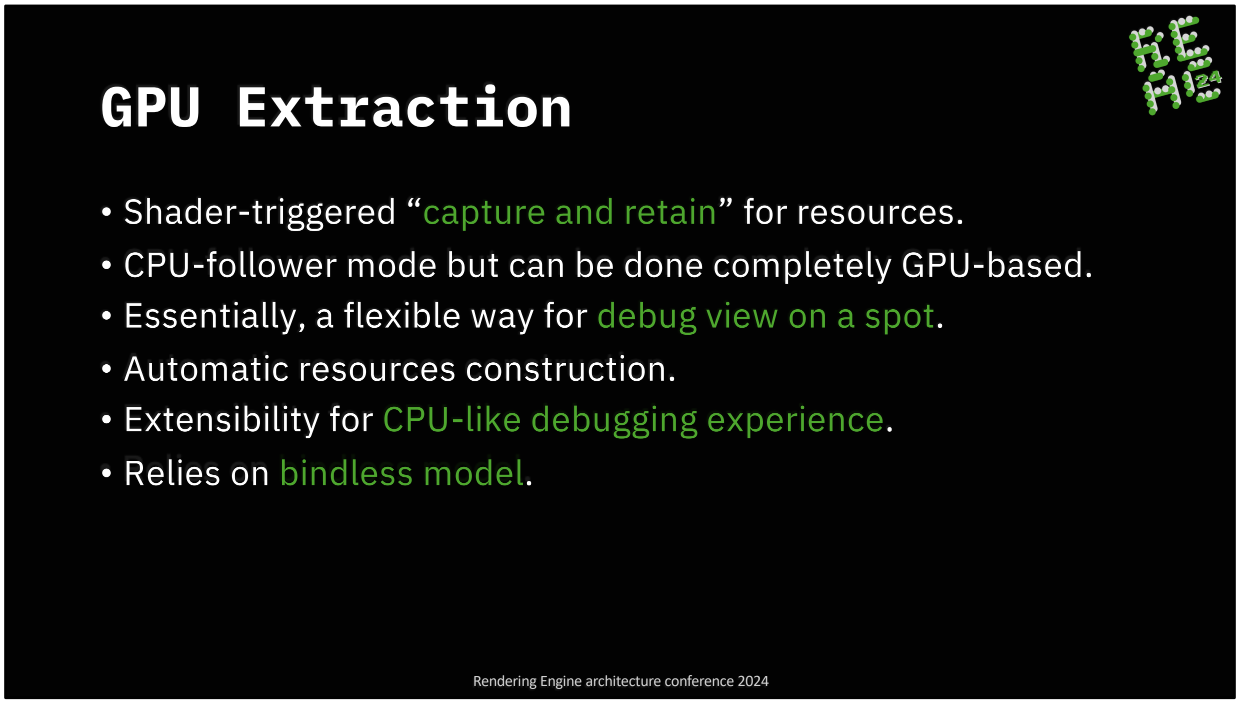
```
// Light the hit point.
float3 result                                = rt_gi_illuminate_point(
                                                 uint3(fullres_coordinates, 0),
                                                 noise_sample,
                                                 w_hit_position,
                                                 v_hit_position,
                                                 w_hit_normal,
                                                 v_hit_normal,
                                                 hit_albedo,
                                                 hit_alpha,
                                                 float4(v_ray_origin, hit_data.dist),
                                                 v_ray_direction,
                                                 default_roughness,
                                                 default_F0,
                                                 allow_emissive,
                                                 cull_to_ddgi);

debug_output_assert                          (and(!isnan(result), !isinf(result)), "Detected NaN or Inf in indirect lighting accumulation");

if (or(isnan(result), isinf(result)))        debug_output_break();
```

Halt GPU immediately. For debugger attachment.   Assert on CPU (next frame). Report message, shader, line.

PC is lacking GPU break possibilities – discussion with Nvidia. *

Rendering Engine architecture conference 2024

Also, GPU Inspection provides a possibility to trigger assert and break execution based on condition in shader code.
This functionality is very handy when looking for obscure and random issue with clear repro-steps.
And it’s a good tool when trying to catch a source of infamous NaNs of INFs.
(Especially the tricky ones when they originate in any sort of accumulation process running over time.)

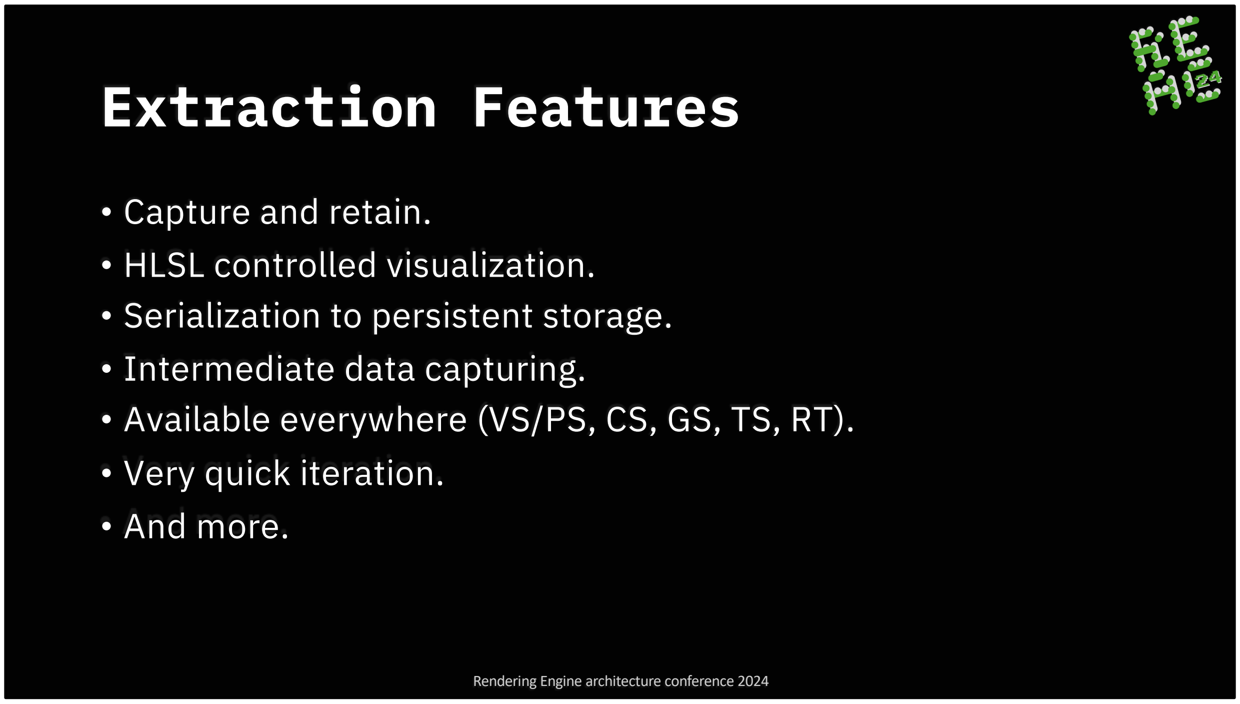


GPU Extraction

- Shader-triggered "capture and retain" for resources.
- CPU-follower mode but can be done completely GPU-based.
- Essentially, a flexible way for debug view on a spot.
- Automatic resources construction.
- Extensibility for CPU-like debugging experience.
- Relies on bindless model.

Rendering Engine architecture conference 2024

The second set of tools called GPU Extraction. It is designed around very simple goal to make it possible to capture and retain various resource types for further inspection and/or visualization.
In most cases, it is used for making snapshots of GPU resident data followed by visualization. (With no need to restart the Engine.)
Furthermore, it makes it possible to capture intermediate processing data from within shader code.



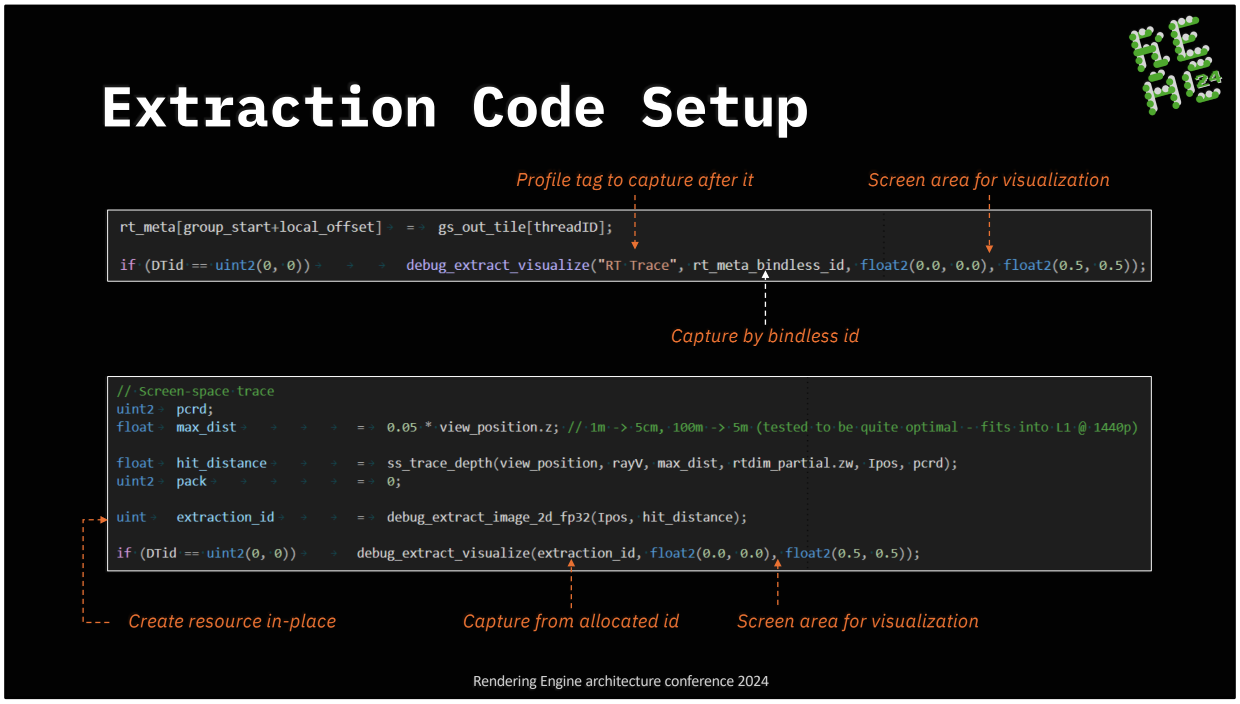
Extraction Features

- Capture and retain.
- HLSL controlled visualization.
- Serialization to persistent storage.
- Intermediate data capturing.
- Available everywhere (VS/PS, CS, GS, TS, RT).
- Very quick iteration.
- And more.

Rendering Engine architecture conference 2024

Just like with inspection – GPU Extraction is available from any shader type.
One can use this tool to:

- Specify which resource and at which point to capture/extract.
- Control visualization for captured data, build complex view layouts, etc.
- Store captured data to files for out-of-engine offline investigation.
- And more.



Extraction Code Setup

```
rt_meta[group_start+local_offset]   =   gs_out_tile[threadID];

if (DTid == uint2(0, 0))            debug_extract_visualize("RT Trace", rt_meta_bindless_id, float2(0.0, 0.0), float2(0.5, 0.5));
```

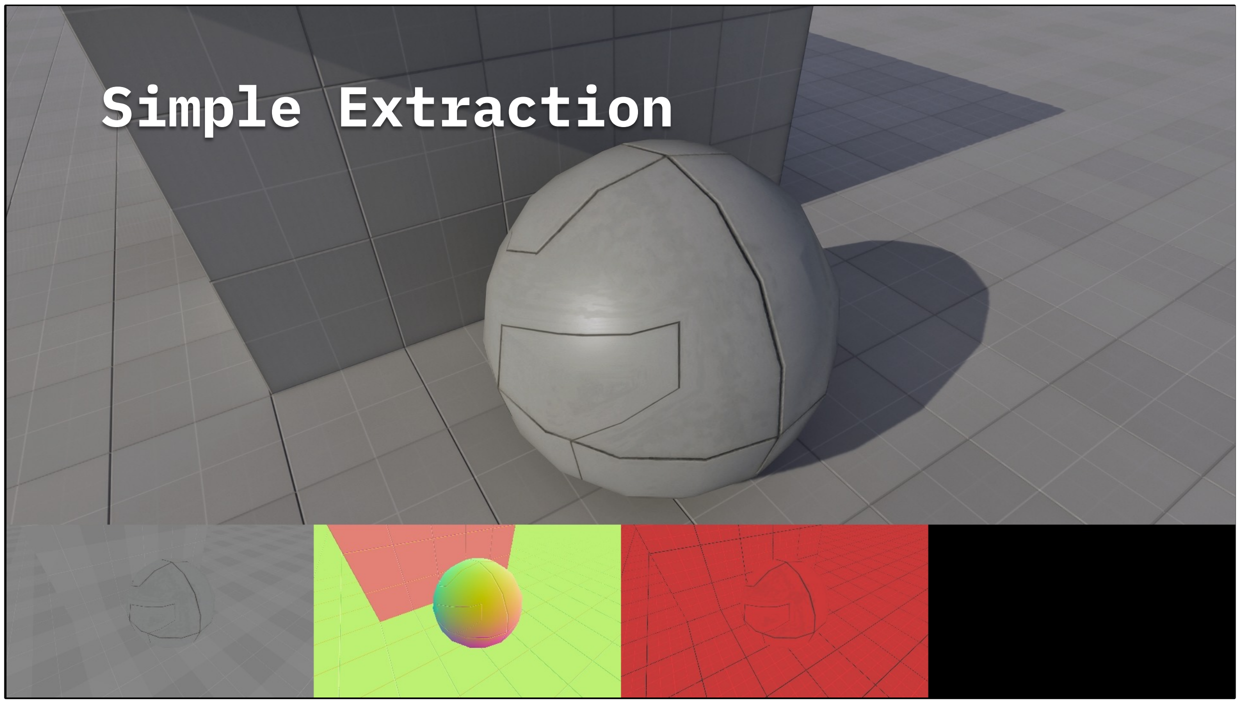
Profile tag to capture after it
Screen area for visualization
Capture by bindless id

```
// Screen-space trace
uint2   pcrd;
float   max_dist            =   0.05 * view_position.z; // 1m -> 5cm, 100m -> 5m (tested to be quite optimal - fits into L1 @ 1440p)

float   hit_distance        =   ss_trace_depth(view_position, rayV, max_dist, rtdim_partial.zw, Ipos, pcrd);
uint2   pack                =   0;

uint    extraction_id       =   debug_extract_image_2d_fp32(Ipos, hit_distance);

if (DTid == uint2(0, 0))    debug_extract_visualize(extraction_id, float2(0.0, 0.0), float2(0.5, 0.5));
```

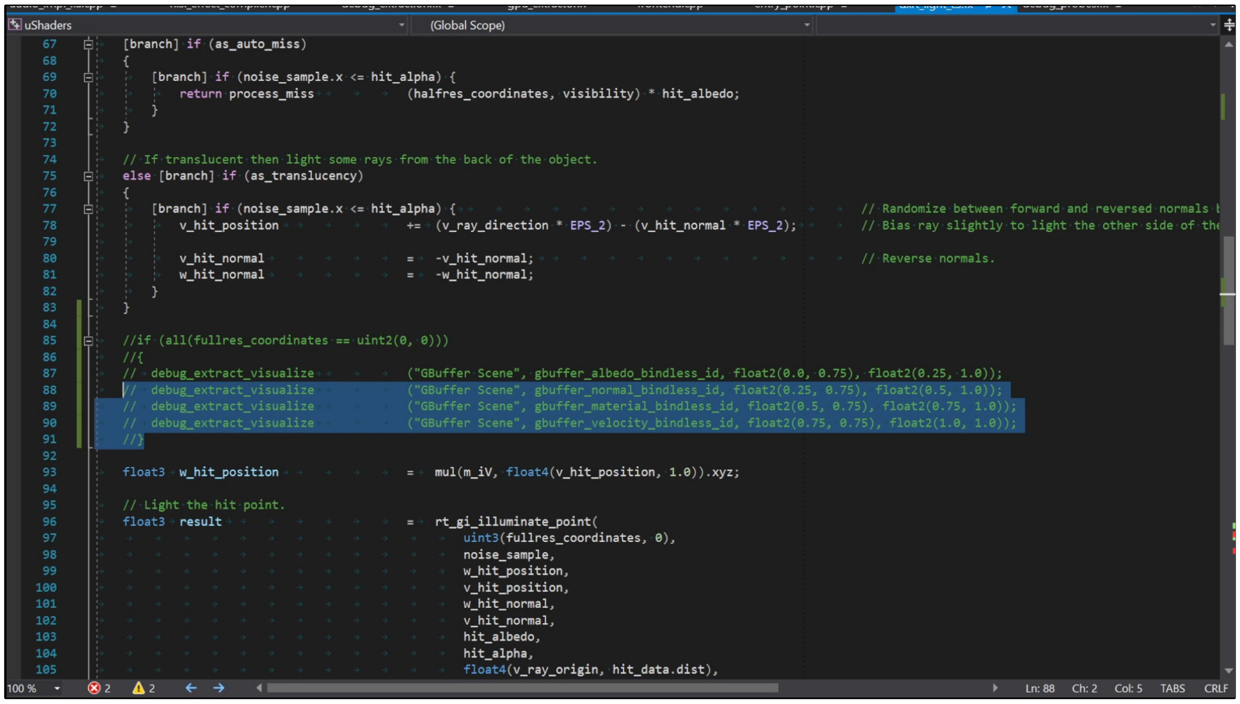
On this slide you can see two examples of GPU Extraction usage.

The top one is a simple texture capturing.
We specify GPU profiling tag to set a point in frame for capture to happen as well as bindless ID of the resource to be captured.
And immediately instruct to visualize captured texture in top-left side of the screen.

The bottom example is slightly more advances.
First – we fill-in intermediate image with per-thread data that is not stored anywhere else except VGPRs while shader code executed.
Then, we trigger another type of visualization by passing intermediate storage ID visualization properties in one go.



Following video clip shows an example of a very simple Gbuffer layers extraction with custom output layout. Everything initiated and controlled from shader code exclusively.



```
[branch] if (as_auto_miss)
{
    [branch] if (noise_sample.x <= hit_alpha) {
        return process_miss             (halfres_coordinates, visibility) * hit_albedo;
    }
}

// If translucent then light some rays from the back of the object.
else [branch] if (as_translucency)
{
    [branch] if (noise_sample.x <= hit_alpha) {                                         // Randomize between forward and reversed normals
        v_hit_position                  += (v_ray_direction * EPS_2) - (v_hit_normal * EPS_2);  // Bias ray slightly to light the other side of the

        v_hit_normal                    = -v_hit_normal;                                // Reverse normals.
        w_hit_normal                    = -w_hit_normal;
    }
}

//if (all(fullres_coordinates == uint2(0, 0)))
//{
//  debug_extract_visualize         ("GBuffer Scene", gbuffer_albedo_bindless_id, float2(0.0, 0.75), float2(0.25, 1.0));
//  debug_extract_visualize         ("GBuffer Scene", gbuffer_normal_bindless_id, float2(0.25, 0.75), float2(0.5, 1.0));
//  debug_extract_visualize         ("GBuffer Scene", gbuffer_material_bindless_id, float2(0.5, 0.75), float2(0.75, 1.0));
//  debug_extract_visualize         ("GBuffer Scene", gbuffer_velocity_bindless_id, float2(0.75, 0.75), float2(1.0, 1.0));
//}

float3 w_hit_position               = mul(m_iV, float4(v_hit_position, 1.0)).xyz;

// Light the hit point.
float3 result                       = rt_gi_illuminate_point(
                                        uint3(fullres_coordinates, 0),
                                        noise_sample,
                                        w_hit_position,
                                        v_hit_position,
                                        w_hit_normal,
                                        v_hit_normal,
                                        hit_albedo,
                                        hit_alpha,
                                        float4(v_ray_origin, hit_data.dist),
```

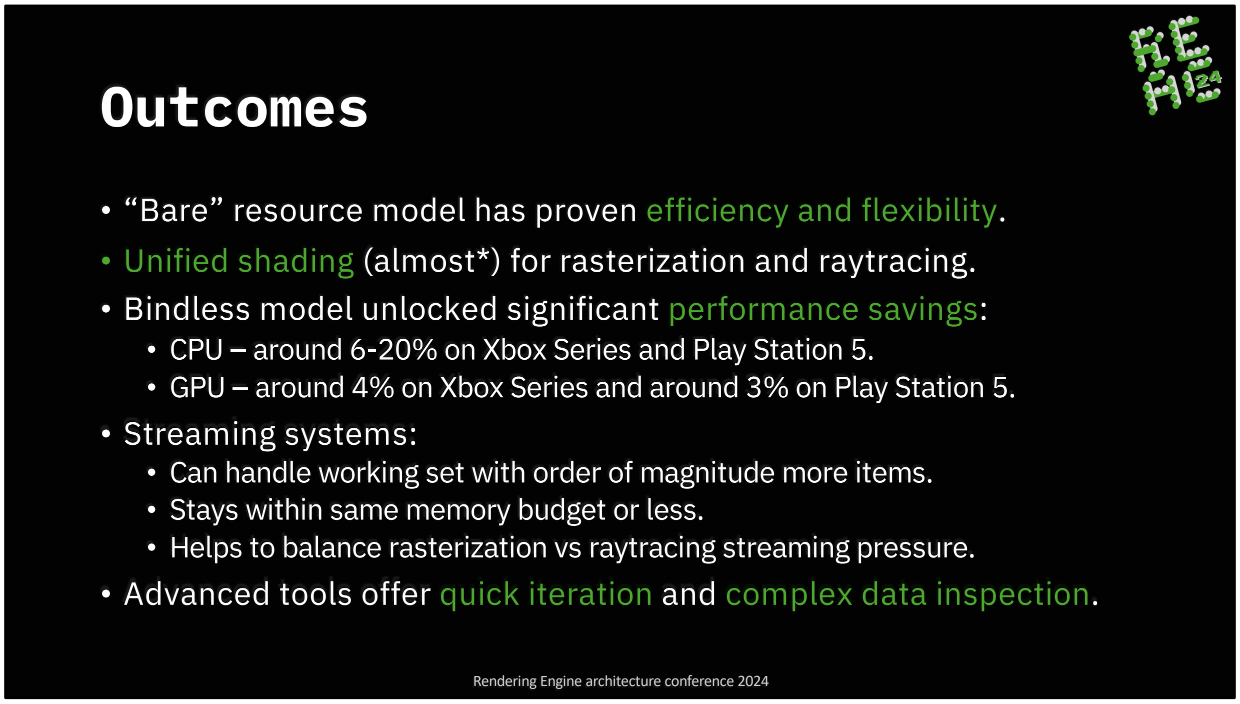
Here we’re in indirect lighting shader. But, because the tool itself available everywhere and there’s access to bindless IDs of Gbuffer layers – we can schedule extraction from here.

In this case it’s a trivial extraction from existing resources. We just specify the profiling tag to be right after Gbuffer generation.
Once shader is recompiled and reloaded – the extraction starts and runs every frame.

This is approach to in-place data visualization is rather useful as one no need to changes CPU code to create extra writable texture, binding that texture to producing shaders, adding visualization code, and so on and so on.

Conclusions

That concludes three topics we wanted to cover in this talk although there's much more left aside due to time constraints.
There's more to in bonus slide though, so, please, feel free to check them out.
And now it's just fitting to summarize the outcomes generated by the solutions proposed.



Outcomes

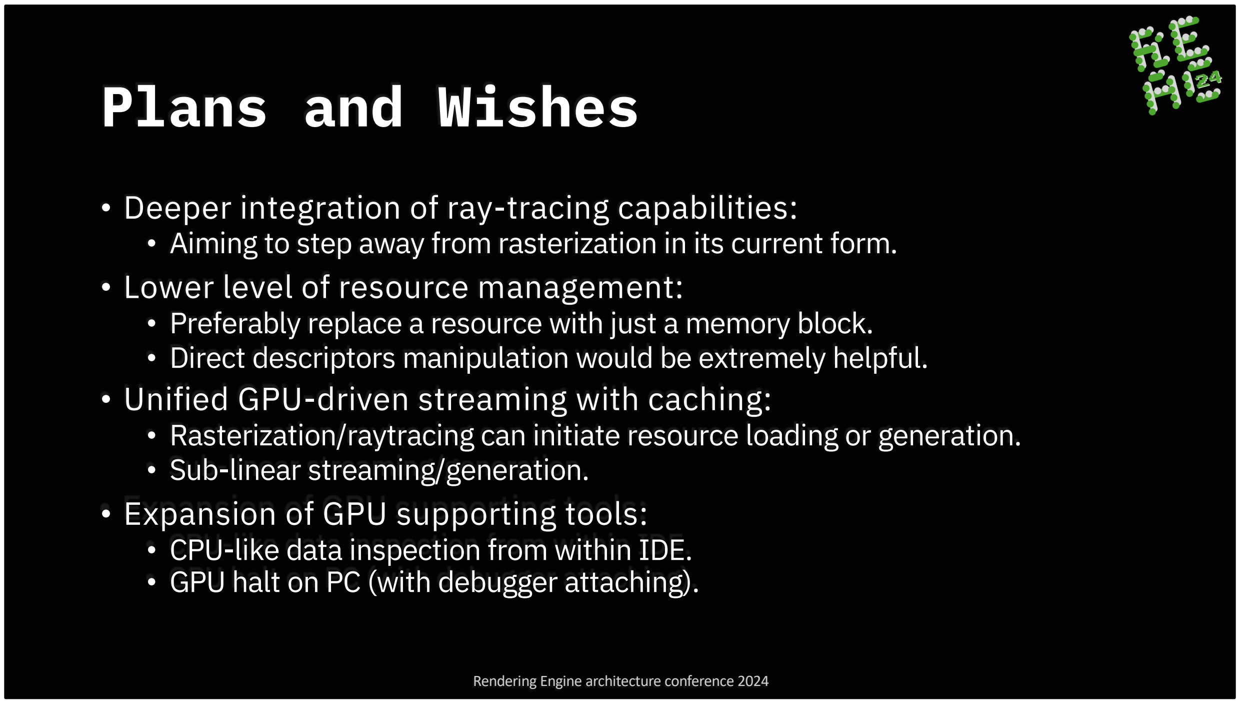
- “Bare” resource model has proven efficiency and flexibility.
- Unified shading (almost*) for rasterization and raytracing.
- Bindless model unlocked significant performance savings:
  - CPU – around 6-20% on Xbox Series and Play Station 5.
  - GPU – around 4% on Xbox Series and around 3% on Play Station 5.
- Streaming systems:
  - Can handle working set with order of magnitude more items.
  - Stays within same memory budget or less.
  - Helps to balance rasterization vs raytracing streaming pressure.
- Advanced tools offer quick iteration and complex data inspection.

While being hesitant about “bare” abstraction level for resources it has proven to be very efficient and surprisingly easy to use.
Proposed management and streaming solutions utilize this “bare” and model to address increasing scene complexity and to produce higher visual quality while staying within original memory and performance budgets.

Explicit separation of views and resources made it so much easier to lift many traditionally low-level systems to cross-platform code.
Furthermore, it simplified bindless model implementation by quite a bit.

With bindless model we managed to lay down a foundation for unified shading in rasterization and ray-trace.
Even more so - to improve CPU *and* GPU performance.

Finally, tools developed primarily for new systems debugging, happened to be very useful when working on other graphics systems and visual effects.
It definitely helped to stick to the “Fast Iteration” design principle and even improve on it.



Plans and Wishes

- Deeper integration of ray-tracing capabilities:
  - Aiming to step away from rasterization in its current form.
- Lower level of resource management:
  - Preferably replace a resource with just a memory block.
  - Direct descriptors manipulation would be extremely helpful.
- Unified GPU-driven streaming with caching:
  - Rasterization/raytracing can initiate resource loading or generation.
  - Sub-linear streaming/generation.
- Expansion of GPU supporting tools:
  - CPU-like data inspection from within IDE.
  - GPU halt on PC (with debugger attaching).

Rendering Engine architecture conference 2024

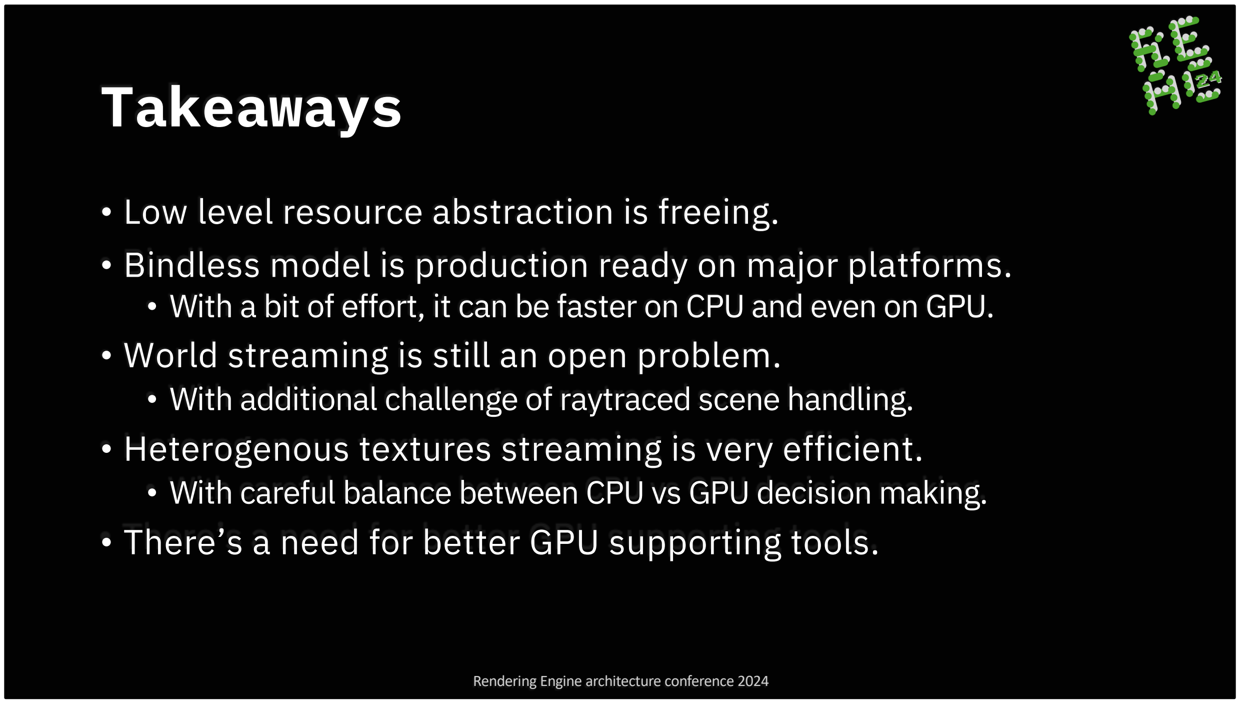
But, as with any solution – there's a number of compromises and shortcomings to be addressed in nearest future (hopefully).

First of all, we're looking into opportunities to unify scene description for rasterization and ray-tracing, and, also, to make ray-tracing passes capable of driving streaming decision just like rasterization does.

Second. while view/resource split has many useful properties – it's still an abstraction level which has cost (despite being minimal).
Introduction of actual resource descriptors as memory blocks with direct access and ability to construct, copy, and query would be extremely helpful to replace views and to move resources management to GPU completely.

As for the GPU supporting tools – the discussed ones are a definitive improvements and it would be great to, at least, reach parity with consoles on PC.
On that note, we discussed with NVIDIA at some point a possibility to halt GPU execution for debugging properties (using multiple GPUs or just remote machine).

Having functionality like that would make an unparalleled improvement for PC GPU debugging experience.



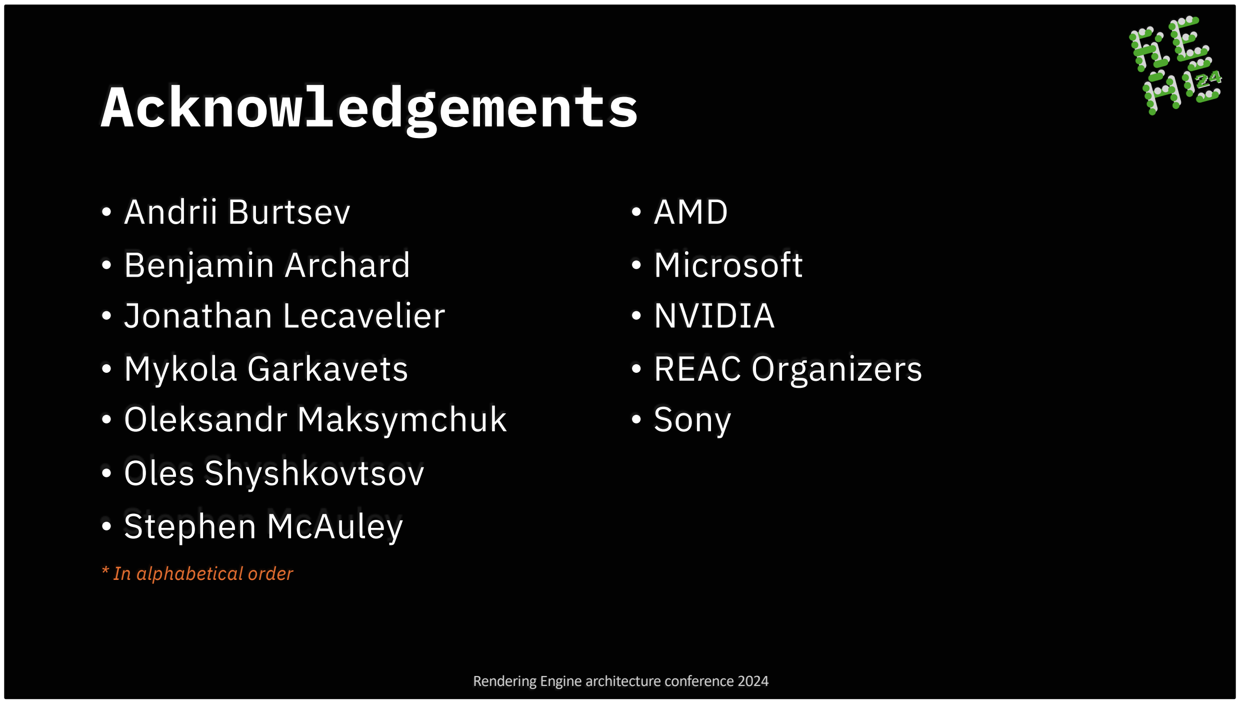
Takeaways

- Low level resource abstraction is freeing.
- Bindless model is production ready on major platforms.
  - With a bit of effort, it can be faster on CPU and even on GPU.
- World streaming is still an open problem.
  - With additional challenge of raytraced scene handling.
- Heterogenous textures streaming is very efficient.
  - With careful balance between CPU vs GPU decision making.
- There’s a need for better GPU supporting tools.

Rendering Engine architecture conference 2024

Finally, if I had to distill this talk to one slide it would, probably, be this one. Hopefully, it represents main messages we wanted to communicate:

- Bare low-level resources exposure sounds unsettling at first but after using it for a fair bit – we’re very happy with this decision.
- Bindless model is most definitely ready for usage in big AAA projects, and it can be faster than traditional direct binding with a little bit of effort.
- Efficient streaming solutions still very much an open problem, especially when it comes to ray-traced scene representation streaming.
- A combination of CPU and GPU metrics to prioritize streaming requests is efficient and performant.
- And these days we need flexible and fast GPU debugging and inspection tools more than ever as more and more work and decision making migrates to GPU.



Acknowledgements

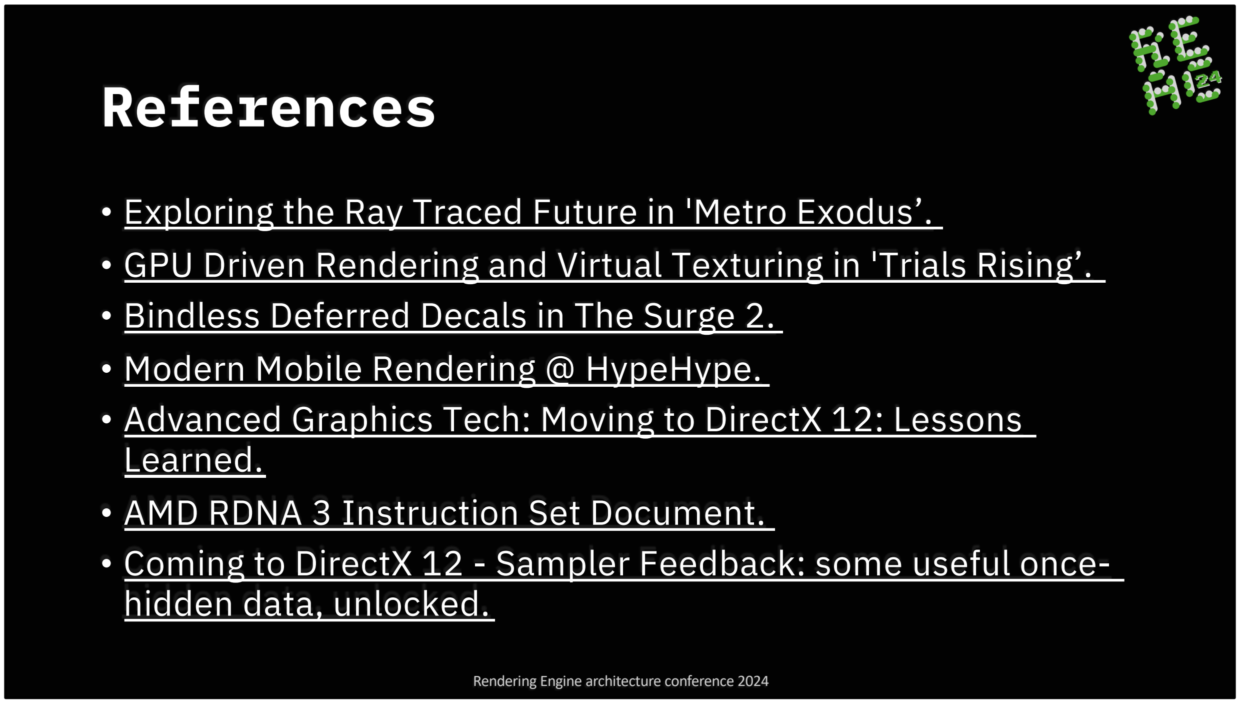
- Andrii Burtsev
- Benjamin Archard
- Jonathan Lecavelier
- Mykola Garkavets
- Oleksandr Maksymchuk
- Oles Shyshkovtsov
- Stephen McAuley

** In alphabetical order*

- AMD
- Microsoft
- NVIDIA
- REAC Organizers
- Sony

Rendering Engine architecture conference 2024

And with that, we've reached the end of the talk but, before jumping to to Q&A, I'd like to take this opportunity to thank the following people for their help and contribution (either in code or advice or presentation reviews).
I would also like to thank AMD, Microsoft, NVIDIA, and Sony for the valuable insight and suggestions that helped to design solutions presented in the talk.



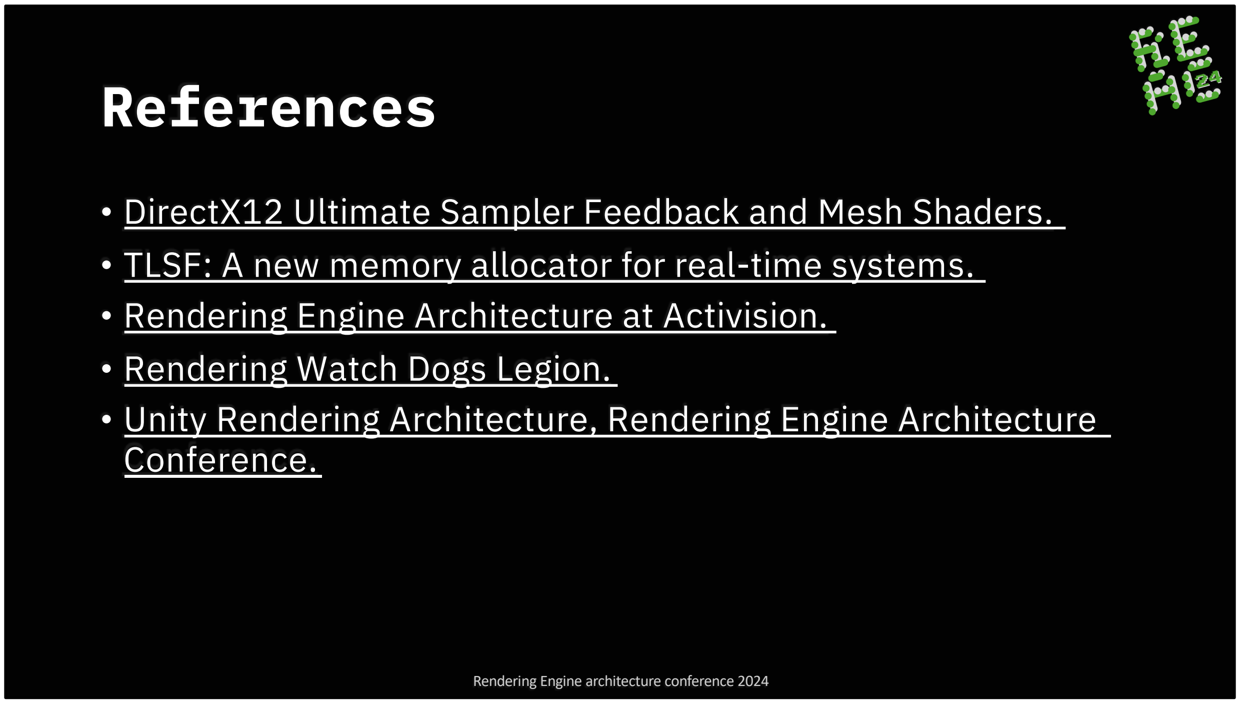
References

- Exploring the Ray Traced Future in 'Metro Exodus'.
- GPU Driven Rendering and Virtual Texturing in 'Trials Rising'.
- Bindless Deferred Decals in The Surge 2.
- Modern Mobile Rendering @ HypeHype.
- Advanced Graphics Tech: Moving to DirectX 12: Lessons Learned.
- AMD RDNA 3 Instruction Set Document.
- Coming to DirectX 12 - Sampler Feedback: some useful once-hidden data, unlocked.

Rendering Engine architecture conference 2024

A lot of talented people helped by indirectly by sharing their knowledge and experience and this talk would have been completely different (if at all possible) without influence of these sources.

So, should you want to dive deeper in some of the topic mentioned throughout the talk – please check out this list.

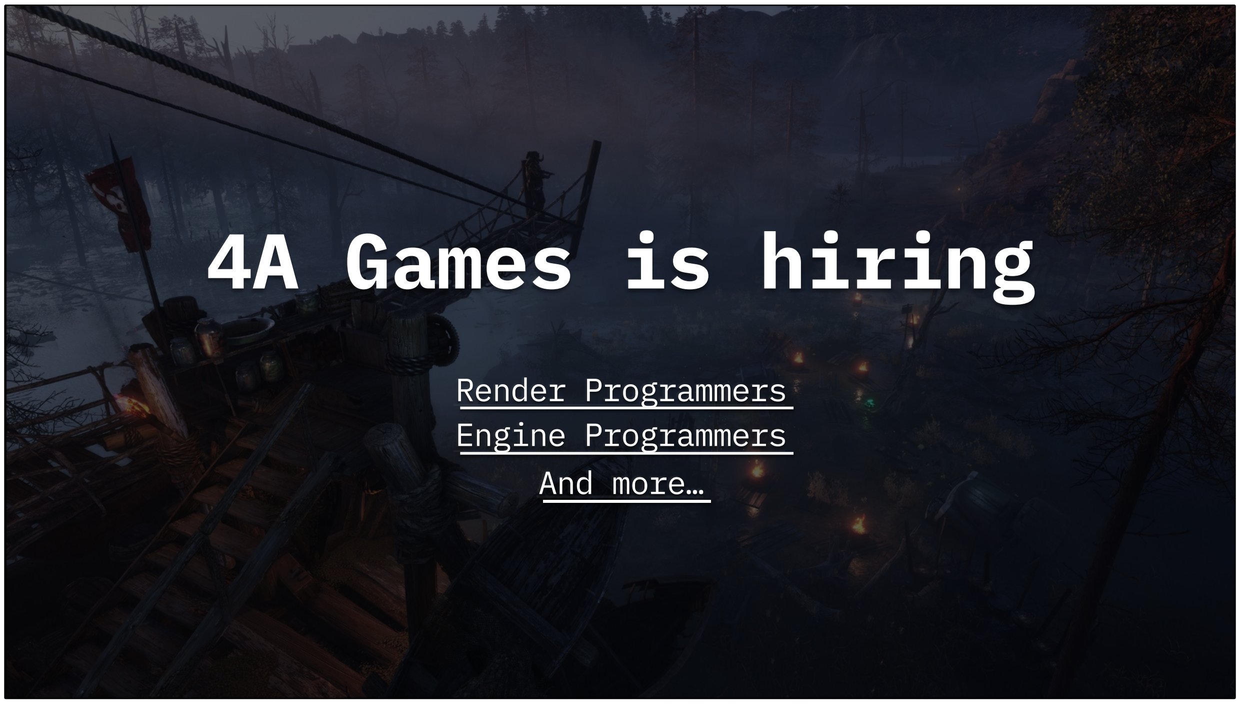


References

- DirectX12 Ultimate Sampler Feedback and Mesh Shaders.
- TLSF: A new memory allocator for real-time systems.
- Rendering Engine Architecture at Activision.
- Rendering Watch Dogs Legion.
- Unity Rendering Architecture, Rendering Engine Architecture Conference.

Rendering Engine architecture conference 2024

Also, I encourage you to look-up the deck once it's released as it has quite a lot of information in speaker notes and some bonus slides.



4A Games is hiring

Render Programmers

Engine Programmers

And more…

Also, if you're interested in low-level graphics code, ray-tracing, and general Engine architecture – check out these open positions (the links attached).

Thank you!

Questions?

RENDERING ENGINE ARCHITECTURE CONFERENCE REAC 24

Thank you all very much for tuning in. I really hope it was useful and justified investment of your time. I'm happy to answer any questions and I hope you have some. Thank you.