

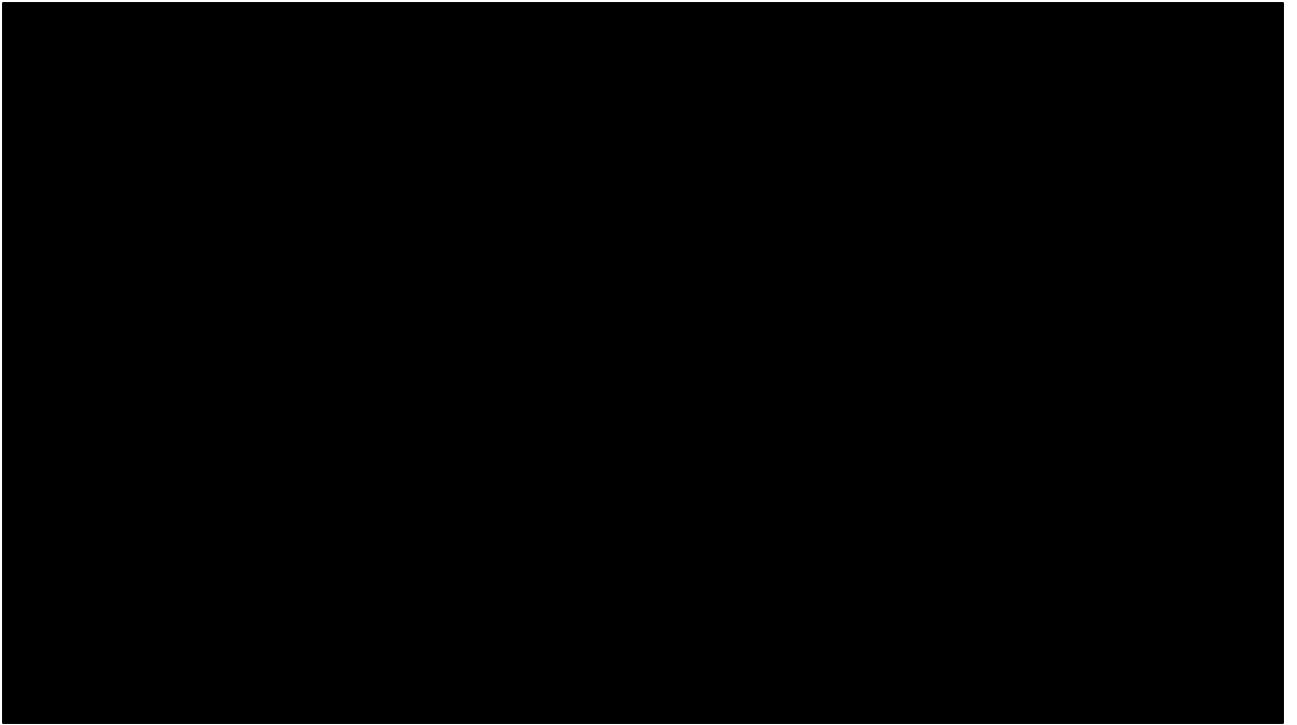


My name is Graham Wihlidal, and I'm a senior rendering engineer at SEED in Stockholm, Sweden. Previously, I was on the Frostbite rendering team, and at BioWare for a long time.



SEED

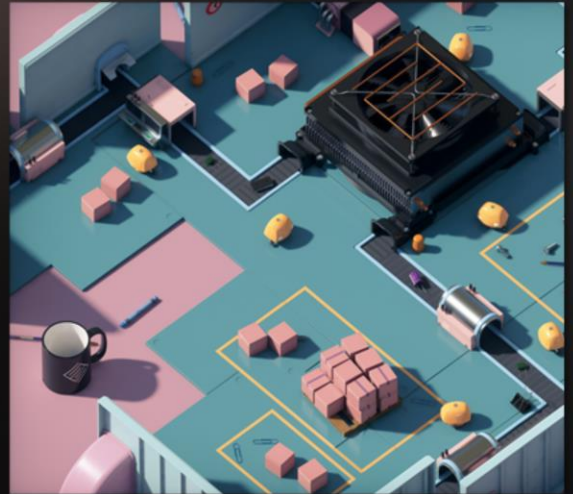
SEED is a technical and creative research division of Electronic Arts. We are a cross-disciplinary team with a mission to explore the future of interactive entertainment. One of our recent projects is an experiment in hybrid real-time rendering, deep learning agents, and procedural level generation.



Here is a video we circulated, showing our recent real-time ray tracing work.

# “PICA PICA”

- Exploratory mini-game & world
- Goals
  - Hybrid rendering with DXR [Andersson 2018]
  - Clean and consistent visuals
  - Self-learning AI agents [Harmer 2018]
  - Procedural worlds [Opara 2018]
  - No precomputation
- Uses SEED's **Halcyon** R&D framework



We have built the PICA PICA from the ground up in our custom R&D framework called Halcyon. It is a flexible experimentation framework that is very capable of rendering fast and shiny pixels.



So lets talk a bit about Halcyon itself

# Halcyon Goals

- Rapid prototyping framework
- Different purpose than Frostbite
  - Fast experimentation vs. AAA games
- Windows, Linux, macOS

Halcyon is a rapid prototyping framework, serving a different purpose than our flagship AAA engine, Frostbite.

And Halcyon is currently supported on Windows, Linux, and macOS.

# Halcyon Goals

- Minimize or eliminate busy-work
  - Artist “meta-data” meshes
    - Occlusion
    - GI / Lighting
    - Collision
    - Level-of-detail
- Live reloading of all assets
  - Insanely fast iteration times

A major goal of Halcyon is to minimize or eliminate busy-work; something I call - artist “meta-data” meshes.

Show me one artist that actually enjoys making these meshes over something more creative, and I guarantee you they are brainwashed, and need an intervention and our caring support.

Another critical goal, is the live reloading of all assets. We don’t want to take a coffee break while we shut down Halcyon, launch a data build, come back, and resume whatever we were doing.

# Halcyon Goals

- Only target modern APIs
  - Direct3D 12
  - Vulkan 1.1
  - Metal 2
- Multi-GPU
  - Explicit heterogeneous mGPU
  - No AFR nonsense
  - No linked adapters

One luxury we had by starting from scratch, was choosing our feature set and min spec. We decided to only target modern APIs, so we are not restricted by legacy.

Another interesting goal is to provide easy access to multiple GPUs, without sacrificing API cleanliness or maintainability. To accomplish this, we decided on explicit heterogeneous mGPU, not linked adapters.

We also are avoiding any AFR nonsense for a number of reasons, including problems with temporal techniques.



# Halcyon Goals

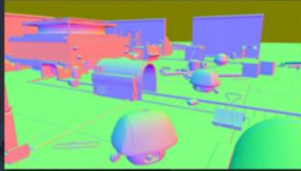
- Local or remote streaming
- Minimal boilerplate code
- Variety of rendering techniques and approaches
  - Rasterization
  - Path and ray tracing
  - Hybrid

We chose to support rendering locally, and also performing some computation or rendering remotely, and transmitting the results back to the application.

In order to deliver on our promise of fast experimentation, we needed to ensure a minimal amount of boilerplate code. This includes code to load a shader, set up pipelines and render state, query scene representations, etc.

This was critical in developing and supporting a vast number of rendering techniques and approaches that we have in Halcyon.

# Hybrid Rendering



Deferred Shading  
(raster)



Direct Shadows  
(ray trace or raster)



Direct Lighting  
(compute)



Reflections  
(ray trace or compute)



Global Illumination  
(ray trace)



Ambient Occlusion  
(ray trace or compute)



Transparency & Translucency  
(ray trace)



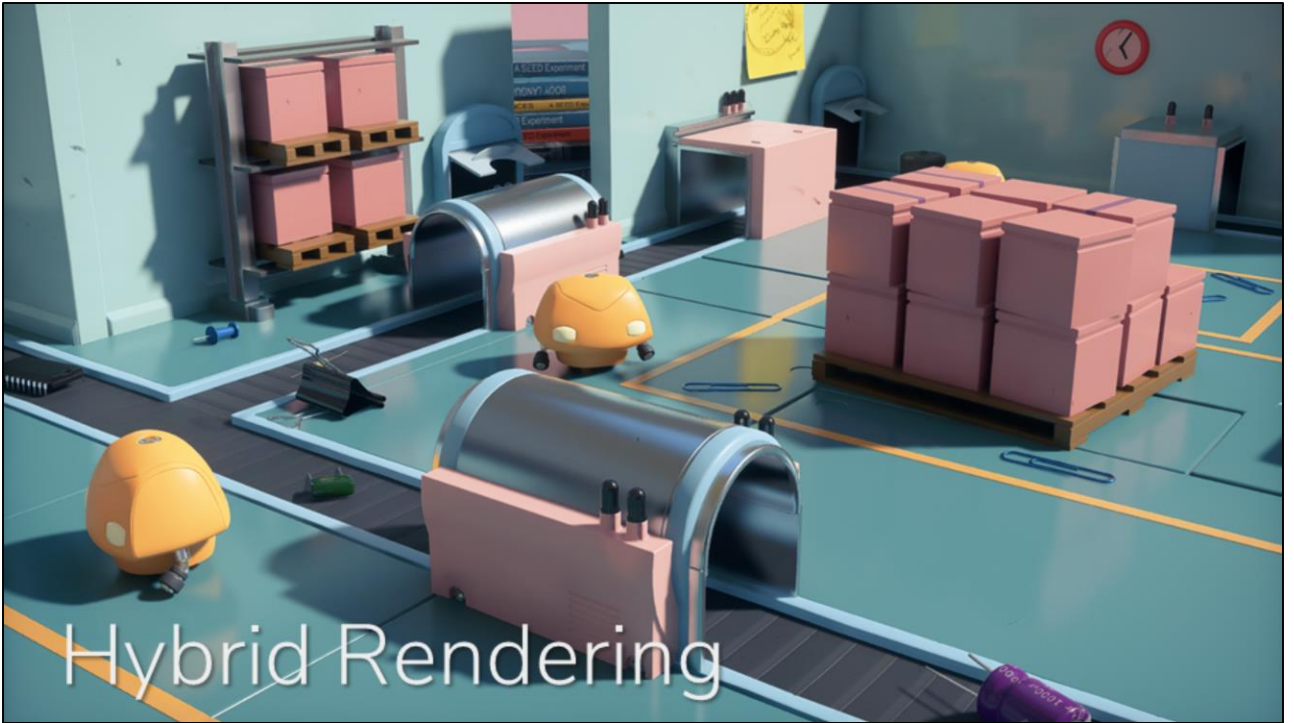
Post Processing  
(compute)

We have a unique rendering pipeline which is inspired by classic techniques in real-time rendering, with ray tracing sprinkled on top.

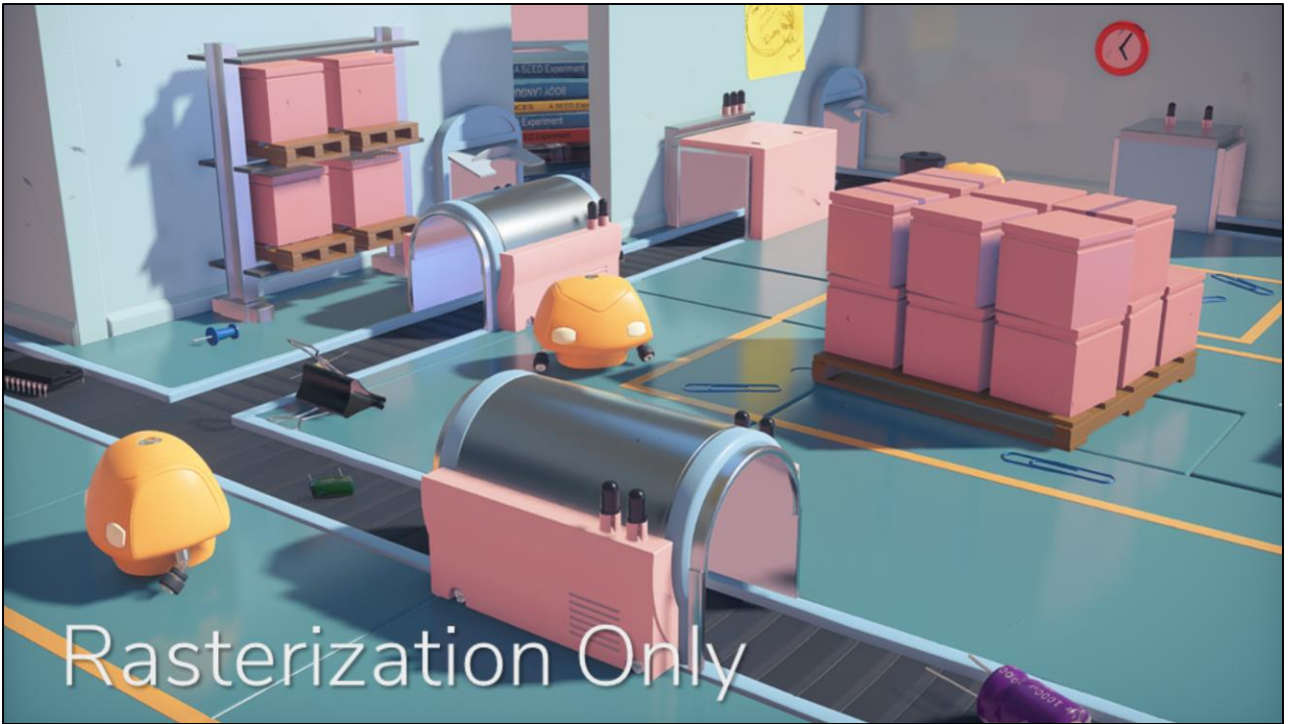
We have a deferred renderer with compute-based lighting, and a fairly standard post-processing stack. And then there are a few pluggable components. We can render shadows via ray tracing or cascaded shadow maps.

Reflections can be traced or screen-space marched. Same story for ambient occlusion.

Only our global illumination and translucency actually require ray tracing.



Here is an example of a scene using our hybrid rendering pipeline



And here is the same scene uses our traditional pure rasterization rendering pipeline.

As you can see, most of our visual fidelity holds up between the two, especially with respect to our materials.

# Halcyon Goals

- “PICA PICA” and Halcyon **built from scratch**
  - Implemented lots of bespoke technology
- Minimal effort to add a new API or platform
- **Efficient and flexible rendering** was a major focus

PICA PICA and Halcyon were both built from scratch, and our goals required us to implement a lot of bespoke technology.

In the end, our flexible architecture means it is minimal effort to add a new API or platform, and also render our scenes very efficiently.

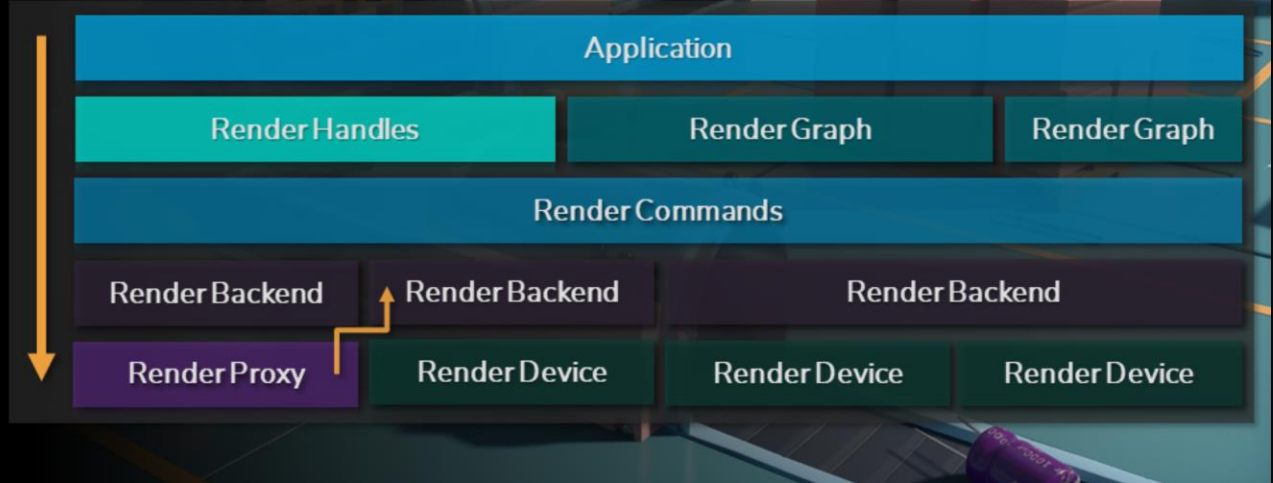
SEED // Halcyon + Vulkan

# Rendering Components

- Render Backend
- Render Device
- Render Handles
- Render Commands
- Render Graph
- Render Proxy



# Rendering Components





# Render Backend

The first component I will talk about is render backend



# Render Backend

- Live-reloadable DLLs
- Enumerates adapters and capabilities
  - Swap chain support
  - Extensions (i.e. ray tracing, sub groups, ...)
  - Determine adapter(s) to use

Render backends are implemented as live-reloadable DLLs. Each backend represents a particular API, and provides enumeration of adapters and capabilities. In addition, the backends will determine the ideal adapters to use, depending on the desired purpose.

# Render Backend

- Provides debugging and profiling
  - RenderDoc integration, validation layers, ...
- Create and destroy render devices

Each render backend also supports a debugging and profiling interface – this provides functionality like RenderDoc integration, CPU and GPU validation layers, etc..

Most importantly, render backends support the creation and destruction of render devices, which I'll cover shortly.

SEED // Halcyon + Vulkan

# Render Backend

- Direct3D 12
- Vulkan 1.1
- Metal 2
- Proxy
- Mock



We have a variety of render backends implemented

# Render Backend

- Direct3D 12
  - Shader Model 6.X
  - **DirectX Ray Tracing**
  - Bindless Resources
  - Explicit Multi-GPU
  - **DirectML** (soon..)
  - ...



For DirectX 12, we support the latest shader model, DirectX ray tracing, full bindless resources, explicit heterogeneous mGPU, and we plan to add support for DirectML.

# Render Backend

- Vulkan 1.1
  - Sub-groups
  - Descriptor indexing
  - External memory
  - Multi-draw indirect
  - Ray tracing (soon..)
  - ...



Vulkan is a similar story to DirectX 12, except we haven't implemented multi-GPU or ray tracing support at this time, but it is planned.

# Render Backend

- Metal 2
  - Early development
  - Primarily desktop
  - Argument buffers
  - Machine learning
  - ...



Metal 2 is still in early development, but very promising. We are primarily interested in using it on macOS instead of iOS.

# Render Backend

- Proxy
  - Not discussed in this presentation

We have another pretty crazy backend which I unfortunately won't get into today, but a near future presentation will go in depth on this one.

# Render Backend

- Mock
  - Performs resource tracking and validation
  - Command stream is parsed and evaluated
  - **No submission to an API**
  - Useful for **unit tests** and **debugging**

Finally, we have our mock render backend which is for unit testing, debugging, and validation. This backend does all the same work the other backends do, except translation from high level to low level just runs a validation test suite instead of submitting to a graphics API.





The next component to discuss is render device

# Render Device

- Abstraction of a logical GPU adapter
  - e.g. VkDevice, ID3D12Device, ...
- Provides interface to GPU queues
- Command list submission

Render device is an abstraction of a logical GPU adapter, such as VkDevice or ID3D12Device.

It provides an interface to various GPU queues, and provides API specific command list scheduling and submission functionality

# Render Device

- Ownership of GPU resources
  - Create & Destroy
- Lifetime tracking of resources
- Mapping render handles → device resources

Render device has full ownership and lifetime tracking of its GPU resources, and provides create and destroy functionality for all the high level render resource types.

The high level application refers to these resources using handles, so render device also internally provides an efficient mapping of these handles to internal device resources.



Speaking of render handles...

# Render Handles

- Resources associated by handle
- Lightweight (64 bits)
- Constant-time lookup
- Type safety (i.e. buffer vs texture)
- Can be serialized or transmitted
- Generational for safety
  - e.g. double-delete, usage after delete

The handles are lightweight (just 64 bits), and the associated resources are fetched with a constant-time lookup. The handles are type safe, like trying to pass a buffer in place of a texture.

The handles can be serialized or transmitted, as long as some contract is in place about what the handle represents.

And the handles are generational for safety, which protects against double-delete, or using a handle after deletion

# Render Handles

- Handles allow **one-to-many cardinality** [handle->devices]
- Each device can have a unique representation of the handle



Handles allow a one-to-many cardinality where a single handle can have a unique representation on each device

# Render Handles

- Can query if a device has a handle loaded
- Safely add and remove devices
  - Handle owned by application, representation can reload on device



There is an API to query if a particular render device has a handle loaded or not. This makes it safely add and remove devices while the application is running. The handle is owned by the application, so the representation can be loaded or reloaded for any number of devices.

# Render Handles

- Shared resources are supported
- Primary device owner, secondaries alias primary



Shared resources are also supported, allowing for copying data between various render devices.



# Render Handles

- Can also **mix and match backends** in the **same process!**
  - Made debugging VK implementation much easier
  - **DX12 on left half** of screen, **VK on right half** of screen



A crazy feature of this architecture is that we can mix and match different backends in the same process!

This made debugging Vulkan much easier, as I could have Dx12 rendering on the left half of the screen, while Vulkan was rendering on the right half of the screen.



An key rendering component is our high level command stream

# Render Commands

- Draw
- DrawIndirect
- Dispatch
- DispatchIndirect
- UpdateBuffer
- UpdateTexture
- CopyBuffer
- CopyTexture
- Barriers
- Transitions
- BeginTiming
- EndTiming
- ResolveTimings
- BeginEvent
- EndEvent
- BeginRenderPass
- EndRenderPass
- RayTrace
- UpdateTopLevel
- UpdateBottomLevel
- UpdateShaderTable

We developed an API agnostic high level command stream that allows the application to efficiently express how a scene should be updated and rendered, but letting each backend control how this is done.

# Render Commands

- Queue type specified
- Spec validation
  - Allowed to run?
  - **e.g. draws on compute**
- Automatic scheduling
  - **Where can it run?**
  - Async compute

```
enum class RenderCommandQueueType : uint8
{
    None = 0x00,
    Copy = 0x01,
    Compute = 0x02,
    Graphics = 0x04,
    All = Copy | Compute | Graphics,
};

struct RenderCommand
{
    RenderCommandType type = RenderCommandType::Count;
};

template<RenderCommandType TYPE, RenderCommandQueueType QUEUETYPE>
struct RenderCommandTyped : RenderCommand
{
    static const RenderCommandType Type = TYPE;
    static const RenderCommandQueueType QueueType = QUEUETYPE;

    RenderCommandTyped() { type = Type; }
};
```

Each render command specifies a queue type, which is primarily for spec validation (such as putting graphics work on a compute queue), and also to aid in automatic scheduling, like with async compute.

SEED // Halcyon + Vulkan

# Render Commands

```
struct RenderCommandDispatch : RenderCommandTyped<RenderCommandType::Dispatch, RenderCommandQueueType::Compute>
{
    RenderResourceHandle pipelineState;
    ShaderArgument shaderArguments[MaxShaderParameters];
    uint32 shaderArgumentsCount = 0;

    uint32 dispatchX = 0;
    uint32 dispatchY = 0;
    uint32 dispatchZ = 0;
};
```

As an example, here is a compute dispatch command, which specifies Compute as the queue type. The underlying backend and scheduler can interpret this accordingly

# Render Command List

- Encodes high level commands
- Tracks queue types encountered
  - Queue mask indicating scheduling rules
- Commands are stateless - parallel recording

The high level commands are encoded into a render command list. As each command is encoded, a queue mask is updated which helps indicate the scheduling rules and restrictions for that command list.

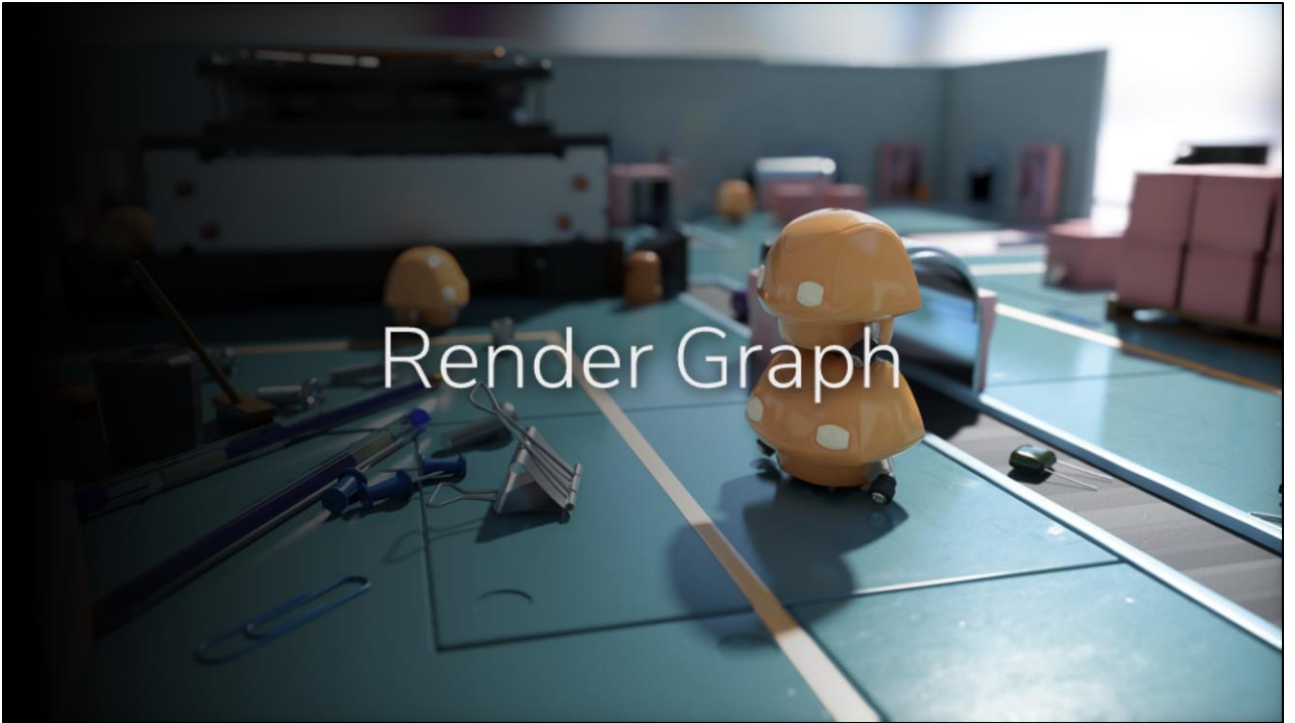
The commands are stateless, which allows for fully parallel recording.

# Render Compilation

- Render command lists are “compiled”
  - Translation to low level API
  - Can compile once, submit multiple times
- Serial operation (memcpy speed)
  - Perfect redundant state filtering

The render command lists are compiled by each render backend, which means the high level commands are translated to the low level API. Command lists can be compiled once, and submitted multiple times, in appropriate cases.

The low level translation is a serial operation by design, as this is basically running at memcpy speed. Doing this part serial means we get perfect redundant state filtering.



Another significant rendering component is our render graph



# Render Graph

- Inspired by FrameGraph [O'Donnell 2017]
- Automatically handle transient resources
- Import explicitly managed resources
- Automatic resource transitions
  - Render target batching
  - DiscardResource
  - Memory aliasing barriers
  - ...

Render graph is inspired by Frostbite's Frame Graph

Just like Frame Graph, we automatically handle transient resources. Complex rendering uses a number of temporary buffers and textures, that stay alive just for the purpose and duration of an algorithm. These are called transient resources, and the graph automatically manages these lifetimes efficiently.

Automatic resource transitions are also performed

# Render Graph

- Basic memory management
  - Not targeting current consoles
  - Fine grained memory reuse sub-optimal with current PC drivers
    - Lose ~5% on aliasing barriers and discards
- Automatic queue scheduling
  - Ongoing research
  - Need heuristics on task duration and bottlenecks
    - e.g. Memory vs ALU
    - Not enough to specify dependencies

Compared to Frame Graph, we opted for a simpler memory management model. We are not targeting current consoles with Halcyon, so we don't fine grained memory reuse. Current PC drivers and APIs are not as efficient in this area (for a variety of reasons), and you lose ~5% on aliasing barriers and discards.

In render graph, we support automatic queue scheduling (graphics, copy, compute, etc..). This is an area of ongoing research, as it's not enough to just specify input and output dependencies. You also need heuristics on task duration and bottlenecks to further improve the scheduling.

# Render Graph

- **Frame Graph** → **Render Graph**: No concept of a “frame”
- Fully **automatic transitions and split barriers**
- Single implementation, regardless of backend
  - Translation from high level render command stream
  - API differences hidden from render graph
- **Support for mGPU**
  - Mostly implicit and automatic
  - Can specify a scheduling policy
  - Not discussed in this presentation

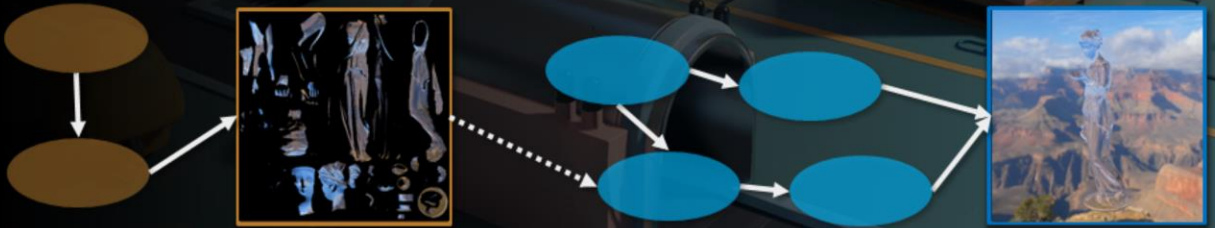
We call our implementation Render Graph, because we don't have the concept of a “frame”.

Our transitions and split barriers are fully automatic, and there is a single implementation, regardless of backend. This is thanks to our high level render command stream, which hides API differences from render graph.

We also support multi-gpu, which is mostly implicit and automatic, with the exception of a scheduling policy you configure on your pass – this represents what devices a pass runs on, and what transfers are performed going in or out of the pass.

# Render Graph

- Composition of multiple graphs at varying frequencies
  - Same GPU: async compute
  - mGPU: graphs per GPU
  - Out-of-core: server cluster, remote streaming



Render graph supports composition of multiple graphs at varying frequencies. These graphs can run on the same GPU - such as async compute. They can run with multi-gpu, with a graphs on each GPU. And they can even run out of core, such as in a server cluster, or on another machine streamed remotely.

# Render Graph

- Composition of multiple graphs at varying frequencies
  - e.g. translucency, refraction, global illumination



There are a number of techniques that can easily run at a different frequency from the main graph, such as object space translucency and reflection, or our surfel based global illumination.

# Render Graph

- Two phases
- **Graph construction**
  - Specify inputs and outputs
  - Serial operation (by design)
- **Graph evaluation**
  - Highly parallelized
  - Record high level render commands
  - Automatic barriers and transitions

Render graph runs in two phases, construction and evaluation.

Construction specifies the input and output dependencies, and this is a serial operation by design.

If you implement a hierarchical gaussian blur as a single pass that gets chained X times, you want to read the size of the input in each step, and generate the proper sized output. Maybe you could split it up into some threads, but tracking dependencies in order to do it in a parallel fashion might be more costly than actually running it serially.

Evaluation is highly parallelized, and this is where high level render commands are recorded, and automatic barriers and transitions are inserted.

```

renderGraph.addPassCallback("Present Pass", [&](RenderGraphBuild& build)
{
    auto& outputTexData = scope.get<RenderGraphOutputTexture>();
    const auto& viewData = scope.get<RenderGraphViewData>();
    const auto& custom = scope.getOptional<RenderGraphCustomFinalTexture>();

    auto finalTexture = build.read(
        custom ? custom->finalTexture
        : scope.get<RenderGraphFinalTexture>().finalTexture, RenderBindFlags::ShaderResource);

    auto outputTexture = outputTexData.outputTexture = build.write(outputTexData.outputTexture, RenderBindFlags::UnorderedAccess);

    return [=](RenderGraphRegistry& registry, RenderCommandList& commandList)
    {
        RenderPoint point = {};
        RenderBox box = {};
        box.w = viewData.viewWidth;
        box.h = viewData.viewHeight;

        commandList.copyTexture(
            registry.getTexture(outputTexture),
            0,
            point,
            registry.getTexture(finalTexture),
            0,
            box);
    };
});

```

← Construction phase

← Evaluation phase

Here is an example render graph pass, with the construction phase up top, and the evaluation phase at the bottom.

SEED // Halcyon + Vulkan

# Render Graph

- Automatic profiling data
- GPU and CPU counters per-pass
- Works with mGPU
  - Each GPU is profiled



Render graph can collect automatic profiling data, presenting you with GPU and CPU counters per-pass

Additionally, this works with mGPU, where each GPU is shown in the list



# Render Graph

- Live debugging overlay
- Evaluated passes in-order of execution
- Input and output dependencies
- Resource version information

There is also a live debugging overlay, using ImGui. This overlay shows the evaluated render graph passes in-order of execution, the input and output dependencies, and the resource version information.

Halcyon Browser - SGPU - NVIDIA TITAN V

File Window Scene Display View Camera Debug GPU  
View Mode: Default  
Display Mode: Full

Render Graph

Pass: Reflection Filter Pass		
Reflection Filter Uniforms (version:0)	Inputs	Reflection Filter Uniforms (version:1)
Reflection Temporal 0 (version:0)		Reflection Temporal 0 (version:1)
Reflection Temporal 1 (version:0)		Reflection Meta Temporal 0 (version:1)
Reflection Meta Temporal 0 (version:0)		Reflection Output Texture (version:1)
Reflection Meta Temporal 1 (version:0)		Reflection Meta Output Texture (version:1)
Reflection Hit0 Texture (version:2)		Reflection Variance Output Texture (version:1)
Reflection Hit1 Texture (version:2)		Reflection Velocity Texture (version:1)
Reflection Output Texture (version:0)		
Pass: Reflection Filter 2 Pass		
Reflection Filter Uniforms (version:0)	Inputs	Reflection Filter Uniforms (version:1)
Reflection Output Texture (version:1)		Reflection Output Texture (version:1)
Reflection Meta Output Texture (version:1)		
Reflection Variance Output Texture (version:1)		
Reflection Output Texture (version:0)		
zBuffer History 0 (version:2)		
Depth (version:2)		
Reflection Texture (version:0)		
Pass: Reflection Merge Pass		
Reflection Output Texture (version:1)	Inputs	Lighting (version:2)
Lighting (version:1)		Outputs
Pass: Temporal AA Pass		
Lighting (version:2)	Inputs	TAA History Texture 1 (version:1)
TAA History Texture 0 (version:0)		TAA Output Texture (version:1)
Temporal Reprojection Texture (version:1)		
TAA History Texture 1 (version:0)		
TAA Output Texture (version:0)		
Pass: Velocity reduce		
Velocity Texture 0 (version:2)	Inputs	Reduced velocity texture (version:1)
Reduced velocity texture (version:0)		Outputs
Pass: Velocity reduce		
Reduced velocity texture (version:1)	Inputs	Reduced velocity texture (version:1)
Reduced velocity texture (version:0)		Outputs
Pass: Velocity dilate		
Reduced velocity texture (version:1)	Inputs	Dilated velocity texture (version:1)
Dilated velocity texture (version:0)		Outputs
Pass: Motion blur		
Depth (version:2)	Inputs	Motion blurred image (version:1)
Velocity Texture 0 (version:2)		Outputs
Dilated velocity texture (version:1)		
TAA Output Texture (version:1)		
Motion blurred image (version:0)		
Pass: Bloom Extract Pass		

# Render Graph

Some of our render graph passes:

- Bloom
- BottomLevelUpdate
- BrdfLut
- CocDerive
- DepthPyramid
- DiffuseSh
- Dof
- Final
- GBuffer
- Gtao
- IblReflection
- ImGui
- InstanceTransforms
- Lighting
- MotionBlur
- Present
- RayTracing
- RayTracingAccum
- ReflectionFilter
- ReflectionSample
- ReflectionTrace
- Rtao
- Screenshot
- Segmentation
- ShaderTableUpdate
- ShadowFilter
- ShadowMask
- ShadowCascades
- ShadowTrace
- Skinning
- Ssr
- SurfelGapFill
- SurfelLighting
- SurfelPositions
- SurfelSpawn
- Svgf
- TemporalAa
- TemporalReproject
- TopLevelUpdate
- TranslucencyTrace
- Velocity
- Visibility

This is a list of some of our render graph passes in PICA PICA



Shaders are also an important component

# Shaders

- **Complex materials**
  - Multiple microfacet layers
  - [Stachowiak 2018]
- Energy conserving
  - Automatic Fresnel between layers
- **All lighting & rendering modes**
  - Raster, path-traced reference, hybrid
- **Iterate with different looks**
  - Bake down permutations for production



Objects with Multi-Layered Materials

We implemented a system which supports multiple microfacet layers arranged into stacks. The stacks could also be probabilistically blended to support prefiltering of material mixtures. This allowed us to rapidly experiment with the look of our demo, while at the same time enforcing energy conservation, and avoiding common gamedev hacks like “metalness”. An in-engine material editor meant that we could quickly jump in and change the look of everything.

The material system works with all our render modes, be that pure rasterization, path tracing, or hybrid.

For performance, we can bake down certain permutations for production.

# Shaders

- Exclusively HLSL
  - Shader Model 6.X
- Majority are compute shaders
- Performance is critical
  - Group shared memory
  - Wave-ops / Sub-groups

We exclusively use HLSL 6 as a source language, and the majority of our shaders are compute

Performance is critical, so we rely heavily on group shared memory, and wave operations

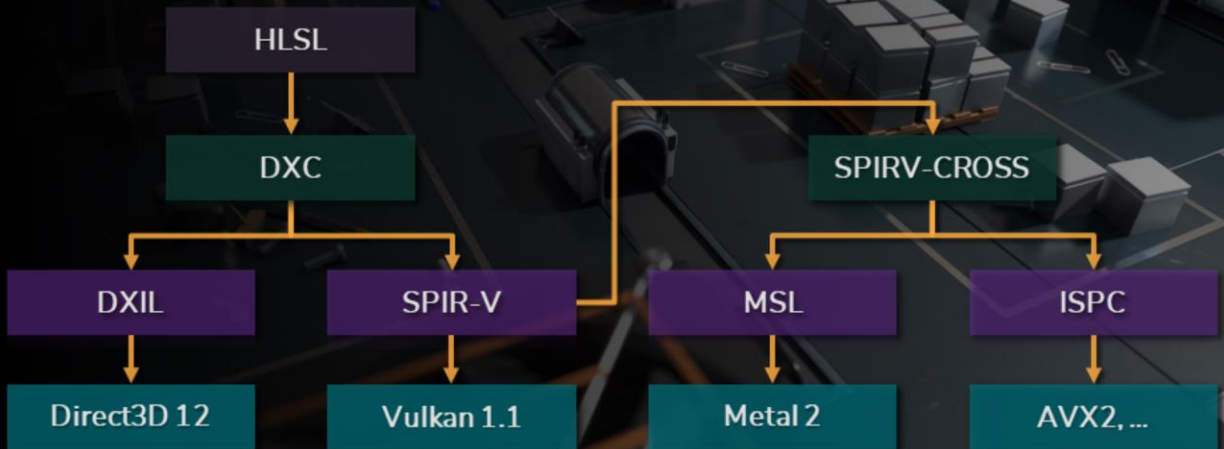
# Shaders

- No reflection
  - Avoid costly lookups
  - Only explicit bindings
  - ... except for validation
- Extensive use of HLSL spaces
  - Updates at varying frequency
- Bindless

We don't rely on any reflection information, outside of validation. We avoid costly lookups by requiring explicit bindings.

We also make extensive use of HLSL spaces, where the spaces can be updated at varying frequency.

# Shaders



Here is a flow graph of our shader compilation. We always start with HLSL compiled with the DXC shader compiler. The DXIL is used by Direct3D 12, and the SPIR-V is used by Vulkan 1.1. We also have support for taking the SPIR-V, running it through SPIRV-CROSS, and generating MSL for Metal, or ISPC to run on the CPU.



# Shader Arguments

- Commands refer to resources with “Shader Arguments”
  - Each argument represents an HLSL space
  - MaxShaderParameters → 4 [Configurable]
  - # of spaces, not # of resources

```
struct RenderCommandDispatch : RenderCommandTyped<RenderCommandType::Dispatch, RenderCommandQueueType::Compute>
{
    RenderResourceHandle pipelineState;
    ShaderArgument shaderArguments[MaxShaderParameters]; ←
    uint32 shaderArgumentsCount = 0;

    uint32 dispatchX = 0;
    uint32 dispatchY = 0;
    uint32 dispatchZ = 0;
};
```

We have a concept in Halcyon called shader arguments, where each argument represents an HLSL space. We limit the maximum number of arguments to 4 for efficiency, but this can be configured. It is important to note that this limit represents the maximum number of spaces, not the maximum number of resources.

# Shader Arguments

- Each argument contains:
  - “ShaderViews” handle
  - Constant buffer handle and offset
- “ShaderViews”
  - Collection of SRV and UAV handles

Each shader argument contains a “ShaderViews” handle, which refers to a collection of SRV and UAV handles. Additionally, each shader argument also contains a constant buffer handle and offset into the buffer.

# Shader Arguments

- Constant buffers are **all dynamic**
  - Avoid temporary descriptors
  - Just a few large buffers, offsets change frequently
  - **VK\_DESCRIPTOR\_TYPE\_UNIFORM\_BUFFER\_DYNAMIC**
  - **DX12 Root Descriptors** (pass in GPU VA)
- **All descriptor sets are only written once**
  - Persisted / cached

Our constant buffers are all dynamic, and we avoid having temporary descriptors. We have just a few large buffers, and offsets into these buffers change frequently. For Vulkan, we use the uniform buffer dynamic descriptor type, and on Dx12 we use root descriptors, just passing in a GPU virtual address.

All our descriptor sets are only written once, then persisted or cached.

```

Texture2D<float4> g_albedo           : register(t0, space0);
Texture2D<float4> g_normal          : register(t1, space0);
Texture2D<float4> g_roughness       : register(t2, space0);
Texture2D<float4> g_metalness       : register(t3, space0);
Texture2D<float4> g_ao              : register(t4, space0);
Texture2D<float4> g_emissive        : register(t5, space0);
Texture2DArray<float4> g_translucency : register(t6, space0);

StructuredBuffer<uint3> g_indexBuffer : register(t0, space1);
StructuredBuffer<VtxInputPosition> g_vbPositions : register(t1, space1);
StructuredBuffer<VtxInputTangentSpace> g_vbTangentSpace : register(t2, space1);
StructuredBuffer<VtxInputTexCoordAndColor> g_vbTexAndColor : register(t3, space1);

ConstantBuffer<GeometryConstants> g_geometry : register(b0, space1);
ConstantBuffer<MaterialConstants> g_material : register(b0, space0);

```

```

{
    // space0
    ShaderArgument(materialConstants.buffer, materialConstantsOffset, materialShaderViews),
    // space1
    ShaderArgument(geometryConstants.buffer, geometryConstantsOffset, geometryShaderViews)
},

```

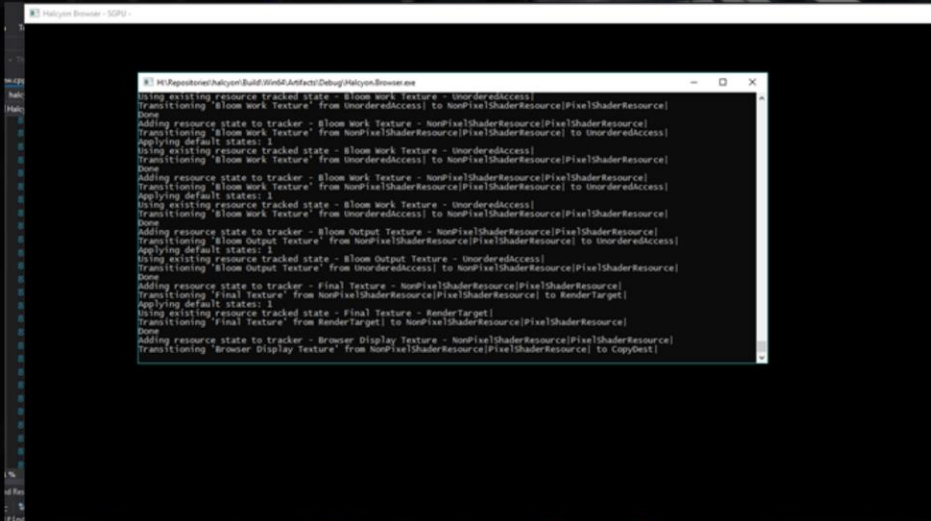
Here is an example HLSL snippet showing one usage of spaces. Each space contains a collection of SRVs and a constant buffer. The shader argument configuration on the CPU is shown below.

# Vulkan Implementation

- Architecture simplified development effort
- Vulkan specific:
  - Backend and device implementation
  - Memory allocators (e.g. AMD VMA)
  - Barrier and transition logic
  - Resource binding model

Our architecture simplified the development effort, for Vulkan we just needed a new backend and device implementation, API specific memory allocators, barrier and transition logic, and a resource binding model that aligned with our shader compilation.

# Vulkan Implementation



```

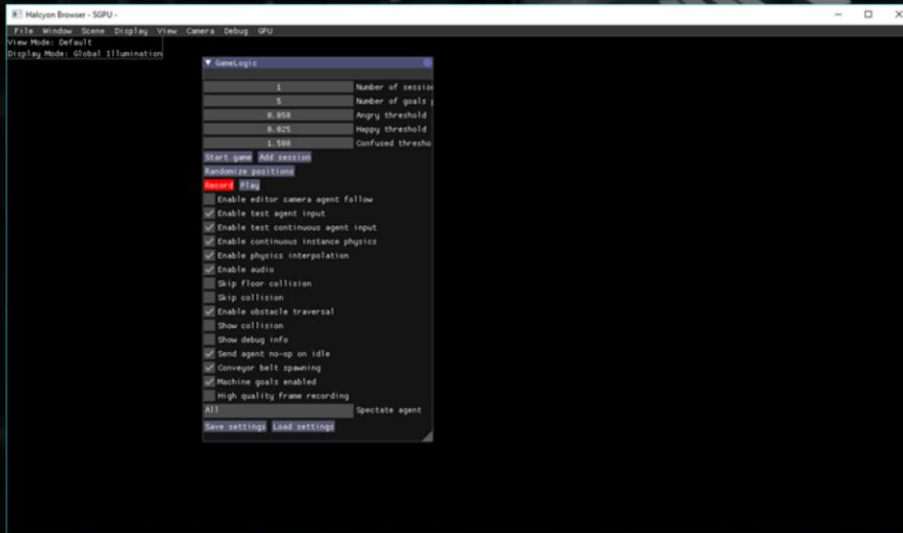
M:\Repositories\Halcyon\Build\Win64\Artifacts\Debug\Halcyon.Browser.exe
Using existing resource tracked state - Bloom work Texture - UnorderedAccess
Transitioning 'Bloom work Texture' from UnorderedAccess to NonPixelShaderResource[PixelShaderResource]
Done
Adding resource state to tracker - Bloom work Texture - NonPixelShaderResource[PixelShaderResource]
Transitioning 'Bloom work Texture' from NonPixelShaderResource[PixelShaderResource] to UnorderedAccess
Applying default states: 1
Using existing resource tracked state - Bloom work Texture - UnorderedAccess
Transitioning 'Bloom work Texture' from UnorderedAccess to NonPixelShaderResource[PixelShaderResource]
Done
Adding resource state to tracker - Bloom work Texture - NonPixelShaderResource[PixelShaderResource]
Transitioning 'Bloom work Texture' from NonPixelShaderResource[PixelShaderResource] to UnorderedAccess
Applying default states: 1
Using existing resource tracked state - Bloom work Texture - UnorderedAccess
Transitioning 'Bloom work Texture' from UnorderedAccess to NonPixelShaderResource[PixelShaderResource]
Done
Adding resource state to tracker - Bloom Output Texture - NonPixelShaderResource[PixelShaderResource]
Transitioning 'Bloom Output Texture' from NonPixelShaderResource[PixelShaderResource] to UnorderedAccess
Applying default states: 1
Using existing resource tracked state - Bloom Output Texture - UnorderedAccess
Transitioning 'Bloom Output Texture' from UnorderedAccess to NonPixelShaderResource[PixelShaderResource]
Done
Adding resource state to tracker - Final Texture - NonPixelShaderResource[PixelShaderResource]
Transitioning 'Final Texture' from NonPixelShaderResource[PixelShaderResource] to RenderTarget
Applying default states: 1
Using existing resource tracked state - Final Texture - RenderTarget
Transitioning 'Final Texture' from RenderTarget to NonPixelShaderResource[PixelShaderResource]
Done
Adding resource state to tracker - Browser Display Texture - NonPixelShaderResource[PixelShaderResource]
Transitioning 'Browser Display Texture' from NonPixelShaderResource[PixelShaderResource] to CopyDest

```

The first stage was to get command translation working, and performing the correct barriers and transitions. This was done using the Vulkan validation layers, and lots of glorious printf debugging

SEED // Halcyon + Vulkan

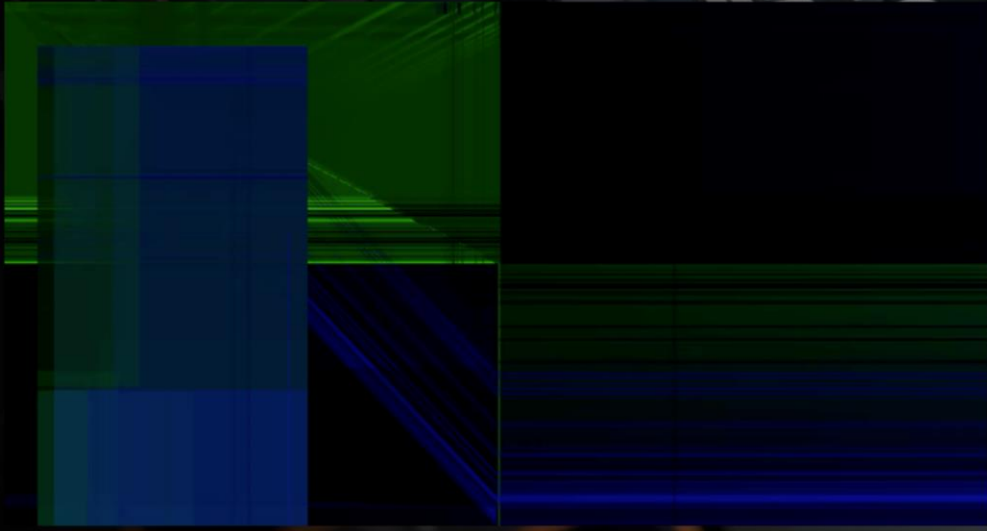
# Vulkan Implementation



The second stage was getting basic ImGui, resource creation, and swap chain flip working. This meant I could easily toggle any display mode or render settings while debugging.

SEED // Halcyon + Vulkan

# Vulkan Implementation

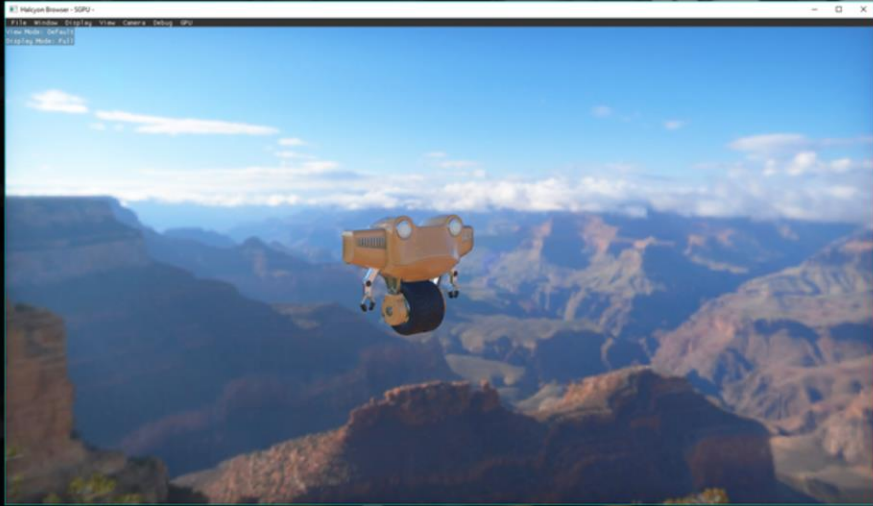


Naturally, fun bugs occurred



SEED // Halcyon + Vulkan

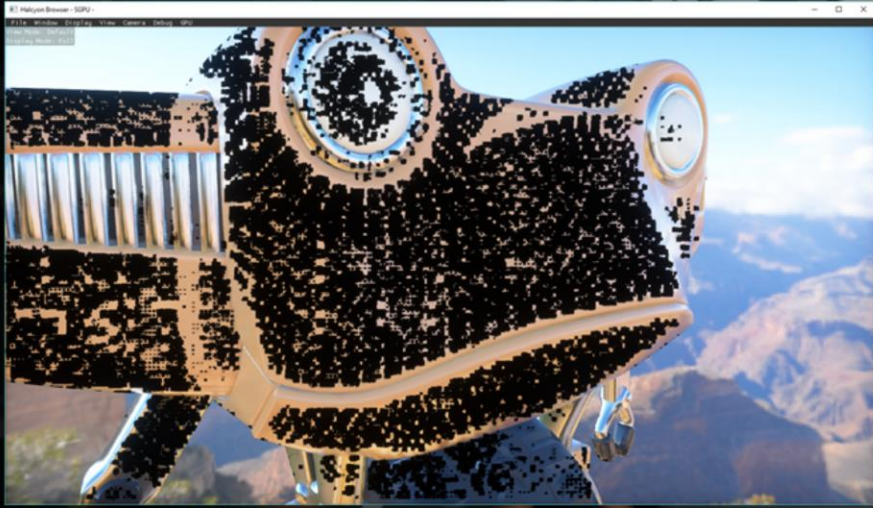
# Vulkan Implementation



The third stage was to bring up more of the Halcyon asset loading operations, and get a basic entry point running

SEED // Halcyon + Vulkan

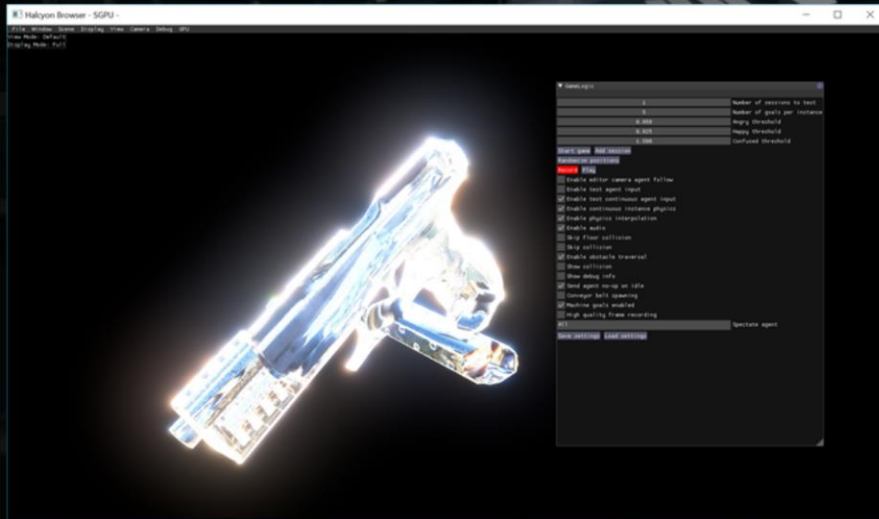
# Vulkan Implementation



Plenty of fun bugs with this, as well

SEED // Halcyon + Vulkan

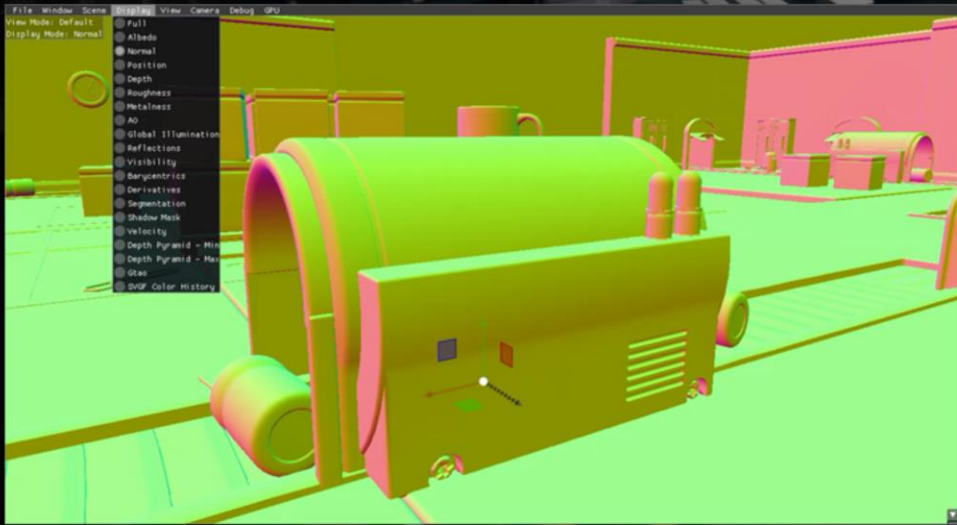
# Vulkan Implementation



Plenty of fun bugs with this, as well

SEED // Halcyon + Vulkan

# Vulkan Implementation



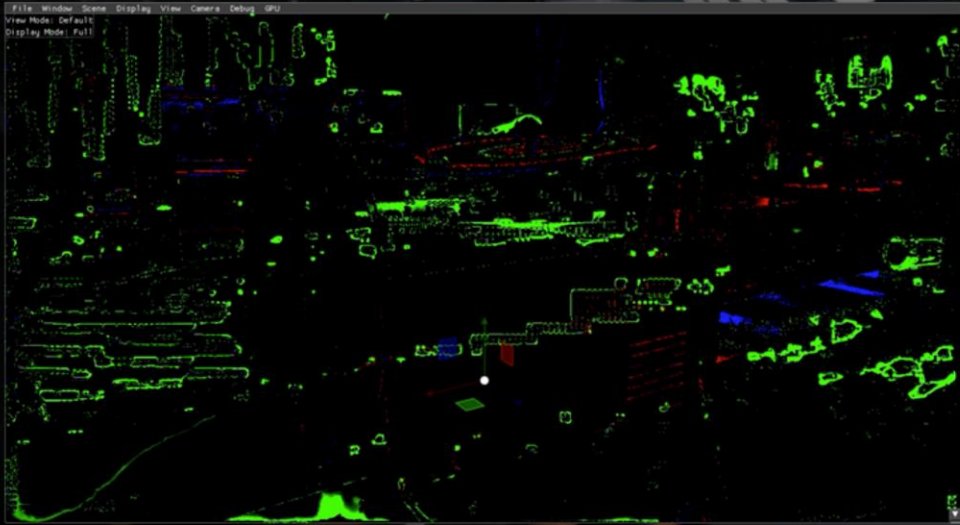
The final stage was to bring up absolutely everything else, including render graph!

To simplify things, I worked on getting just the normals and albedo display modes working.

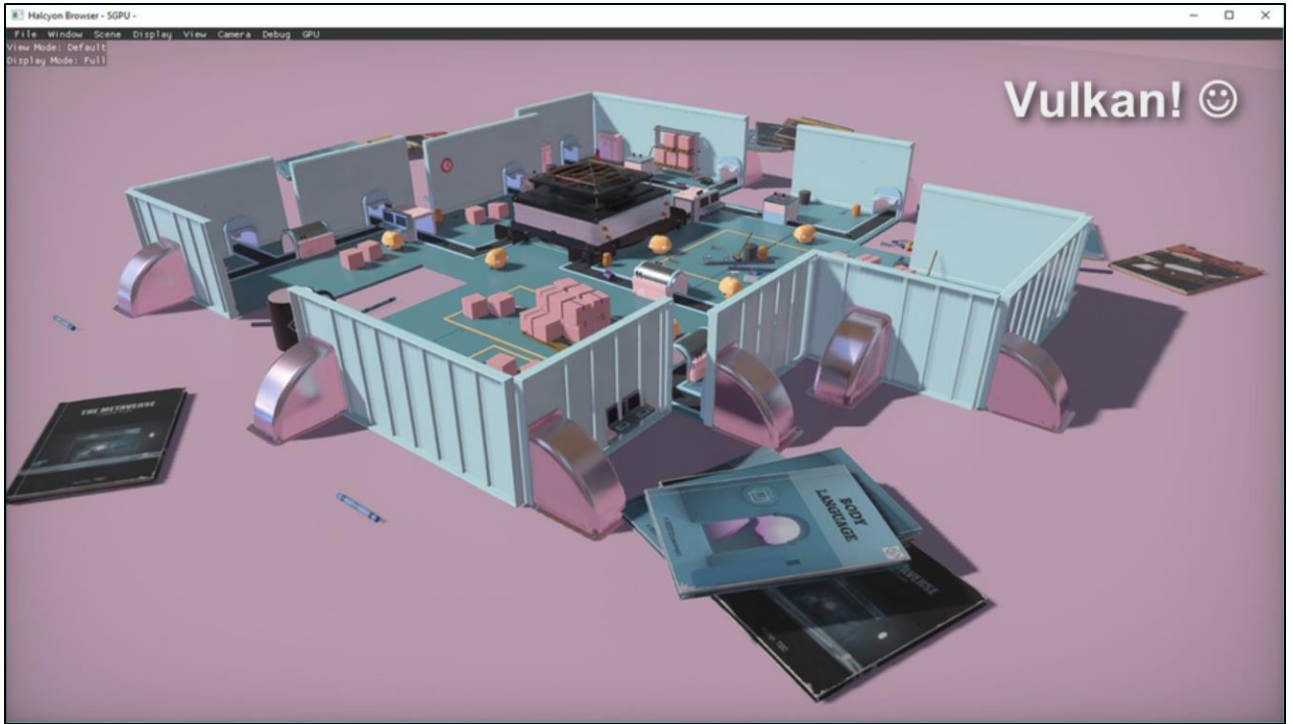
I relied on the fact that render graph will cull any passes not contributing to the final result, so I could easily remove problematic passes from running while I get the basics working.

SEED // Halcyon + Vulkan

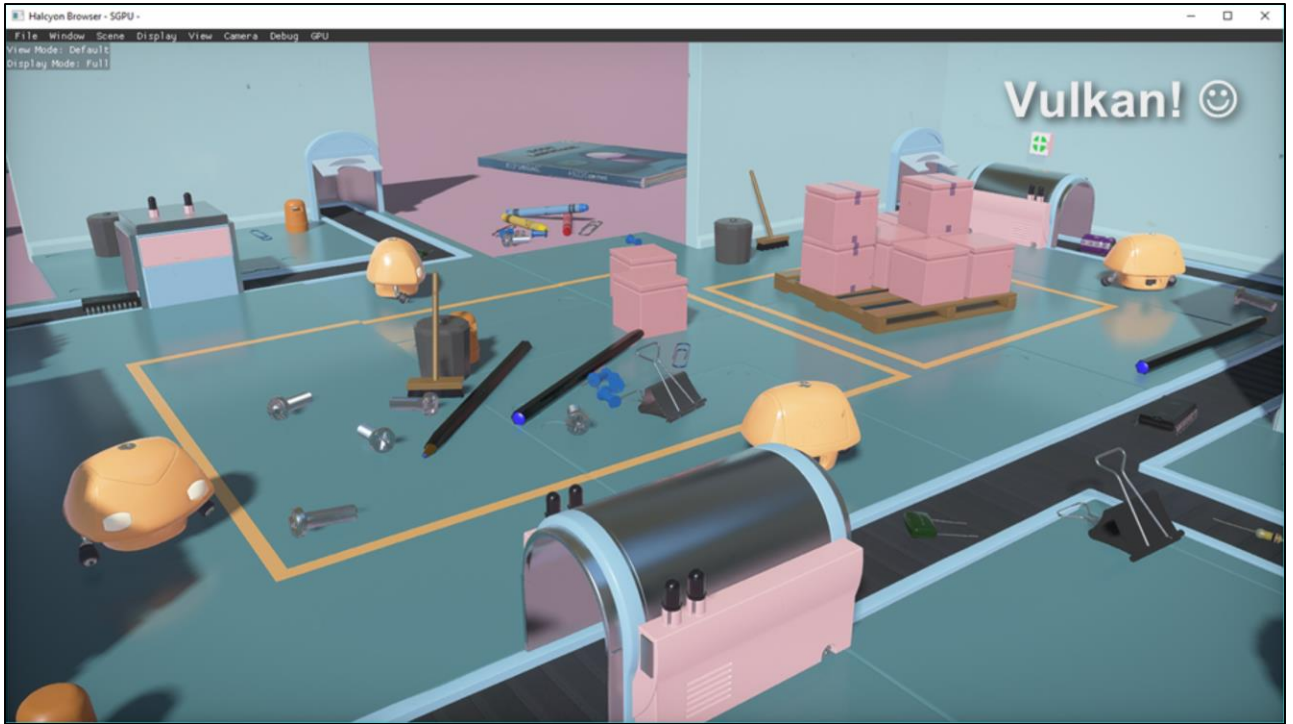
# Vulkan Implementation



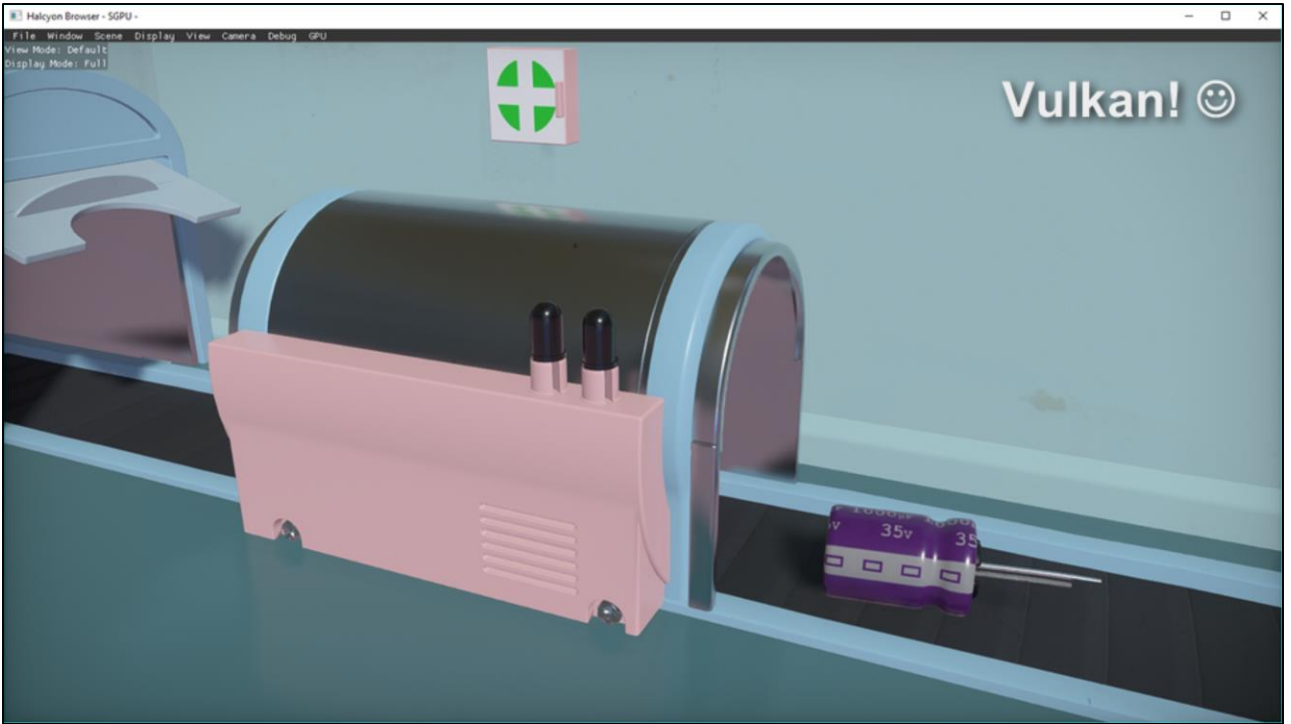
With that working, I started to bring up the more complicated passes. As expected, there were plenty of fun issues to sort through.



Eventually, everything worked!



Eventually, everything worked!



Eventually, everything worked!



# Vulkan Implementation

- Shader compilation (HLSL → SPIR-V)
  - Patch SPIR-V to match DX12
  - Using **spirv-reflect** from Hai and Cort
- `spvReflectCreateShaderModule`
- **`spvReflectEnumerateDescriptorSets`**
- **`spvReflectChangeDescriptorBindingNumbers`**
- `spvReflectGetCodeSize / spvReflectGetCode`
- `spvReflectDestroyShaderModule`

An important part of our Vulkan backend, was consuming HLSL as a source language, and fixing up the SPIR-V to behave the same as our Dx12 resource binding model.

We decided to use the spirv-reflect library from Hai and Cort; it does a great job at providing SPIR-V reflection using DX12 terminology, but we use it exclusively to patch up our descriptor binding numbers.

# SPIR-V Patching

- **SPV\_REFLECT\_RESOURCE\_FLAG\_SRV**
  - Offset += 1000
- **SPV\_REFLECT\_RESOURCE\_FLAG\_SAMPLER**
  - Offset += 2000
- **SPV\_REFLECT\_RESOURCE\_FLAG\_UAV**
  - Offset += 3000

SRVs, Samplers, and UAVs are simple. These types are uniquely namespaced in DX12, so t0 and s0 wouldn't collide. This is not the case in SPIR-V, so we apply a simple offset to each type to emulate this behavior.

# SPIR-V Patching

- **SPV\_REFLECT\_RESOURCE\_FLAG\_CBV**
  - Offset Unchanged: 0
  - Descriptor Set += **MAX\_SHADER\_ARGUMENTS**
- CBVs move to their own descriptor sets
  - ShaderViews become persistent and immutable

Constant or uniform buffers are a bit more interesting. We want to move CBVs to their own descriptor sets, in order to make our ShaderViews representing the other resource types persistent and immutable.

To do so, we don't adjust the offset, as we'll have a single CBV per descriptor set. However, we do shift the descriptor set number by the max number of shader arguments.

This means if descriptor set 0 contained a constant buffer, that constant buffer would move to descriptor set 5 (if max shader arguments is 4).

# SPIR-V Patching

- If 2 of 4 HLSL spaces in use:

Set 0	SRVs ( $\geq 1000$ )	Samplers ( $\geq 2000$ )	UAVs ( $\geq 3000$ )
Set 1	SRVs ( $\geq 1000$ )	Samplers ( $\geq 2000$ )	UAVs ( $\geq 3000$ )
Set 2	Unbound		
Set 3	Unbound		
Set 4	Dynamic Constant Buffer (Offset: 0)		
Set 5	Dynamic Constant Buffer (Offset: 0)		

If a dispatch or draw is using 2 HLSL spaces, the patched SPIR-V will require the following descriptor set layout. Notice the shifted offsets for the SRVs, Samplers, and UAVs, and how the dynamic constant buffers have been hoisted out to their own descriptor sets.

# Vulkan Implementation

- Translate **commands**
  - Read command list
  - Write Vulkan API

```
#define COMPILE_PACKET(TYPE_STRUCT)
case TYPE_STRUCT::Type:
    if (!compileCommand(*static_cast<const TYPE_STRUCT*>(command)))
        return false;
        break

for (const auto* command : recorded)
{
    switch (command->type)
    {
        COMPILE_PACKET(RenderCommandDraw);
        COMPILE_PACKET(RenderCommandDrawIndirect);
        COMPILE_PACKET(RenderCommandDispatch);
        COMPILE_PACKET(RenderCommandDispatchIndirect);
        COMPILE_PACKET(RenderCommandUpdateBuffer);
        COMPILE_PACKET(RenderCommandUpdateTexture);
        COMPILE_PACKET(RenderCommandCopyBuffer);
        COMPILE_PACKET(RenderCommandCopyTexture);
        COMPILE_PACKET(RenderCommandBarriers);
        COMPILE_PACKET(RenderCommandTransitions);
        COMPILE_PACKET(RenderCommandBeginTiming);
        COMPILE_PACKET(RenderCommandEndTiming);
        COMPILE_PACKET(RenderCommandResolveTimings);
        COMPILE_PACKET(RenderCommandBeginEvent);
        COMPILE_PACKET(RenderCommandEndEvent);
        COMPILE_PACKET(RenderCommandBeginRenderPass);
        COMPILE_PACKET(RenderCommandEndRenderPass);
        break;
    default:
        HcyFail();
        return false;
    }
}

#undef COMPILE_PACKET
```

Another important aspect of a new render backend is the translation from our high level command stream to low level API calls.

SEED // Halcyon + Vulkan

```
bool RenderCompileContextVulkan::compileCommand(const RenderCommandDispatch& command)
{
    const auto pipelineState = device.getComputePipelineState(command.pipelineState);
    if (!applyPipelineState(pipelineState))
    {
        return false;
    }

    applyShaderArguments(command.pipelineState, command.shaderArguments, command.shaderArgumentsCount);
    applyTransitions();

    vkCmdDispatch(commandBuffer, command.dispatchX, command.dispatchY, command.dispatchZ);
    return true;
}
```

Here is an example of translating a high level compute dispatch command to Vulkan.

```
bool RenderCompileContextVulkan::compileCommand(const RenderCommandBeginTiming& command)
{
    auto timingHeap = const_cast<RenderTimingHeapVulkan*>(device.getTimingHeap(command.timingHeap));
    HcyAssert(timingHeap);
    const uint32 slot = 2 * command.region + 0;
    HcyAssert(slot >= 0 && slot < (timingHeap->desc.regionCount * 2));
    auto& buffer = timingHeap->buffers[timingHeap->currentBuffer];
    vkCmdWriteTimestamp(
        commandBuffer,
        VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
        buffer.queryPool,
        slot);
    buffer.regions[command.region].begin = slot;
    return true;
}
```

Here is an example of translating a high level begin timing command to Vulkan timestamps

```
bool RenderCompileContextVulkan::compileCommand(const RenderCommandResolveTimings& command)
{
    auto timingHeap = const_cast<RenderTimingHeapVulkan*>(device.getTimingHeap(command.timingHeap));
    HcyAssert(timingHeap);
    HcyAssert(command.regionCount > 0);
    HcyAssert(command.regionCount <= timingHeap->desc.regionCount);
    auto& buffer = timingHeap->buffers[timingHeap->currentBuffer];
    buffer.resolveStart = buffer.regions[command.regionStart].begin;
    buffer.resolveCount = (command.regionCount * 2);

    vkCmdCopyQueryPoolResults(
        commandBuffer,
        buffer.queryPool,
        buffer.resolveStart,
        buffer.resolveCount,
        buffer.readBack->buffer(),
        buffer.resolveStart * sizeof(uint64),
        sizeof(uint64),
        VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT);

    buffer.writeFence->signalGpu(queue.commandQueue());

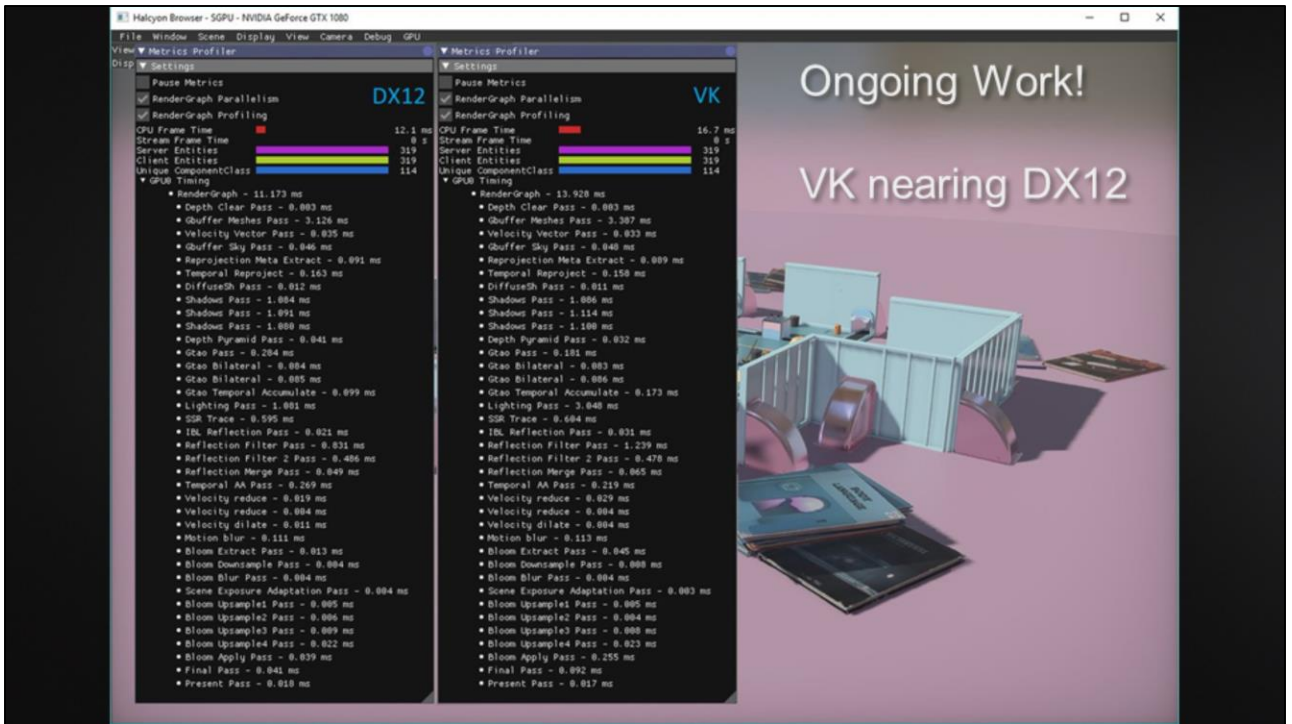
    vkCmdResetQueryPool(commandBuffer, buffer.queryPool, buffer.resolveStart, buffer.resolveCount);

    timingHeap->previousBuffer = timingHeap->currentBuffer;
    timingHeap->currentBuffer = (device.m_frameIndex % timingHeap->buffers.size());

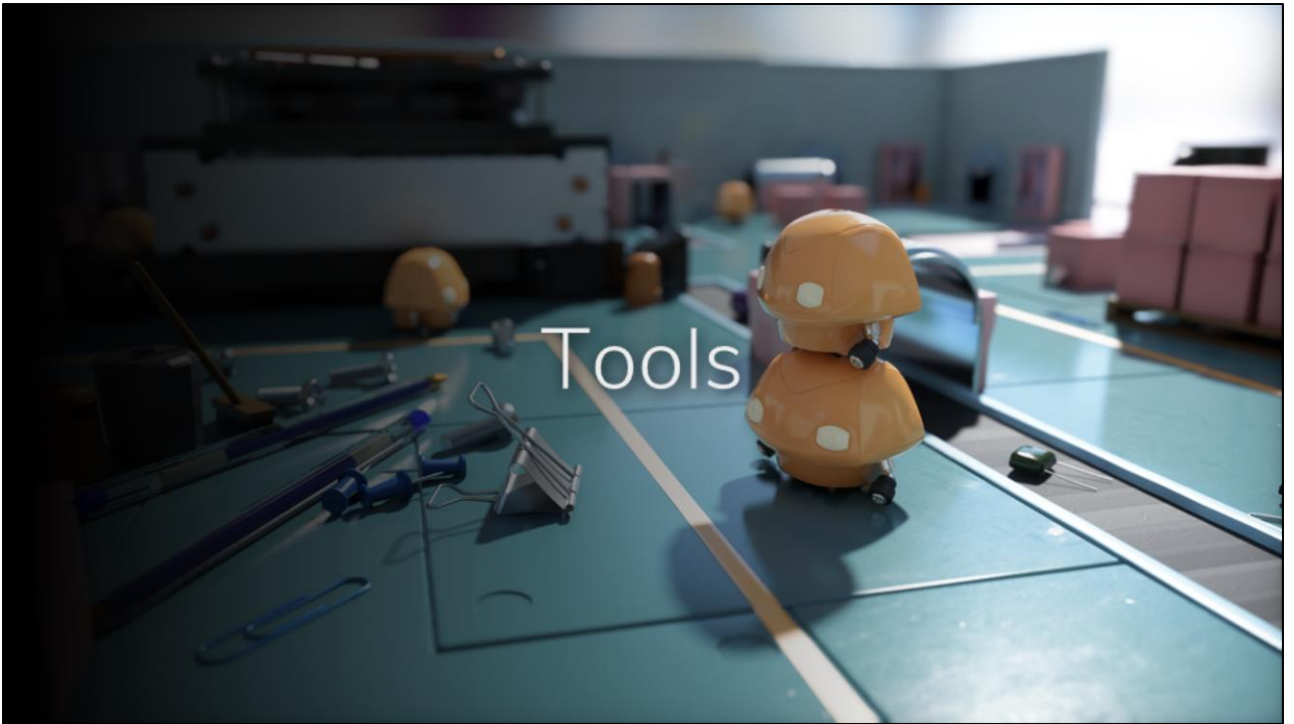
    return true;
}
```

And here is an example of translating a high level resolve timings command to Vulkan. Notice that the complexity behind fence tracking or resetting the query pool is completely hidden from the calling code. Each backend implementation can choose how to handle this efficiently.





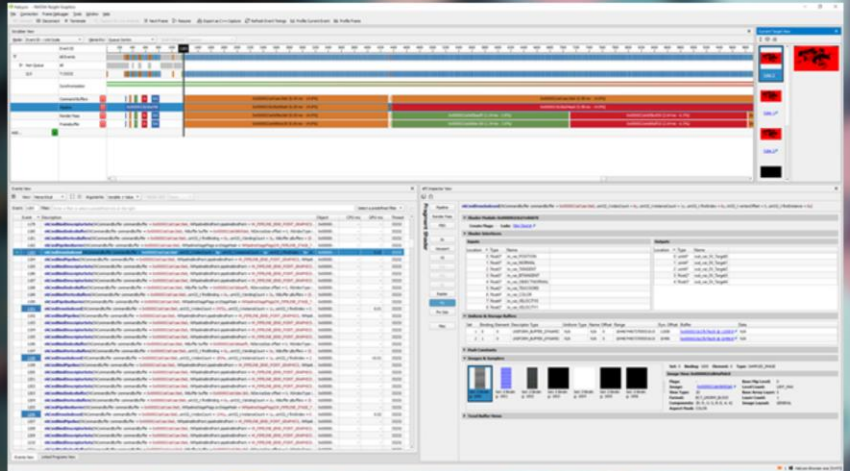
This comparison shows that our Vulkan implementation is nearing the performance of our DirectX 12 version, and is completely usable. There are a number of reasons for the delta, but none of them represent any amount of significant work to resolve.



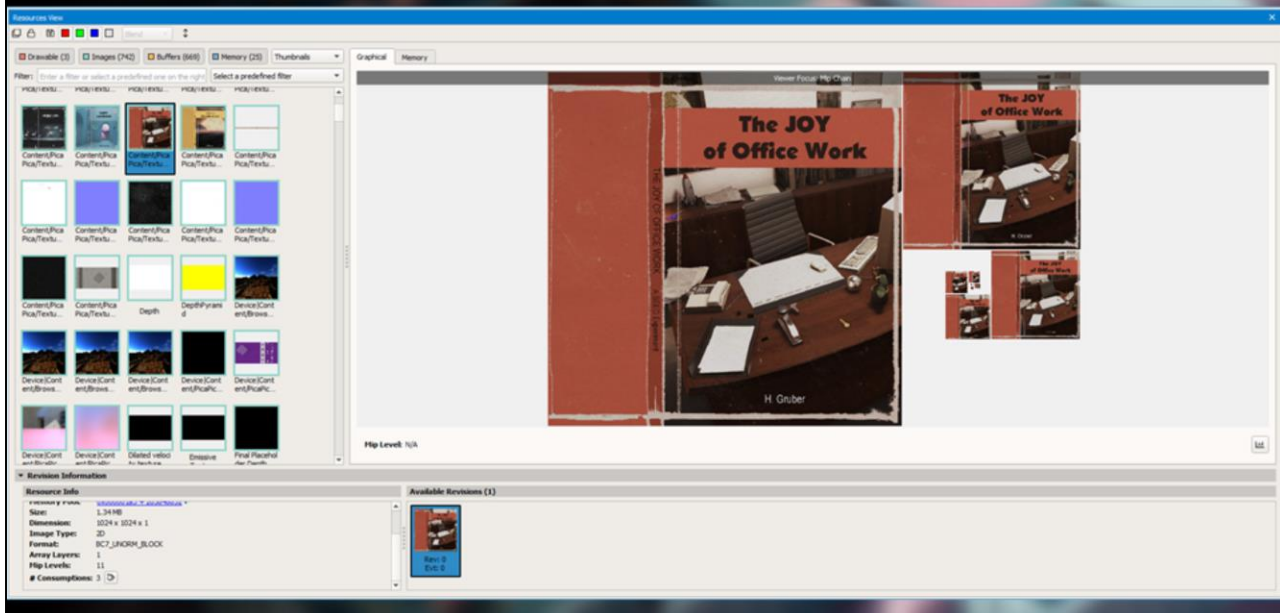
I will briefly mention some useful tools used for the Vulkan implementation

# Tools

- RenderDoc
- NV Nsight
- AMD RGP



For debugging and profiling, the usual suspects were quite helpful, and used extensively.



For debugging and profiling, the usual suspects were quite helpful, and used extensively.

## Tools

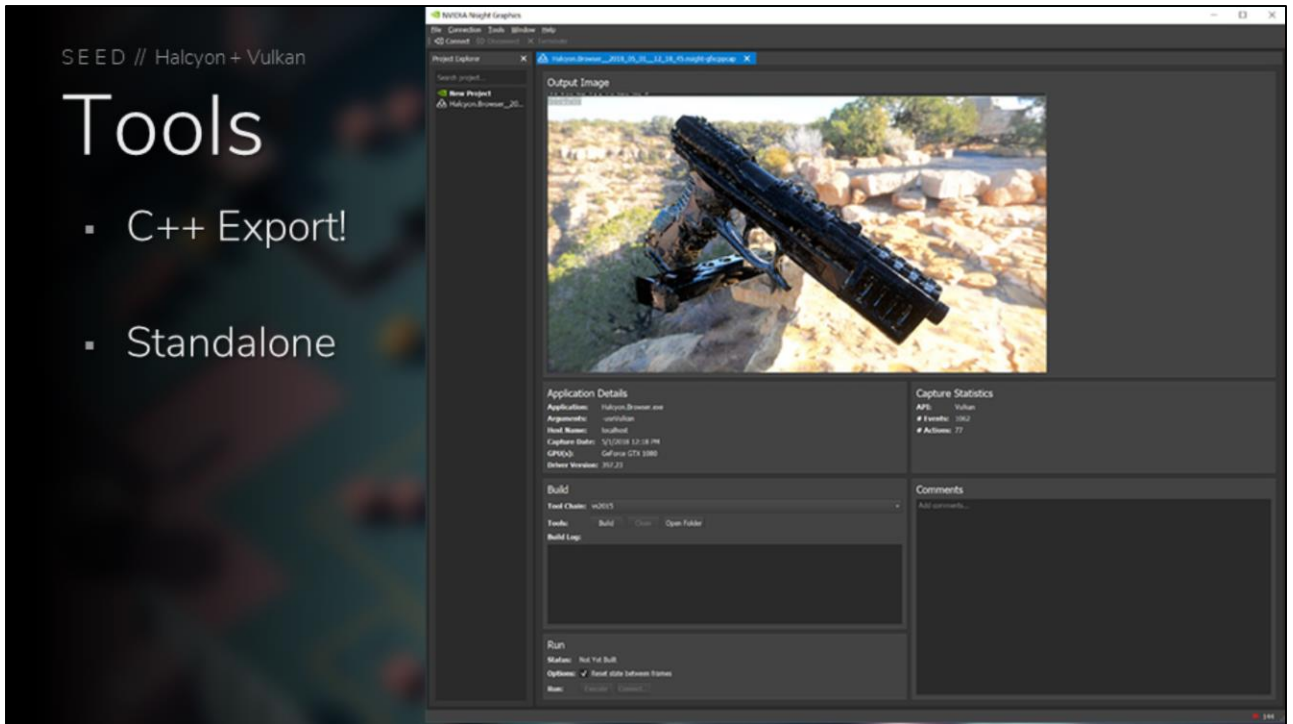
API Statistics View

Summary  
Draws: 1001 Dispatches: 38 Clears: 27 BBs: 1 Presents: 1 Command List Executes: 30 Signals: 0 Waits: 0 Fsync Data Updates: 42 Non-API: 29 Other: 1095 Total: 1204

Details  
Filter: Enter a filter

API Call	Count	Avg CPU ms	I CPU ms	Avg GPU ms	I GPU ms
vkQueueSubmit()	30	0.30	3.00	0.00	0.00
vkCreateBufferView()	4	0.30	0.30	0.00	0.00
vkAllocateDescriptorSets()	41	0.00	2.40	0.00	0.00
vkResetCommandPool()	1	4.00	0.00	0.00	0.00
vkCreateImageView()	104	0.00	3.40	0.00	0.00
vkWaitForFences()	2	<0.01	0.01	0.00	0.00
vkCreateRenderPass()	13	<0.01	0.01	0.00	0.00
vkAcquireNextImageKHR()	1	<0.01	<0.01	0.00	0.00
vkResetFences()	2	<0.01	<0.01	0.00	0.00
vkFlagMemory()	1	<0.01	<0.01	0.00	0.00
vkCreateDescriptorPool()	41	<0.01	0.04	0.00	0.00
vkUpdateDescriptorSets()	41	<0.01	0.04	0.00	0.00
vkCreateFramebuffer()	13	<0.01	<0.01	0.00	0.00
vkImageMemory()	1	<0.01	<0.01	0.00	0.00
vkDestroyRenderPass()	13	<0.01	<0.01	0.00	0.00
vkDestroyFramebuffer()	13	<0.01	<0.01	0.00	0.00
vkDestroyDescriptorPool()	41	<0.01	<0.01	0.00	0.00
vkDestroyImageView()	104	<0.01	<0.01	0.00	0.00
vkCmdBindDescriptorSets()	5,143	0.00	0.00	0.00	0.00
vkCmdBindPipeline()	1,070	0.00	0.00	0.00	0.00
vkCmdBindIndexBuffers()	1,070	0.00	0.00	0.00	0.00
vkCmdBindVertexBuffers()	1,070	0.00	0.00	0.00	0.00
vkCmdDrawIndexed()	1,070	0.00	0.00	<0.01	11.40
vkCmdPipelineBarrier()	614	0.00	0.00	0.00	0.00
vkCmdUpdateBuffer()	38	0.00	0.00	0.00	0.00
vkCmdDispatch()	38	0.00	0.00	0.14	3.10
vkBeginCommandBuffer()	28	0.00	0.00	0.00	0.00
vkEndCommandBuffer()	28	0.00	0.00	0.00	0.00
vkCmdBeginRenderPass()	27	0.00	0.00	0.00	0.00
vkCmdEndRenderPass()	27	0.00	0.00	0.00	0.00
vkCmdSetScissor()	12	0.00	0.00	0.00	0.00
vkCmdSetViewport()	9	0.00	0.00	0.00	0.00
vkDestroyBufferView()	4	0.00	0.00	0.00	0.00
vkCmdDraw()	2	0.00	0.00	3.40	3.20
vkCmdWriteTimestamp()	2	0.00	0.00	0.00	0.00
vkCmdCopyImage()	1	0.00	0.00	0.00	0.00
vkCmdCopyPoolResults()	1	0.00	0.00	0.00	0.00
vkCmdResetQueryPool()	1	0.00	0.00	0.00	0.00
vkCmdBindImage()	1	0.00	0.00	0.00	0.00
vkQueuePresentKHR()	1	0.00	0.00	0.00	0.00

The API statistics view in Nvidia Nsight is a great way to look at the count and cost of each API call.



Another awesome feature with Nsight is the ability to export a capture as a standalone C++ application. Nsight will write out binary blobs of your resources in the capture, and write out source code issues your API calls. You can build this app and debug problems in an isolated solution.



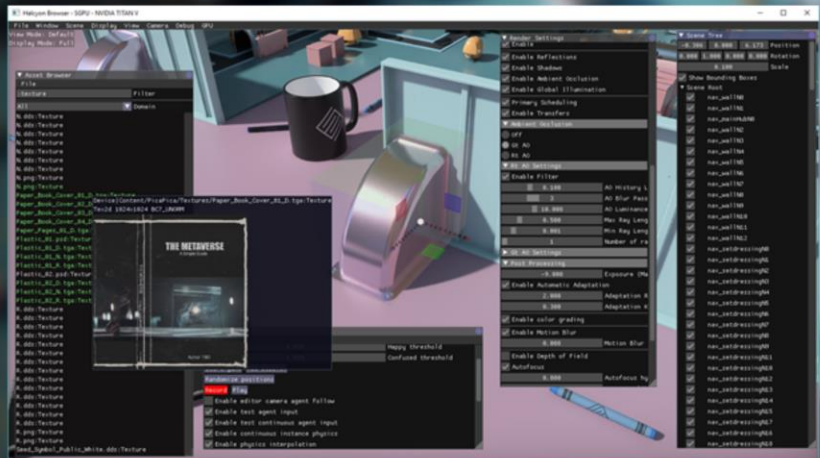


Here is our bug being reproduced in the standalone C++ export

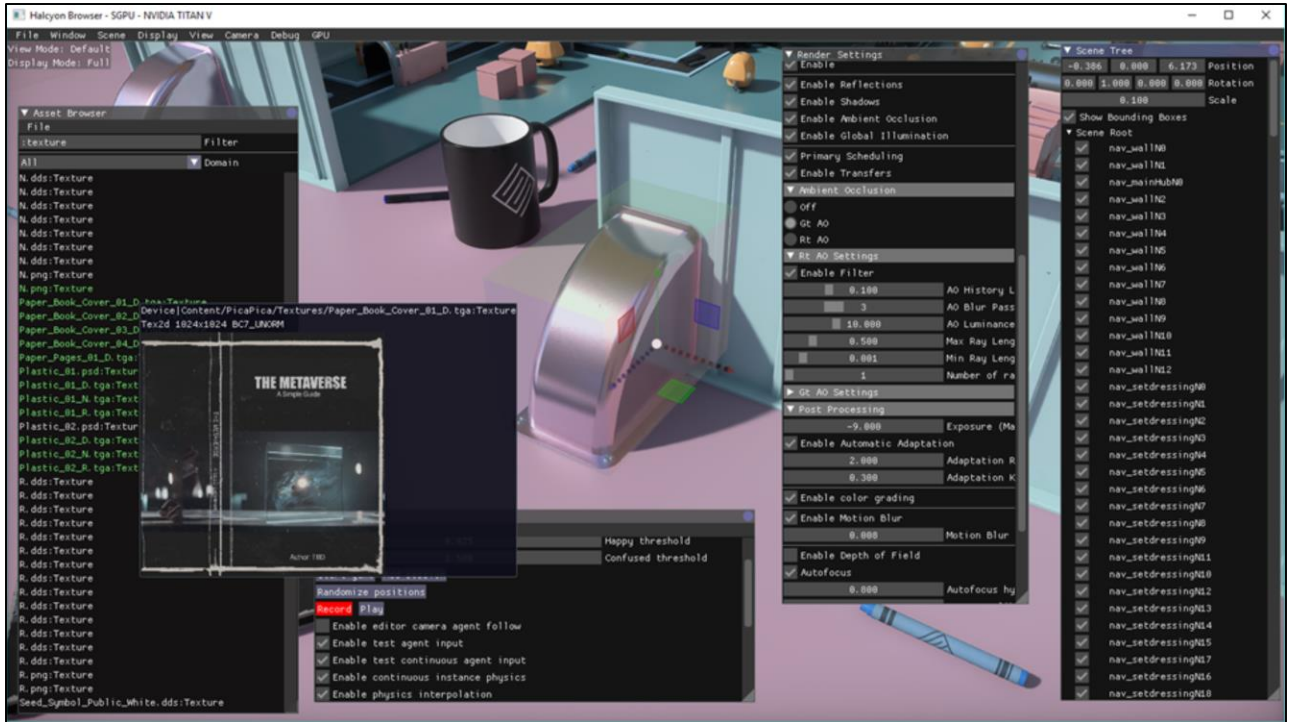


# Dear ImGui + ImGuizmo

- Live tweaking
- **Very useful!**



With our goal of having everything be live-reloadable and live-tweakable, we used DearImGui and ImGuizmo extensively for a number of useful overlays



# References

- **[Stachowiak 2018]** Tomasz Stachowiak. "Towards Effortless Photorealism Through Real-Time Raytracing".  
[available online](#)
- **[Andersson 2018]** Johan Andersson, Colin Barré-Brisebois. "DirectX: Evolving Microsoft's Graphics Platform".  
[available online](#)
- **[Harmer 2018]** Jack Harmer, Linus Gisslén, Henrik Holst, Joakim Bergdahl, Tom Olsson, Kristoffer Sjöo and Magnus Nordin. "Imitation Learning with Concurrent Actions in 3D Games".  
[available online](#)
- **[Opara 2018]** Anastasia Opara. "Creativity of Rules and Patterns".  
[available online](#)
- **[O'Donnell 2017]** Yuriy O'Donnell. "Frame Graph: Extensible Rendering Architecture in Frostbite".  
[available online](#)

# Thanks

- Matthäus Chajdas
- Rys Sommefeldt
- Timothy Lottes
- Tobias Hector
- Neil Henning
- John Kessenich
- Hai Nguyen
- Nuno Subtil
- Adam Sawicki
- Alon Or-bach
- Baldur Karlsson
- Cort Stratton
- Mathias Schott
- Rolando Caloca
- Sebastian Aaltonen
- Hans-Kristian Arntzen
- Yuriy O'Donnell
- Arseny Kapoulkine
- Tex Riddell
- Marcelo Lopez Ruiz
- Lei Zhang
- Greg Roth
- Noah Fredriks
- Qun Lin
- Ehsan Nasiri,
- Steven Perron
- Alan Baker
- Diego Novillo

A special thanks to all these people, that were helpful in our Vulkan journey

# Thanks

## ▪ SEED

- Johan Andersson
- Colin Barré-Brisebois
- Jasper Bekkers
- Joakim Bergdahl
- Ken Brown
- Dean Calver
- Dirk de la Hunt
- Jenna Frisk
- Paul Greveson
- Henrik Halen
- Effeli Holst
- Andrew Lauritzen

- Magnus Nordin
- Niklas Nummelin
- Anastasia Opara
- Kristoffer Sjöo
- Ida Winterhaven
- Tomasz Stachowiak

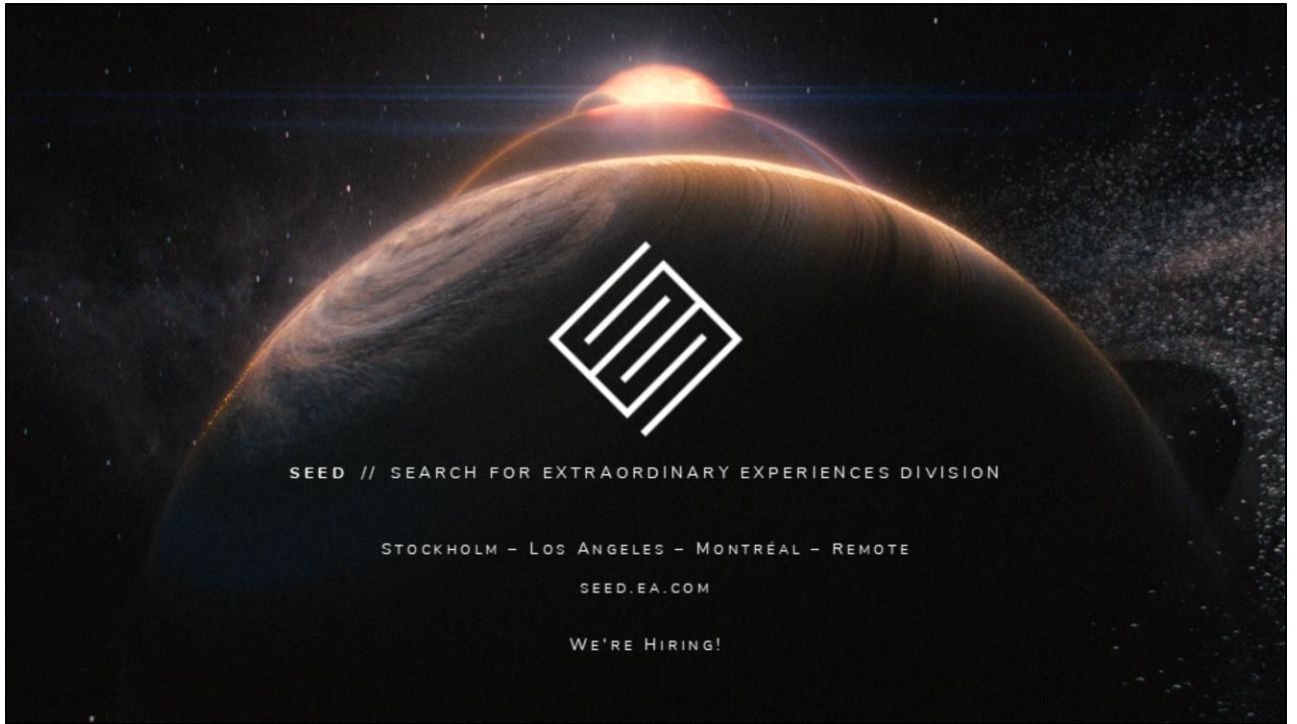
## ▪ Microsoft

- Chas Boyd
- Ivan Nevraev
- Amar Patel
- Matt Sandy

## ▪ NVIDIA

- Tomas Akenine-Möller
- Nir Benty
- Jiho Choi
- Peter Harrison
- Alex Hyder
- Jon Jansen
- Aaron Lefohn
- Ignacio Llamas
- Henry Moreton
- Martin Stich

And before I finish, I would like to thank all the people who contributed to the PICA PICA project. It was a very awesome and dedicated effort by our team, and we could not have done it without our external partners either.



On one last note, I would like to point out that we're hiring for multiple positions at SEED. If you're interested, please give me a shout!



Thank you for listening! And if we still have some time, I would love to answer your questions