Halcyon Architecture "Director's Cut"

Graham Wihlidal
SEED – Electronic Arts

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.

PICA PICA Trailer

https://www.youtube.com/watch?v=LXo0WdlELJk

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

SEED // Halcyon Architecture "Director's Cut"

"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

SEED // Halcyon Architecture "Director's Cut"

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.

SEED // Halcyon Architecture "Director's Cut"

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.

SEED // Halcyon Architecture "Director's Cut"

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.

SEED // Halcyon Architecture "Director's Cut"

Halcyon Goals

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

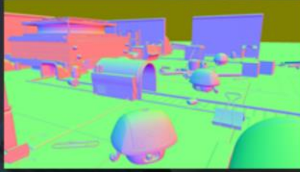
We chose to support rendering locally, and also performing some computation or rendering remotely, and transmitting the results back to the application.

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.

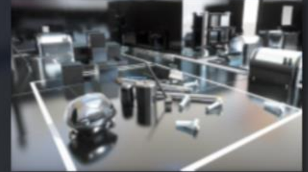
SEED // Halcyon Architecture "Director's Cut"

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)

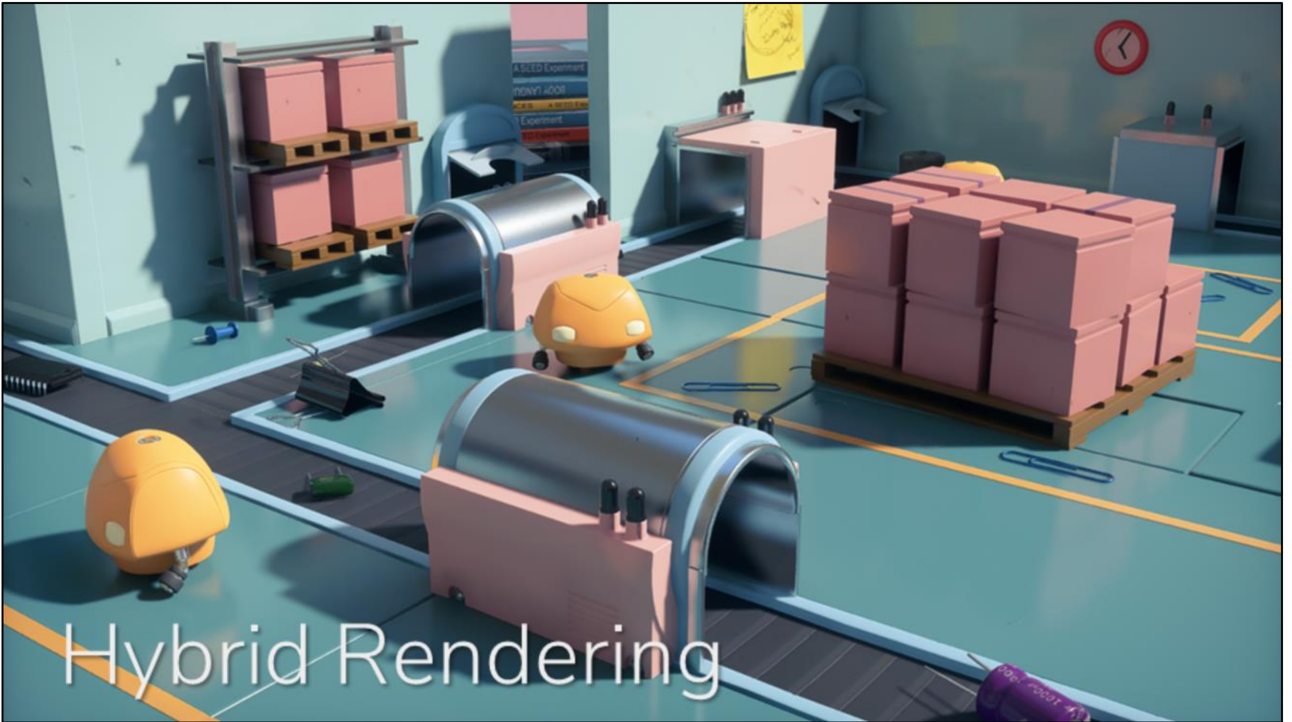
Post Processing (compute)

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

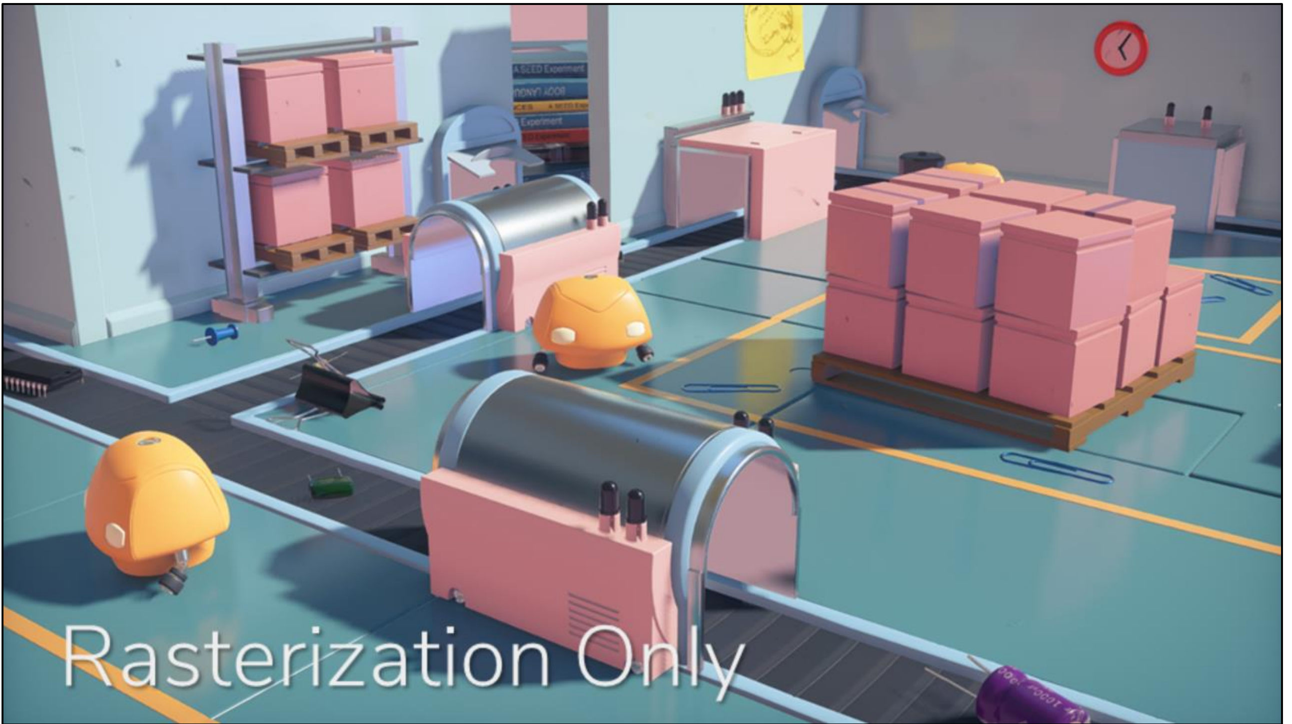
We have a deferred renderer with compute-based lighting, and a fairly standard post-processing stack. And then there are a few pluggable components. We can render shadows via ray tracing or cascaded shadow maps.

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.

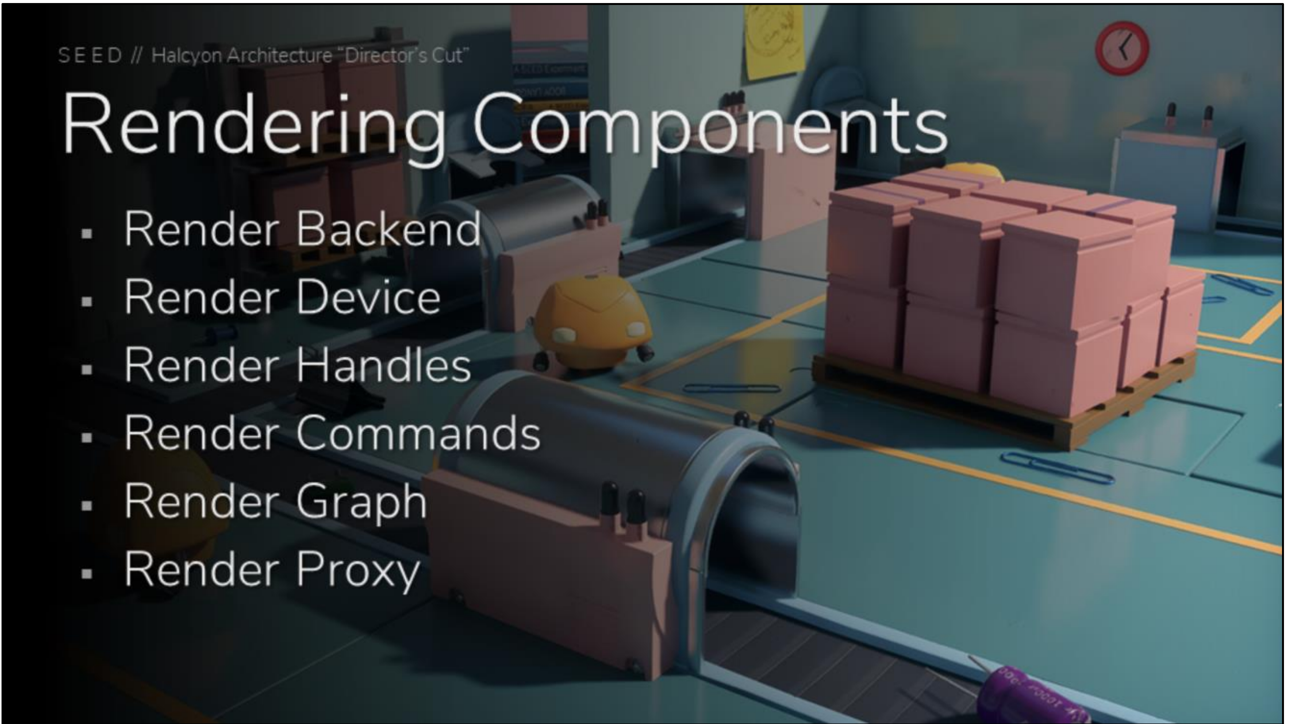
SEED // Halcyon Architecture "Director's Cut"

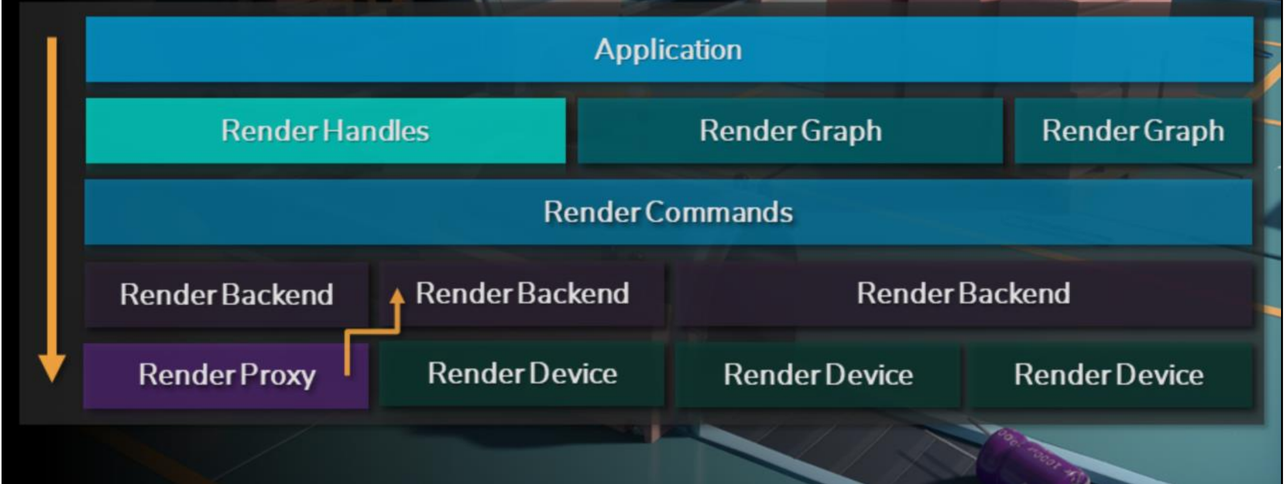
Halcyon Goals

- "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 we can render large and dynamic scenes very efficiently.


SEED // Halcyon Architecture "Director's Cut"
Rendering Components
Render Backend
Render Device
Render Handles
Render Commands
Render Graph
Render Proxy



There are a number of render components making up our architecture.

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.

SEED // Halcyon Architecture "Director's Cut"

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.

Render Backend

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

We have a variety of render backends implemented

SEED // Halcyon Architecture "Director's Cut"

Render Backend

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

For Direct3D 12, we support the latest shader model 6, 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 Direct3D 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.

Render Backend

- Proxy
  - Discussed later in the presentation

We have another pretty crazy backend which I discuss later in this presentation.

SEED // Halcyon Architecture "Director's Cut"

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

SEED // Halcyon Architecture "Director's Cut"

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 provides an efficient mapping of these handles to internal device resources.

First off, I will describe what our render handles are

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

Our rendering resources are associated by handle.

The handles are lightweight (just 64 bits), and the associated resources are fetched with a constant-time lookup. The handles are type safe, protecting against errors 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

SEED // Halcyon Architecture "Director's Cut"

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.

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

SEED // Halcyon Architecture "Director's Cut"

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

SEED // Halcyon Architecture "Director's Cut"

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 allowing each backend to control how this is done, including render state changes and resource barriers or transitions.

SEED // Halcyon Architecture "Director's Cut"

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 Architecture "Director's Cut"

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

SEED // Halcyon Architecture "Director's Cut"

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.

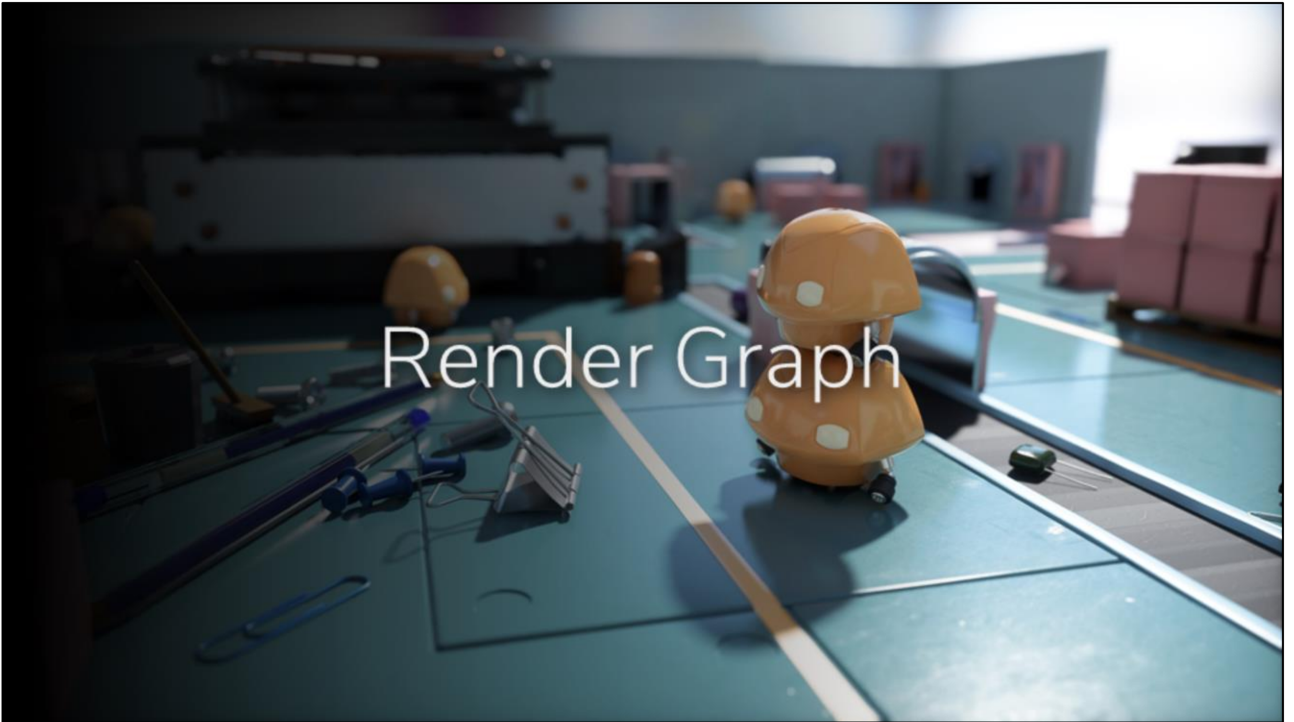
The commands are stateless, which allows for fully parallel recording.

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

SEED // Halcyon Architecture "Director's Cut"

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

SEED // Halcyon Architecture "Director's Cut"

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 need such 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.

SEED // Halcyon Architecture "Director's Cut"

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

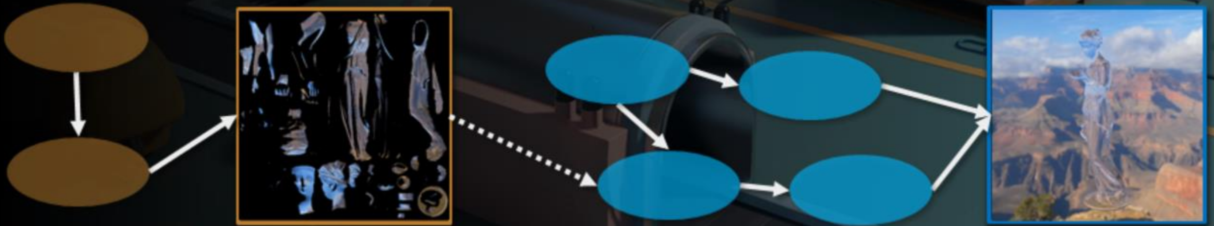
We call our implementation Render Graph, because we don't have the concept of a "frame".

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 graph on each GPU. And they can even run out of core, such as in a server cluster, or on another machine streamed remotely.

SEED // Halcyon Architecture "Director's Cut"

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.

SEED // Halcyon Architecture "Director's Cut"

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 construction in a parallel fashion might be more costly than just running it serially.

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

```cpp
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);
    };
});
```

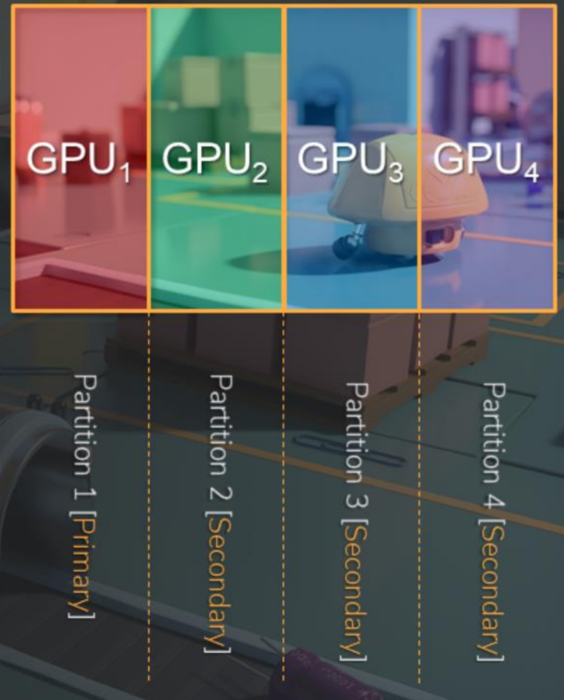
← Construction phase

← Evaluation phase

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

Render Graph

- Explicit heterogeneous mGPU
- Parallel fork-join approach
- Resources copied through system memory using copy queue
  - ~1ms for every 15mb transferred
- Minimize PCI-E transfers
  - Immutable data replicated
  - Tightly pack data

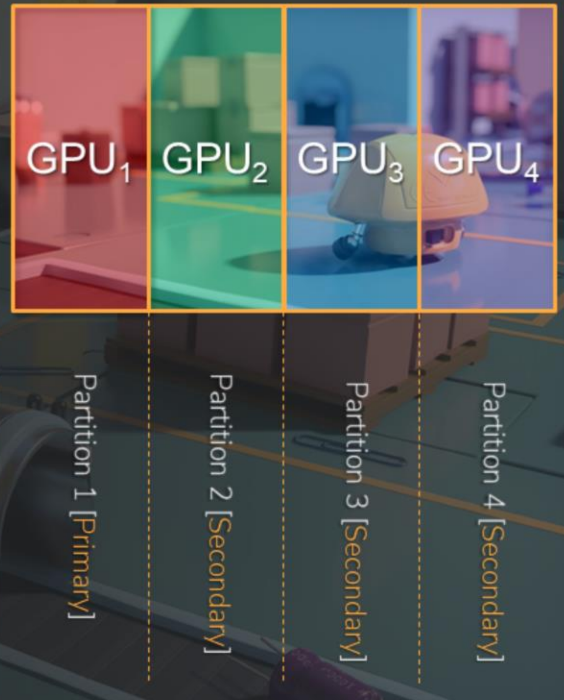


As mentioned, we support explicit heterogeneous multi-GPU. We use a parallel fork-join approach, and we copy resources through system memory using the copy queue. It costs roughly 1ms to transfer 15mb of data, so it's important to minimize how much we transfer. This is done by redundantly replicating immutable data to all GPUs (such as meshes and textures), and also tightly packing or compressing data to minimize the transfer size.

SEED // Halcyon Architecture "Director's Cut"

Render Graph

- Workloads are divided into **partitions**
  - Based on GPU device count
- Single primary device
- Other devices are secondaries
- Variety of scheduling and transfer patterns are necessary
- Simple rules engine



Our workloads are divided into partitions, where the number of partitions is based on the number of available GPUs.

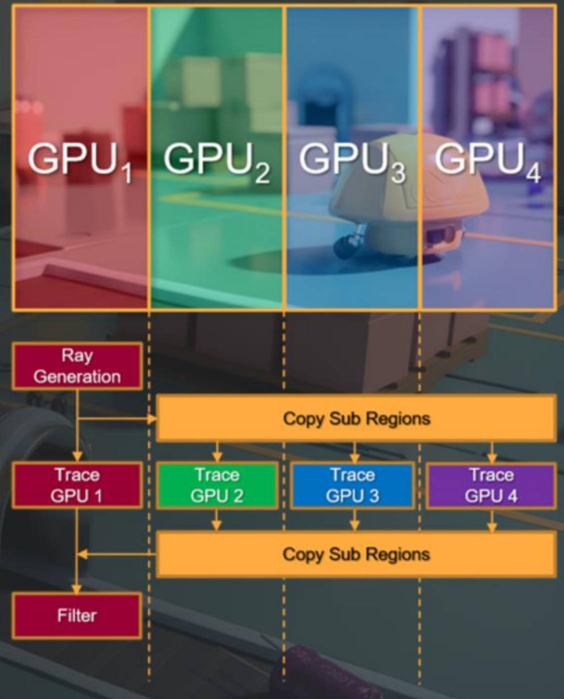
We designate one of the GPUs as the primary device, and the other devices are designated as secondaries.

With complex rendering and computation, there are a variety of scheduling and transfer patterns needed. We built a simple rules engine to drive the logic of our workloads.

SEED // Halcyon Architecture "Director's Cut"

Render Graph

- Run ray generation on primary GPU
- Copy results in sub-regions to other GPUs
- Run tracing phases on separate GPUs
- Copy tracing results back to primary GPU
- Run filtering on primary GPU



For our ray tracing workloads, the common pattern is to run the ray generation on the primary GPU. Then we copy slices or sub-regions of this data to the other secondary GPUs, where each GPU performs tracing on its own sub-region or partition.

The tracing results are then copied back to the primary GPU, and then filtering is performed exclusively on the primary GPU.

This approach avoids many problems with temporal techniques, and also means we don't need to transfer as much data, such as the full g-buffer, to the secondary GPUs.

SEED // Halcyon Architecture "Director's Cut"

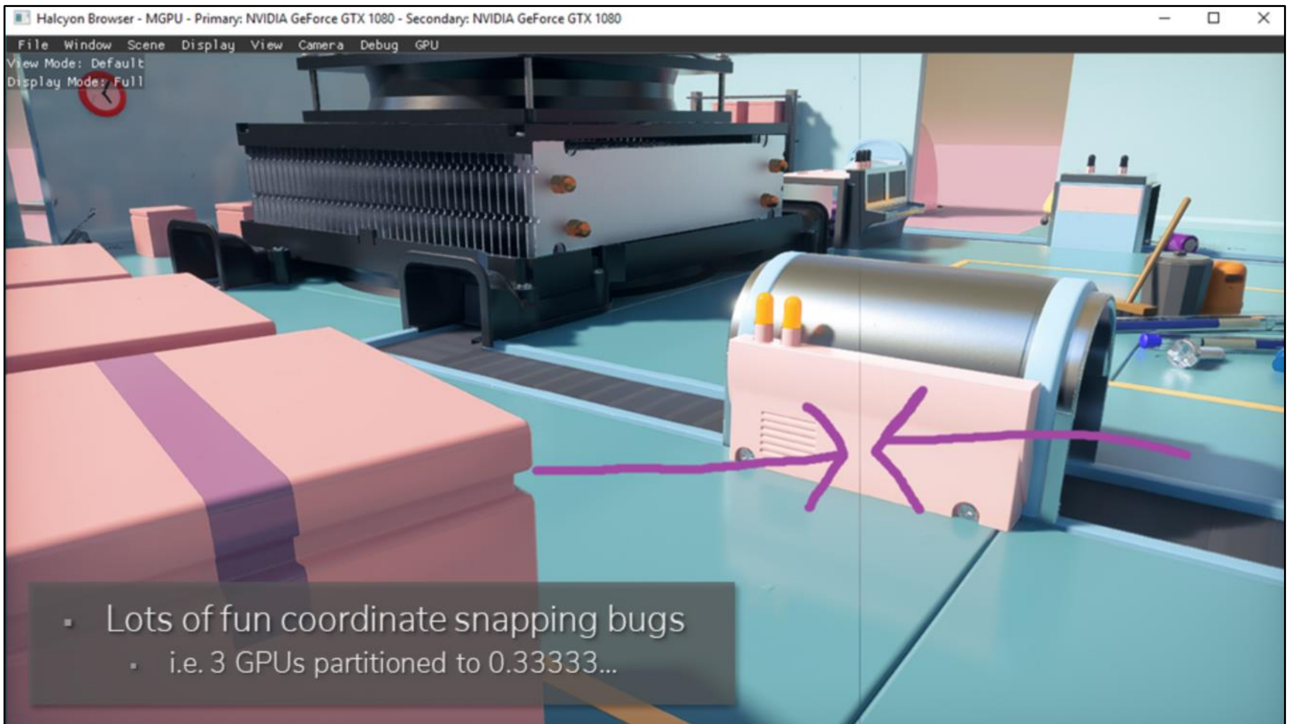
Render Graph

- Only width is divided
- Simplifies textures vs. buffers
- Passes are unaware of GPU count

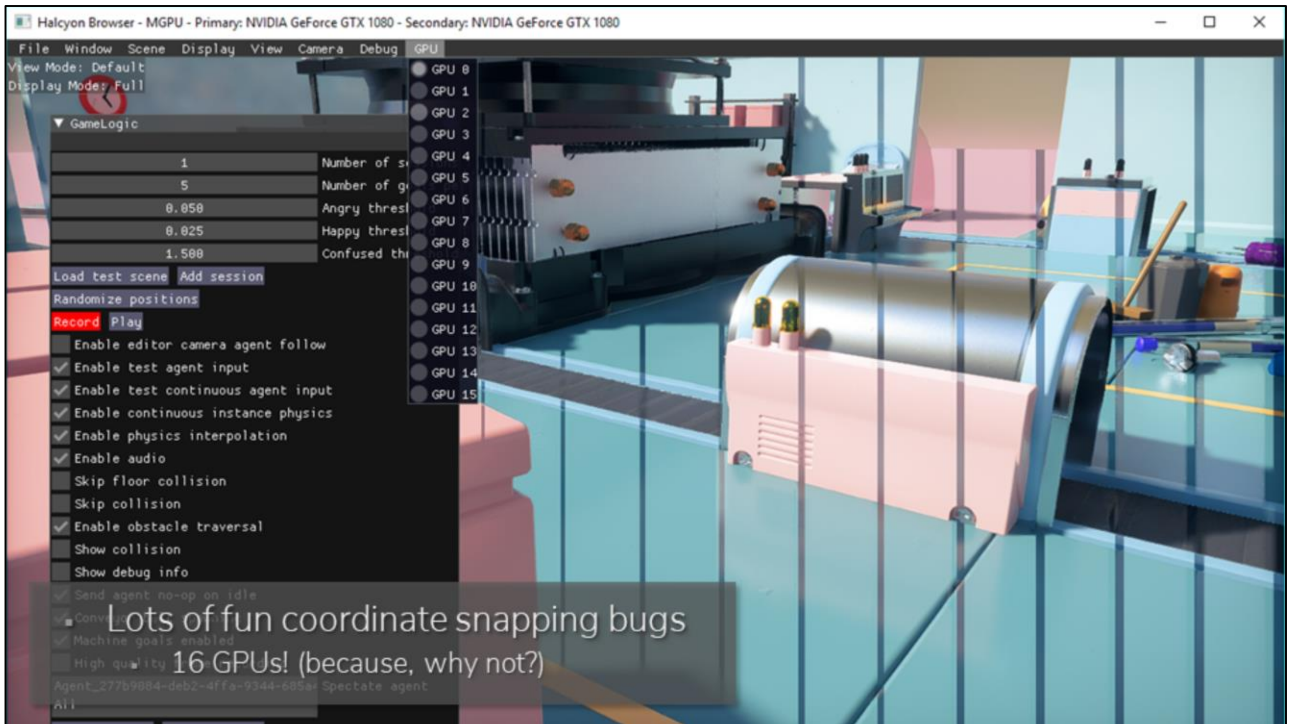
```
glm::vec4 partitionWindow = scheduleData.mgpuShadows ? registry.partitionIsolated() : registry.partitionAll();
const uint32 dispatchOffset = uint32(desc.width * partitionWindow.x);
const uint32 dispatchWidth = uint32(desc.width * partitionWindow.z);

// ...

commandList.dispatch2d(
    pipelineState,
    { ShaderArgument(dynamicConstants.buffer, constantsOffset, registry.createShaderViews(srvs, uavs)) },
    dispatchWidth, desc.height
);
```

The workloads are only divided into partitions based on the X axis, or width. This simplifies textures vs. buffers, since we can treat all sub-regions as 1D instead of 2D.

Passes are unaware of the GPU count, which keeps the code clean, and avoids any edge cases or bugs with untested mGPU configurations. This code snippet shows all that is needed for a pass to scale the workloads, regardless of how many GPUs are in use.



The automatic scaling window works quite well for us, but there were lots of fun coordinate snapping bugs to fix up, like 3 GPUs partitioning to 0.3 repeating.



And then some interesting bugs when we did crazy configurations, like 16 GPUs, because why not? ☺

Render Graph

- RenderGraphSchedule
  - **NoDevices** → Pass is disabled
  - **AllDevices** → Pass runs on all devices
  - **PrimaryDevice** → Pass only runs on primary device
  - **SecondaryDevices** → Pass runs on secondaries if count > 1, otherwise primary
  - **OnlySecondaryDevices** → Pass only runs on secondary devices, disabled unless mGPU

Requested Per Pass →

```
void setSchedule(RenderGraphSchedule passSchedule);

void setTransfer(
    RenderGraphResource resource,
    RenderTransferPartition src,
    RenderTransferPartition dst,
    RenderTransferFilter dstFilter);
```

Our simple rules engine allows for a pass to specify the scheduling, and also request transfers in or out of the pass.

Render Graph

- RenderTransferPartition
  - **PartitionAll** → Select all partitions from device
  - **PartitionIsolated** → Select isolated region from device
- RenderTransferFilter
  - **AllDevices** → Transfer completes on all devices
  - **PrimaryDevice** → Transfer completes on the primary device
  - **SecondaryDevices** → Transfer completes on all secondary devices

Requested Per Pass →

```
void setSchedule(RenderGraphSchedule passSchedule);

void setTransfer(
    RenderGraphResource resource,
    RenderTransferPartition src,
    RenderTransferPartition dst,
    RenderTransferFilter dstFilter);
```

The transfers specify what partitions are copied from source to destination.

And the transfers also specify rules for the transfer destination, allowing for basic copies, multi-cast copies, etc..

Render Graph

- PartitionAll → PartitionAll
  - Copies full resource on one GPU to full resource on all specified GPUs
- PartitionAll → PartitionIsolated
  - Copies full resource on one GPU to isolated regions on all specified GPUs (partial copies)
- PartitionIsolated → PartitionAll
  - (Invalid configuration)
- PartitionIsolated → PartitionIsolated
  - Copies isolated region on one GPU to isolated regions on all specified GPUs (partial copies)

The transfer partition is specified for both the source and the destination GPU, allowing for a variety of transfer patterns.

```
renderGraph.addPassCallback("Shadow Mask Pack", [&](RenderGraphBuild& build)
{
    build.setSchedule(RenderGraphSchedule::AllDevices);

    // ...

    build.setTransfer(
        outputTexture,
        RenderTransferPartition::PartitionIsolated,
        RenderTransferPartition::PartitionIsolated,
        RenderTransferFilter::PrimaryDevice);

    auto pipelineState = pipelines.pipelineState(ShaderPipelineId::ShadowMaskPack);

    return [=](RenderGraphRegistry& registry, RenderCommandList& commandList)
    {
        glm::vec4 partitionWindow = scheduleData.mgpuShadows ? registry.partitionIsolated() : registry.partitionAll();
        const uint32 dispatchOffset = uint32(desc.width * partitionWindow.x);
        const uint32 dispatchWidth = uint32(desc.width * partitionWindow.z);

        // ...

        commandList.dispatch2d(
            pipelineState,
            { ShaderArgument(dynamicConstants.buffer, constantsOffset, registry.createShaderViews(srvs, uavs)) },
            dispatchWidth, desc.height
        );
    };
});
```

Devices this pass will run on

Schedule transfers in or out

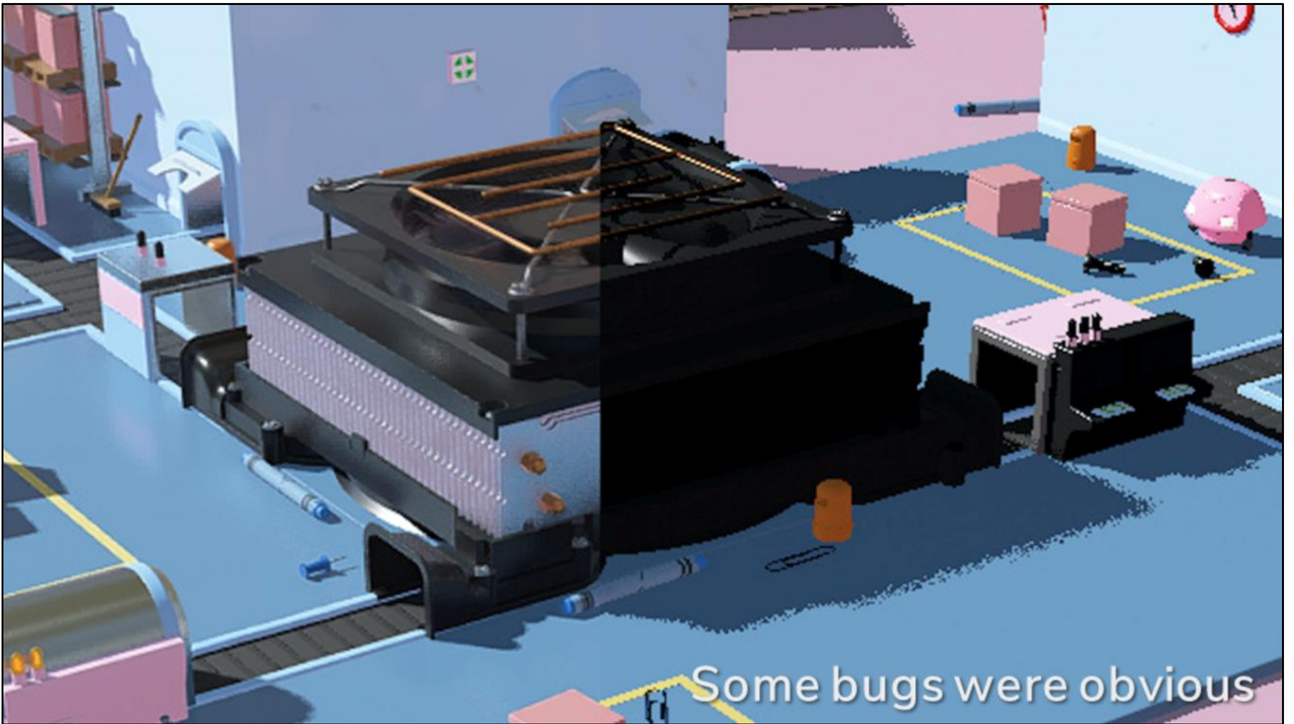
Scaling work dimensions for each GPU

Here is an example render graph pass which specifies the mGPU scheduling policy, in this case, to run on all available devices.

This example also schedules a transfer of the resultant data back to the primary device. Isolated to isolated means that each device will only copy the sub-region or partition that it was responsible for computing.

In the bottom half of the code, you can see the evaluation phase which is scaling the work dimensions for the compute dispatch, based on the mGPU configuration.

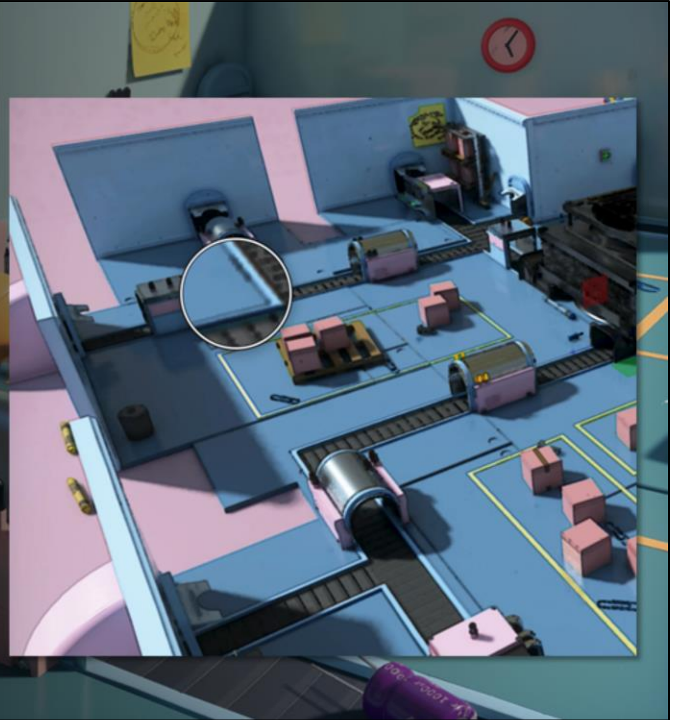
When developing mGPU support for our render graph passes, some scheduling or transfer bugs were obvious.



Here’s another example

Render Graph

- Some bugs were subtle
  - Weird cell shading? ☺
- Incorrect transfers?
  - Transfers in (input data)
  - Transfers out (result data)
- Incorrect scheduling?
  - Pass not running
  - Pass running when it shouldn't
  - Partition window



And then some bugs were very subtle, like this weird cell shading.

Incorrect transfers would result in input data being incorrect or uninitialized, or result data not being copied back to the main GPU for filtering.

Incorrect scheduling would cause passes to not run, to run when they shouldn't, or have an incorrect partitioning window, interfering with the work of another GPU.

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 – we ended up adding mGPU support to reflections, shadows, lighting, global illumination, shadows, and ambient occlusion.

SEED // Halcyon Architecture "Director's Cut"

Render Graph

- Implicit data flow via explicit scopes
  - "Long-distance" extensible parameter passing
- Scope given to each render pass
  - Can create nested scope for sub-graph
  - Results stored into scope
- Hygiene via nesting and shadowing

```
{
    gbuffer <- render_opaque()
    gbuffer <- render_decals(gbuffer)

    {
        gbuffer <- render_opaque()
        render_lighting(gbuffer)
    } -> envmap

    apply_envmap(gbuffer, envmap)
}
```

```
struct RenderGraphAreaLight
{
    RenderGraphResource triangleLightList;
    uint32 triangleCount;
};
```

The initial implementation of render graph would explicitly chain input and output dependencies between passes, but this lead to messy coupling, and also explicit checks like if a depth pre-pass has already run import the depth target, and if not, clear the depth target or disable depth.

We improved the implementation to support implicit data flow using explicit scopes. This allows for long-distance extensible parameter passing. A scope is given to each render pass, which supports nested scopes for sub-graphs, and the results are stored in the scope. Hygiene is provided via nesting and shadowing.

Render Graph

- Lookup by type
  - **scope.get<T>() -> &T**
- Parameters in "plain old data" structs
  - RenderGraphResource, RenderHandle
  - float, int, mat4, etc.

```
{
    gbuffer <- render_opaque()
    gbuffer <- render_decals(gbuffer)

    {
        gbuffer <- render_opaque()
        render_lighting(gbuffer)
    } -> envmap

    apply_envmap(gbuffer, envmap)
}
```

```
struct RenderGraphAreaLight
{
    RenderGraphResource triangleLightList;
    uint32 triangleCount;
};
```

The scopes can be looked up by type, such as depth or gbuffer resources. The lookup returns parameters stored in plain old data structs.

SEED // Halcyon Architecture "Director's Cut"

Render Graph DSL

- Experimental
- Macro Magic

```
_def_pass(Present) {
    _use_pipeline(PresentCs) {
        auto& output = _scope(RenderGraphOutputTexture);
        const auto viewData = _scope(RgViewData);

        auto inputTexture = rg.read(_scope(RgMainTexture).texture, RenderBindFlags::ShaderResource);
        output.texture = rg.write(output.texture, RenderBindFlags::UnorderedAccess);

        _when_drawn {
            _def_srvs(
                _res_tex2d(inputTexture),
            );

            _def_uavs(
                _res_tex2d(output.texture),
            );

            commandList.dispatch2d(
                _shader_args,
                viewData.viewWidth,
                viewData.viewHeight
            );
        };
    };
}
```

We have also been experimenting with various implementations of a DSL for Render Graph – currently it's using hacky macro magic, but we're thinking about writing a code generator using Rust, relying on many amazing features of the language to develop our DSL.

The goal here is to make Render Graph fully data driven, and even serializable.

SEED // Halcyon Architecture "Director's Cut"

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

Inputs		Outputs
Reflection Filter Uniforms (version:0) Reflection Temporal 0 (version:0) Reflection Temporal 1 (version:0) Reflection Meta Temporal 0 (version:0) Reflection Meta Temporal 1 (version:0) Reflection Hit0 Texture (version:2) Reflection Hit1 Texture (version:2)	Inputs	Reflection Filter Uniforms (version:1) Reflection Temporal 0 (version:1) Reflection Meta Temporal 0 (version:1) Reflection Output Texture (version:1) Reflection Meta Output Texture (version:1) Reflection Variance Output Texture (version:1) Reflection Velocity Texture (version:1)

Pass: Reflection Filter 2 Pass

Inputs		Outputs
Reflection Filter Uniforms (version:0) Reflection Output Texture (version:1) Reflection Meta Output Texture (version:1) Reflection Variance Output Texture (version:1) Reflection Output Texture (version:0) Gbuffer History 0 (version:2) Depth (version:2)	Inputs	Reflection Filter Uniforms (version:1) Reflection Output Texture (version:1)

Pass: Reflection Merge Pass

Inputs		Outputs
Reflection Output Texture (version:1) Lighting (version:1)	Inputs	Lighting (version:2)

Pass: Temporal AA Pass

Inputs		Outputs
Lighting (version:2) TAA History Texture 0 (version:0) Temporal Reprojection Texture (version:1) TAA History Texture 1 (version:0) TAA Output Texture (version:0)	Inputs	TAA History Texture 1 (version:1) TAA Output Texture (version:1)

Pass: Velocity reduce

Inputs		Outputs
Velocity Texture 0 (version:2) Reduced velocity texture (version:0)	Inputs	Reduced velocity texture (version:1)

Pass: Velocity reduce

Inputs		Outputs
Reduced velocity texture (version:1) Reduced velocity texture (version:0)	Inputs	Reduced velocity texture (version:1)

Pass: Velocity dilate

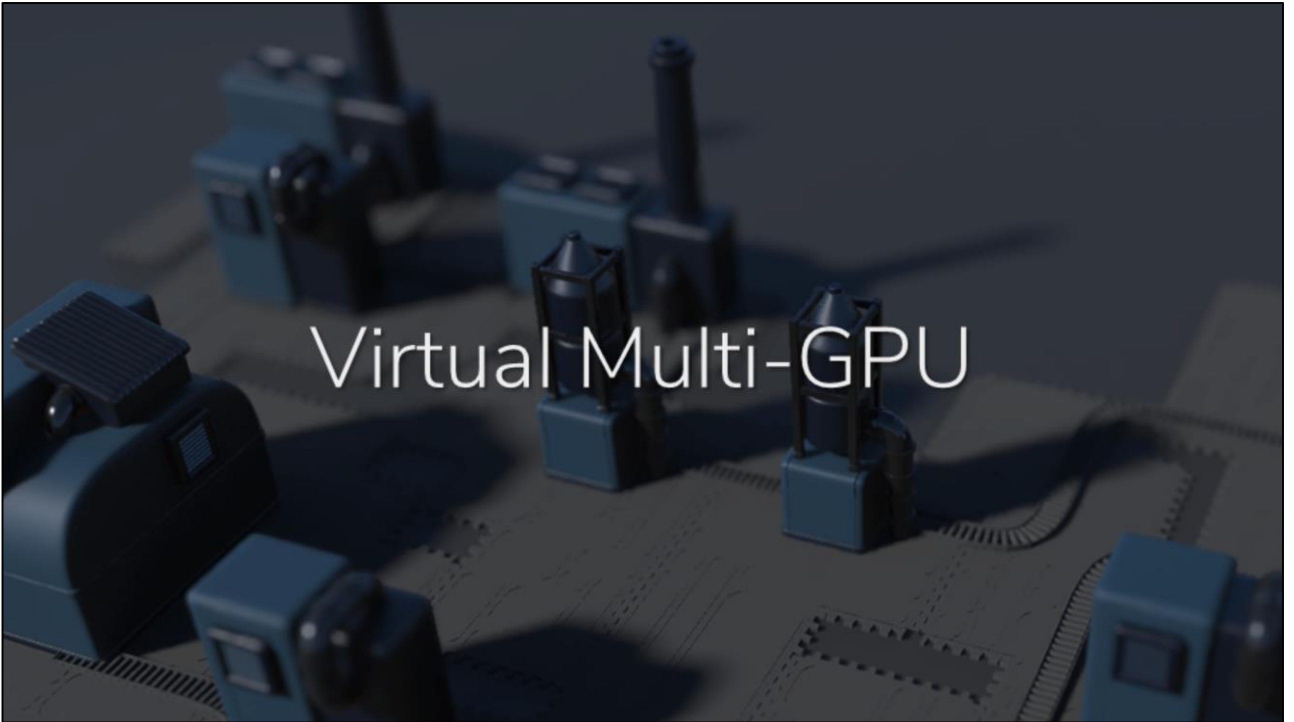
Inputs		Outputs
Reduced velocity texture (version:1) Dilated velocity texture (version:0)	Inputs	Dilated velocity texture (version:1)

Pass: Motion blur

Inputs		Outputs
Depth (version:2) Velocity Texture 0 (version:2) Dilated velocity texture (version:1) TAA Output Texture (version:1) Motion blurred image (version:0)	Inputs	Motion blurred image (version:1)

Pass: Bloom Extract Pass

THE METAVERSE
A Simple Guide
Author: TBD

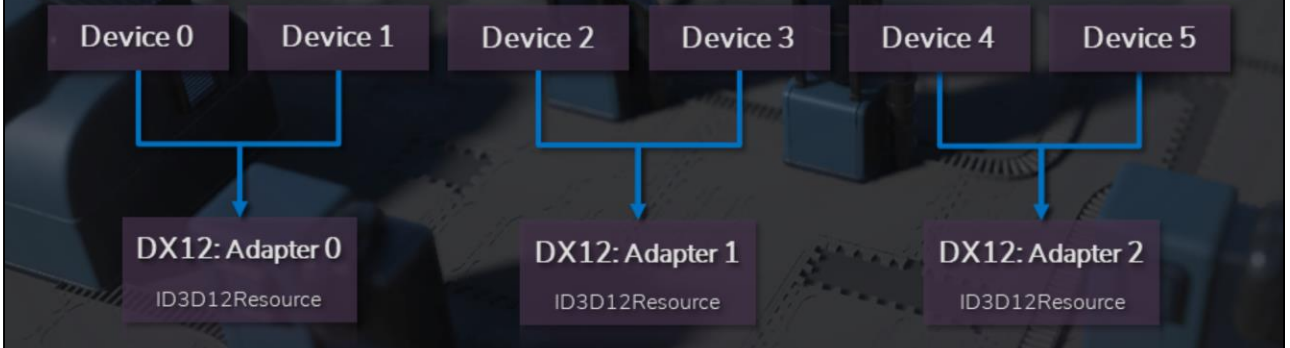


In order to make multi-GPU development easier, we built something we call virtual multi-GPU

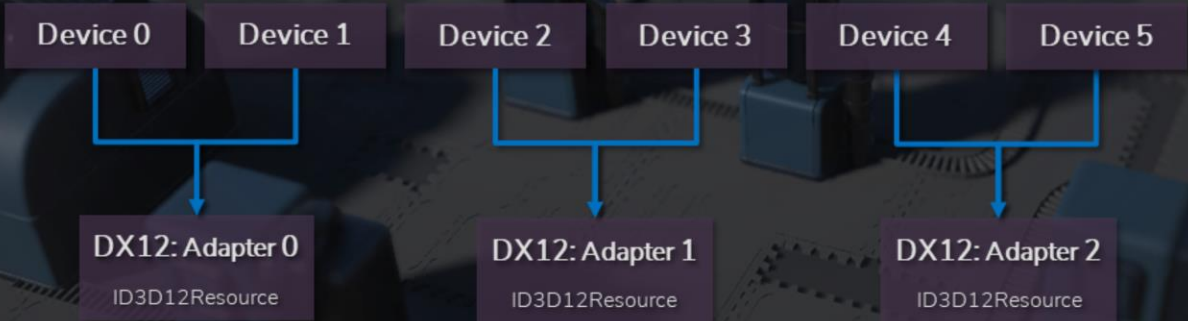
Virtual Multi-GPU

- Most developers have single GPU
- Uncommon for 2 GPU machines
- Rare for 3+ GPU
  - Practical for show floor and cranking up to 11
  - Impractical for regular development ☺

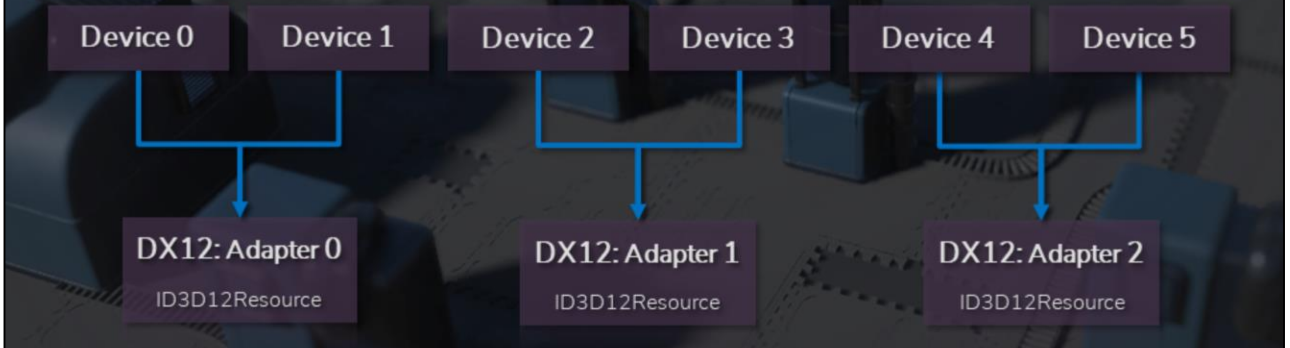
Most developers typically have just a single GPU in their workstations. It is quite uncommon for 2 GPU machines, and it is rare for machines to have more than 2 GPUs. Having a huge machine is practical for the show floor, and cranking settings up to 11, but this is impractical for regular development. Testing these configurations becomes challenging.



Our solution was to build a device indirection table. We enumerate available device adapters, and based on a setting specifying how many copies of a device are desired, we redirect our own virtual device indices to actual adapter indices.



We do this by creating multiple instances of a given device. Similar to multiple applications sharing a GPU, the OS (such as WDDM on Windows 10) will execute multiple instances of a device sequentially.



This approach increases the overall wall time, so don't use it for end to end profiling. However, this is amazing for development and testing, even estimating the possible performance characteristics of the mGPU algorithm in isolation.

SEED // Halcyon Architecture "Director's Cut"

Virtual Multi-GPU

- PICA PICA developers all had 1 GPU
- Limited testing with 2 GPUs
- Show floor at GDC 2018 was 4 GPUs
  - Virtual-only testing...
  - Crossed fingers
  - Worked flawlessly!

There is a fun story around this stuff.. For PICA PICA, all developers only had a single GPU, and there was very limited testing with 2 GPUs.

The show floor at GDC 2018 was a machine with 4 GPUs, a configuration we had only ever tested using virtual mGPU. We crossed our fingers, and hoped we didn't make a horrible mistake by relying exclusively on virtual mGPU for testing.

The demo worked flawlessly on the first try with the actual hardware, and gave us the wall time improvements we had estimated.

SEED // Halcyon Architecture "Director's Cut"

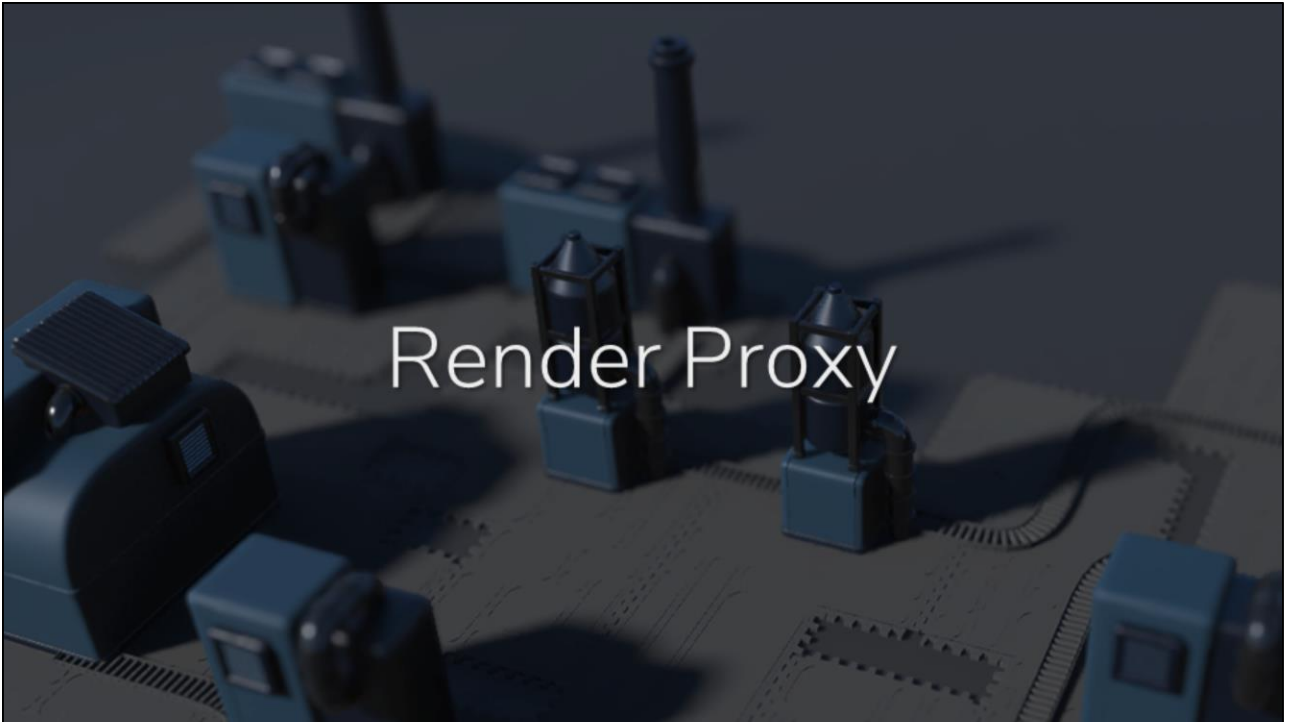
Virtual Multi-GPU

- Develop and debug multi-GPU with only a single GPU
- Virtual mGPU reliably reproduces most bugs!
- Entire features developed without physical mGPU
  - i.e. Surfel GI (the night before GDC.. ☺)

Virtual mGPU allows us to develop multi-GPU with only a single GPU

Virtual mGPU even reproduces most mGPU bugs in the same way, which is incredibly useful.

Some developers never tested on physical mGPU, and developed entire features like our surfel GI using only virtual mGPU. (The night before GDC, I might add..)



Another interesting capability of our architecture is our render proxy

Render proxy is another render backend implementation, like DirectX12 or Vulkan, except this allows us to mix any API with any OS.

Render Proxy

- Render API calls are routed remotely
- Uses gRPC (high performance RPC framework)
- Use an API on an incompatible OS
  - e.g. Direct3D 12 on macOS or Linux

The way it works is our render API calls get routed remotely using gRPC, which is a high performance RPC framework developed by Google.

This lets us use an API on an incompatible OS, such as using Direct3D 12 on macOS or Linux.

This is especially nice for bringing up new platforms, as we can get base memory, IO, and core functionality running, without initially worrying about the graphics system.

Render Proxy

- Scale large workloads with a GPU cluster
  - Some API as render graph mGPU
- Only rendering is routed, scene state is local
- Work from the couch!
  - i.e. DirectX ray tracing on a MacBook ☺

Render proxy allows us to scale large workloads with a GPU cluster. Render graph passes which support mGPU can transparently scale to the cluster without any additional code, which is a really powerful design.

Another interesting property is that only the rendering is routed, the scene state is local. This means that the rendering is based on the actual local version of code and data, which allows for fast iteration and development.

This lets us work from the couch! Such as using Direct X ray tracing with a Turing GPU from a Macbook

Render device → gRPC

```
bool RenderProxy::createShader(uint32 deviceIndex, RenderResourceHandle handle, const RenderShaderDesc& desc)
{
    grpc::ClientContext context;

    rpc::CreateShader request = {};
    request.mutable_instance()->CopyFrom(m_instance);
    request.set_deviceindex(deviceIndex);

    translate(handle, *request.mutable_handle());

    rpc::CreateShader_Desc& rpcDesc = *request.mutable_desc();
    rpcDesc.set_type(translate(desc.type));
    rpcDesc.set_data(desc.data, desc.dataSize);

    rpc::RenderProxyResult response = {};
    response.set_success(false);

    grpc::Status status = m_stub->DeviceCreateShader(&context, request, &response);
    return status.ok();
}
```

With our render handle design, it's pretty easy to implement gRPC proxy functions of our render backend interface. Create and destroy resource calls will pass the handle value and initial data over to the remote proxies to create their representation of that handle.

High level command lists are recorded as normal, and these command streams are replicated over to the remote proxies, referencing the resource handles. Remote compilation and submission of the command lists is performed, and then the results are sent back to the application.

Protobuf 3 Schema

```
message CreateShader
{
    message Desc
    {
        RenderShaderType type    = 1;
        bytes data               = 2;
    }

    RenderBackendInstance instance   = 1;
    uint32 deviceIndex               = 2;
    RenderResourceHandle handle      = 3;
    Desc desc                        = 4;
}

message RenderProxyResult
{
    bool success = 1;
}

rpc DeviceCreateShader(CreateShader) returns (RenderProxyResult)
{
    option (google.api.http) =
    {
        post: "/v1/render/create/shader"
        body: "*"
    };
}
```

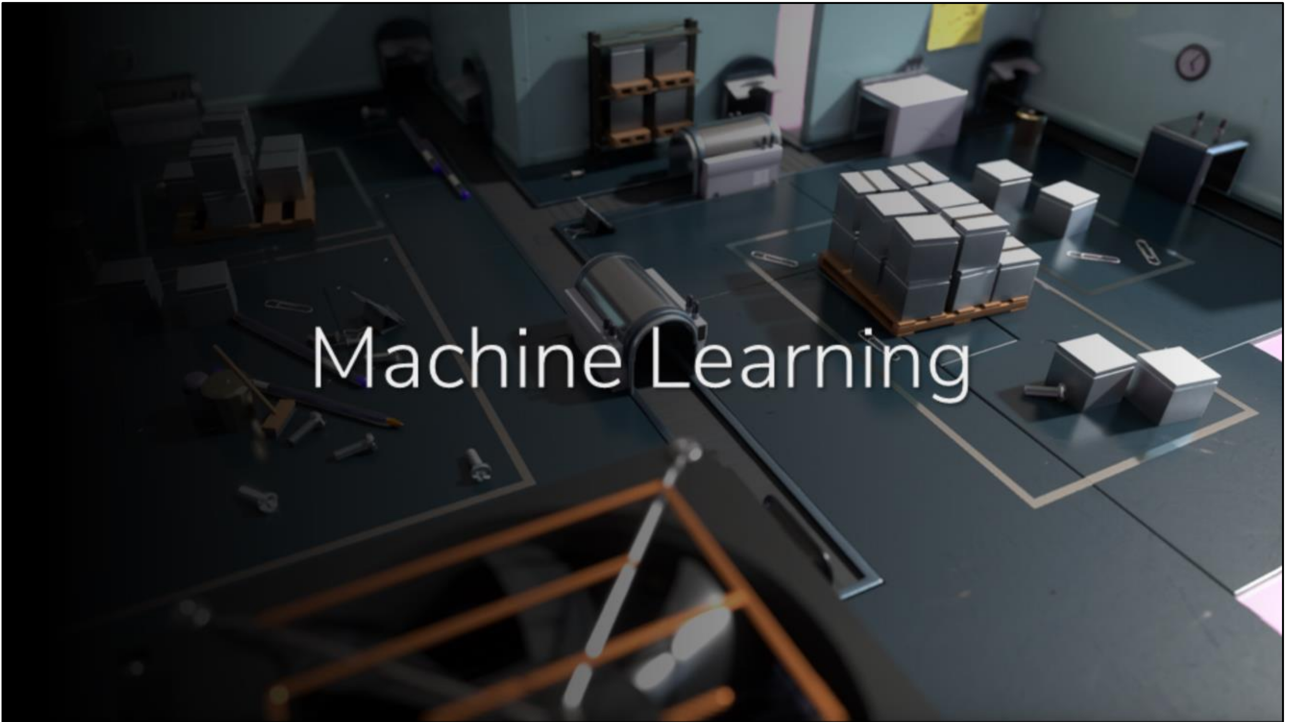
We define the RPC data types and functions with Google Protocol Buffers, which supports gRPC, and also HTTP 1.1 using a reverse REST gateway, so even web browsers can submit work to our render proxy backend and get back the results.

SEED // Halcyon Architecture "Director's Cut"

Render Proxy

- The possibilities are endless!

With our render proxy architecture, the possibilities are truly endless!

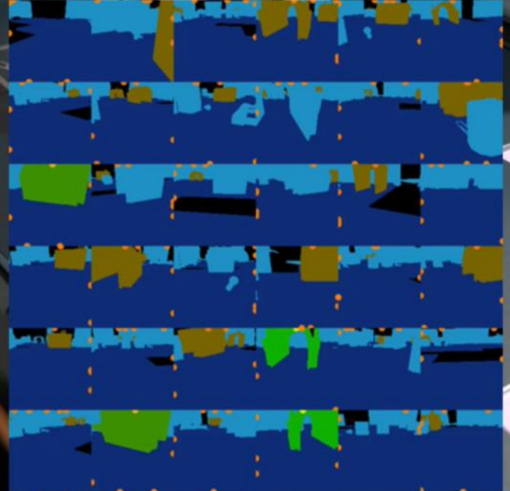


I also wanted to briefly cover some of our machine learning support in Halcyon

SEED // Halcyon Architecture "Director's Cut"

Machine Learning

- Deep reinforcement learning
- Rendering 36 semantic views
- Training with TensorFlow
  - On-premise GPU cluster
- Inference with TensorFlow
  - CPU AVX2
  - In-process



PICA PICA uses deep reinforcement learning, trained with Halcyon rendering 36 semantic views.

The training is performed with TensorFlow and our on-premise GPU cluster.

The PICA PICA demo did in-process inference with TensorFlow using AVX2 on the CPU

SEED // Halcyon Architecture "Director's Cut"

Machine Learning

- Adding inferencing with DirectML
  - Hardware accelerated inferencing operators
  - Resource management
  - Schedule ML work explicitly
  - Interleave ML work with other GPU workloads
- Fall back for other APIs

We are adding inferencing support with DirectML to Halcyon. This will provide hardware accelerated inferencing operators. DirectML allows us to do our own resource management, schedule ML work explicitly, and also interleave ML work with other GPU workloads.

We are investigating a fall back for other APIs if DirectML is not available.

Machine Learning

- Treat trained ML models like any other 3D asset
- Render Graph abstractions
  - Reference the same render resources
  - Similar to chaining compute passes
- Record "meta" render commands
  - Backends can fuse or transform, if desired

We can treat the trained ML models like any other 3D assets.

We will also expose DirectML with render graph abstractions. We want to reference the same render resources, and provide a similar approach to chaining compute passes.

Similar to our other high level commands bracketed within a render pass, we will record some form of "meta" render commands for ML, bracketed together, and this will allow the various backends to fuse or transform these commands for performance, if desired.

Machine Learning

- Provide operators as render commands
  - Activation
  - Convolution
  - Elementwise
  - FC
  - GRU
  - LSTM
  - MatMul
  - Normalization
  - Pooling
  - Random
  - RNN
  - etc.

As mentioned, we will provide various operators as render commands, such as the ones shown here.

Lastly, I wanted to quickly mention some interesting aspects of our asset pipelines

SEED // Halcyon Architecture "Director's Cut"

Asset Pipelines

- Geometry
- Animations
- Shaders
- Sounds
- Music
- Textures
- Scenes
- etc.

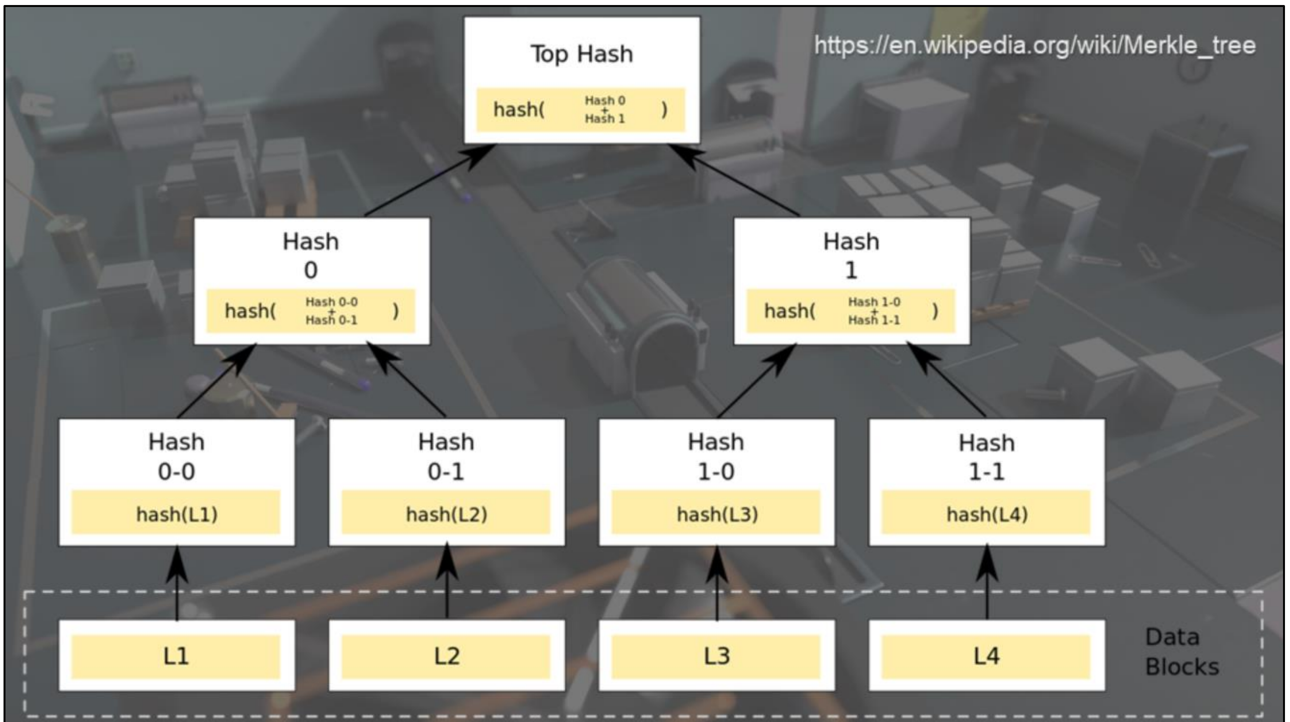
Just like any other engine or framework, we have a variety of asset types that we load from a source or intermediate representation, process with a pipeline, and produce a more optimal representation for runtime.

Asset Pipelines

- Everything is content addressable
  - Hash of data is the identity
  - Sha256
- Merkle trees!
  - Dependency evaluation

The cool thing is that all our assets are content addressable - we don't reference resources by virtual file system path, instead we only reference by a hash like sha256.

We use Merkle trees for efficient dependency evaluation



The general idea with Merkle trees is that the leaf nodes are data blocks, which are hashed, and nodes further up the tree are hashes of their respective children.

You can test if a leaf node is part of a given tree by computing a number of hashes proportional to the logarithm of the number of leaf nodes in the tree. This allows for efficient dependency evaluation, avoiding redundant network transmission of data, and secure verification of data contents.

SEED // Halcyon Architecture "Director's Cut"

Asset Pipelines

- Containerized, running on Kubernetes
  - Google Cloud Platform
  - On-Premise Cluster
    - AMD 1950X TR
    - NV Titan V
- Communication using gRPC and Protobuf
- Google Cloud Storage

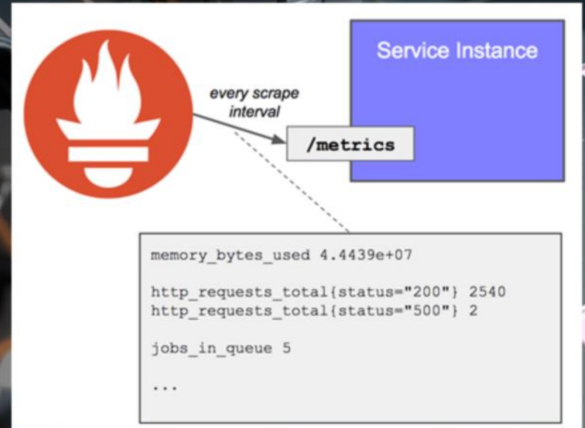
Our pipelines are all containerized in Linux Docker, running on Kubernetes. We can scale our pipelines on Google Cloud Platform, and also in our on-premise cluster, comprised of AMD Threadrippers and Nvidia Titan Vs.

Communication between Halcyon and the pipelines is done with gRPC and Protobuf, and our content addressable data is all backed by Google Cloud Storage.

SEED // Halcyon Architecture "Director's Cut"

Asset Pipelines

- Analytics with Prometheus and Grafana
  - Publish custom metrics
  - Scraped into rich UI
  - Collecting data is important!



Our pipelines support detailed analytics with Prometheus and Grafana. We publish custom metrics to an HTTP endpoint, and these metrics get scraped into a rich UI.

We can't stress enough how important collecting data is – for profiling how long pipelines are taking including bottlenecks to improve, and also for tracking down errors and bugs.

This is what our on-premise GPU cluster metrics look like when displayed with Grafana. This is a very rich and responsive Web UI, and it is very easy for us to add in all sorts of interesting metrics and dashboards for whatever we want to track.

Shaders are also an important component

SEED // Halcyon Architecture "Director's Cut"

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.

SEED // Halcyon Architecture "Director's Cut"

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

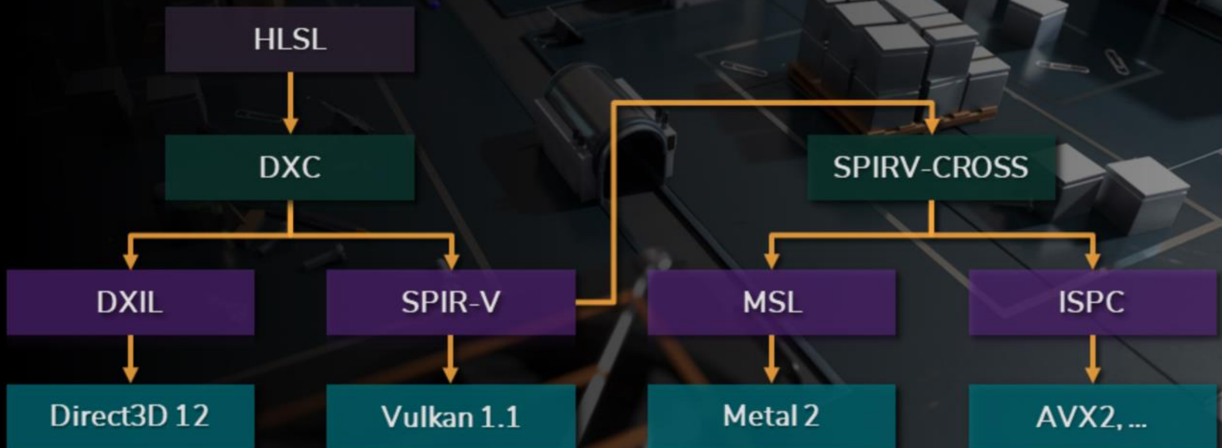
Performance is critical, so we rely heavily on group shared memory, and wave operations

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.



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.

SEED // Halcyon Architecture "Director's Cut"

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.

```hlsl
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);
```

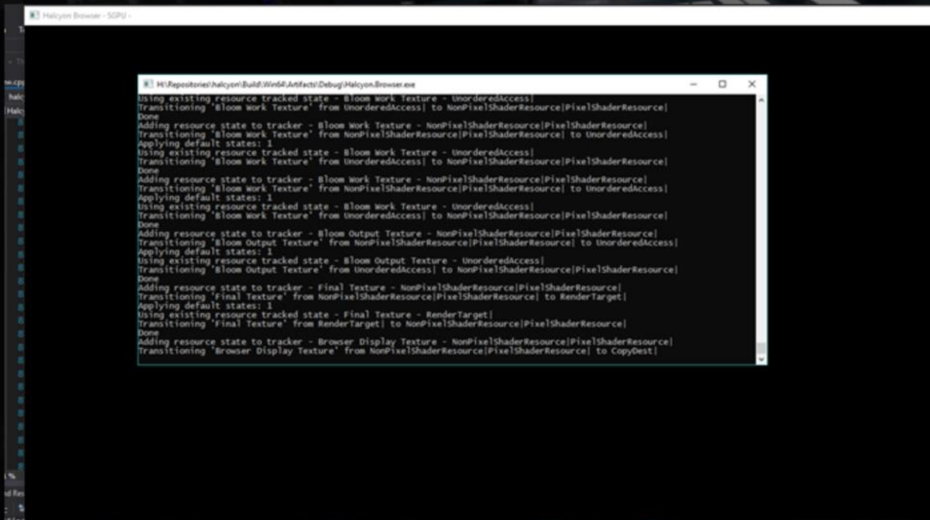
```cpp
{
    // space0
    ShaderArgument(materialConstants.buffer, materialConstantsOffset, materialShaderViews),

    // 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

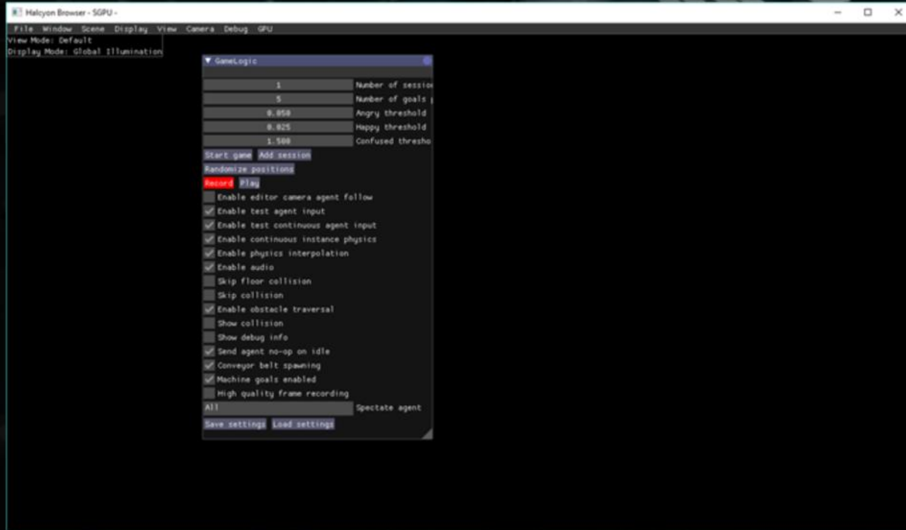
Our architecture simplified the development effort, for Vulkan we just needed a new backend and device implementation, API specific memory allocators, barrier and transition logic, and a resource binding model that aligned with our shader compilation.



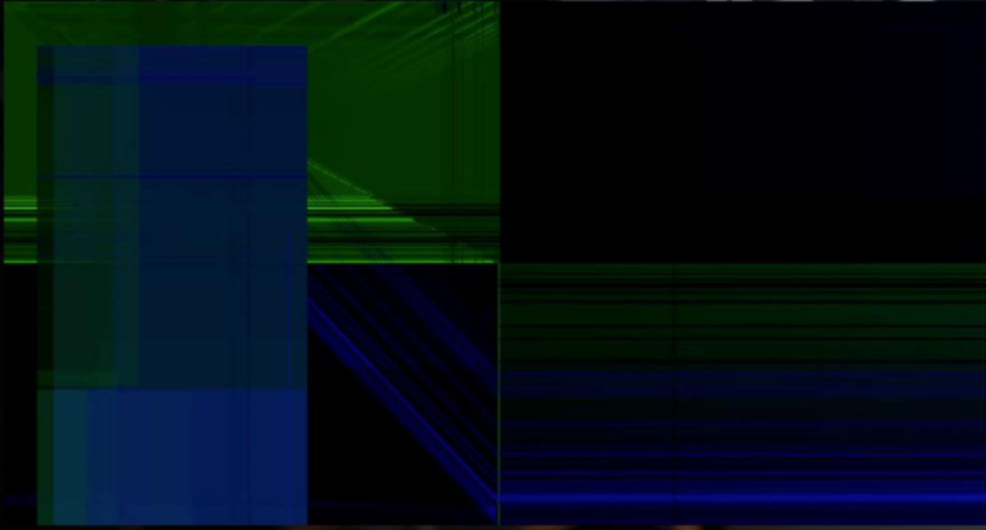
The first stage was to get command translation working, and performing the correct barriers and transitions. This was done using the Vulkan validation layers, and lots of glorious printf debugging

SEED // Halcyon Architecture "Director's Cut"

Vulkan Implementation



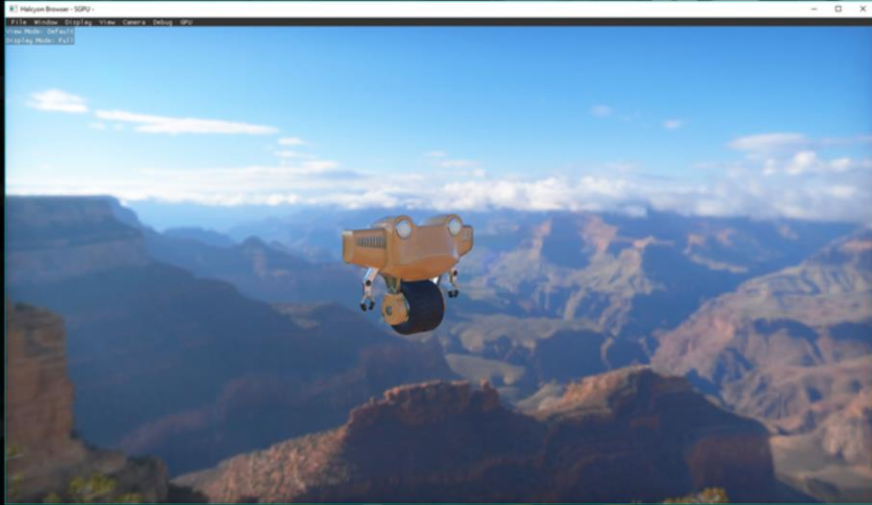
The second stage was getting basic ImGui, resource creation, and swap chain flip working. This meant I could easily toggle any display mode or render settings while debugging.



Naturally, fun bugs occurred

SEED // Halcyon Architecture "Director's Cut"

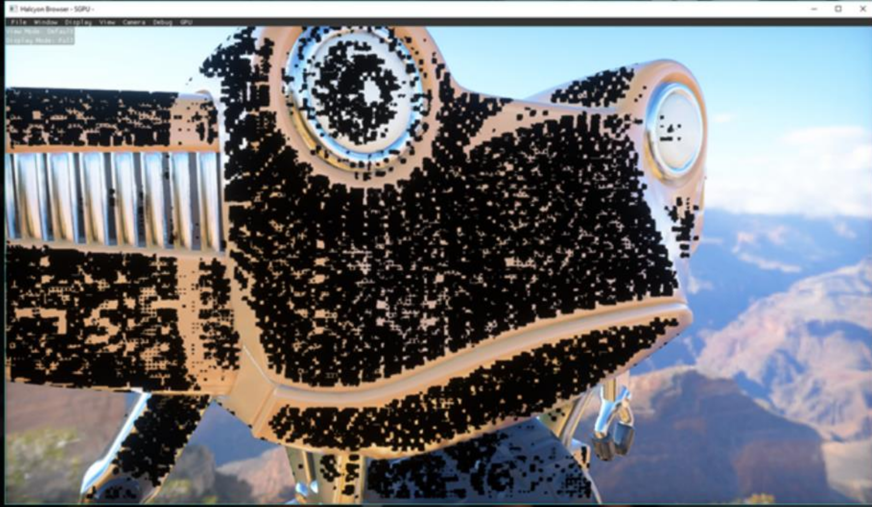
Vulkan Implementation



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

SEED // Halcyon Architecture "Director's Cut"

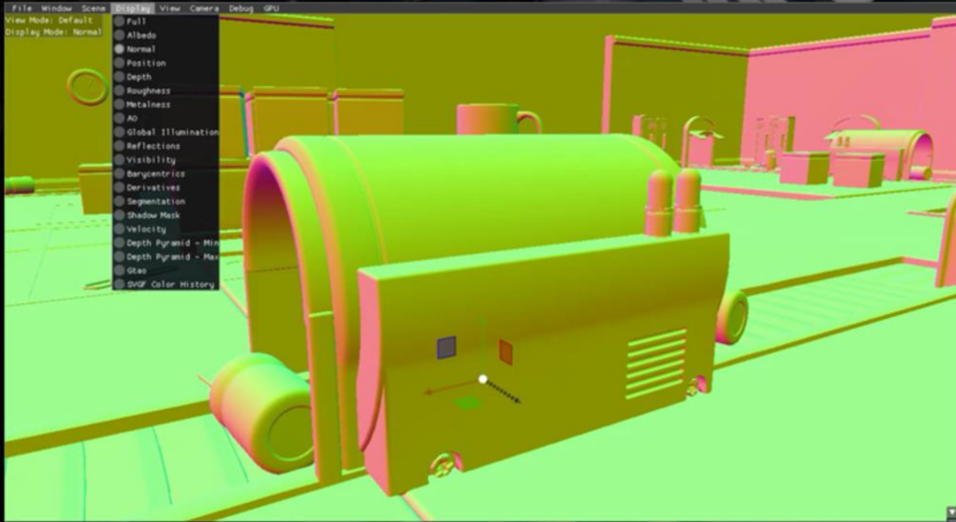
Vulkan Implementation



Plenty of fun bugs with this, as well

Vulkan Implementation

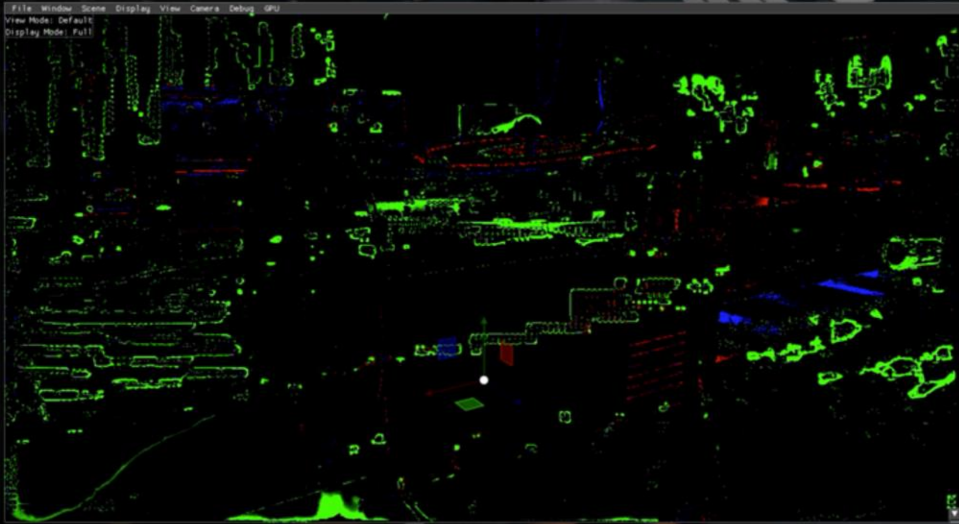
Plenty of fun bugs with this, as well



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.

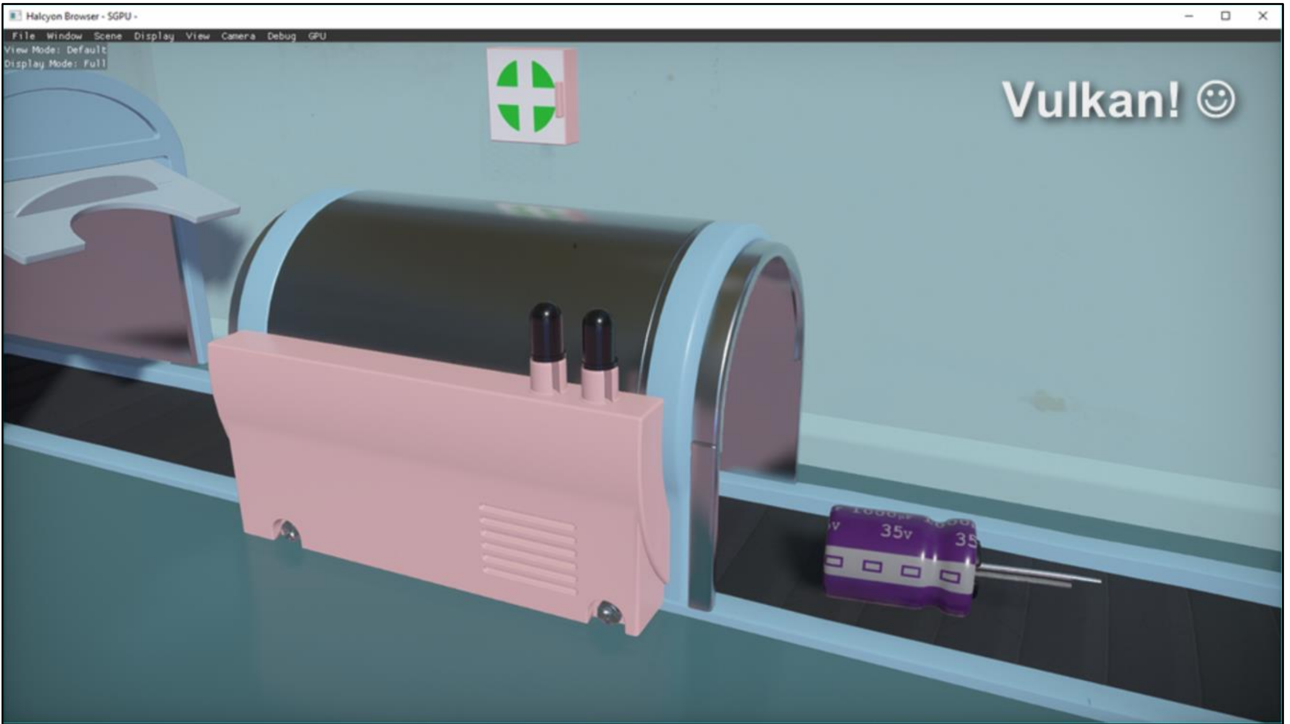
I relied on the fact that render graph will cull any passes not contributing to the final result, so I could easily remove problematic passes from running while I get the basics working.



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.

SEED // Halcyon Architecture "Director's Cut"

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 name spaced 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.

SEED // Halcyon Architecture "Director's Cut"

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).

SEED // Halcyon Architecture "Director's Cut"

SPIR-V Patching

- If 2 of 4 HLSL spaces in use:

Set 0	SRVs (>=1000)	Samplers (>=2000)	UAVs (>=3000)
Set 1	SRVs (>=1000)	Samplers (>=2000)	UAVs (>=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 Architecture "Director's Cut"

```
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.

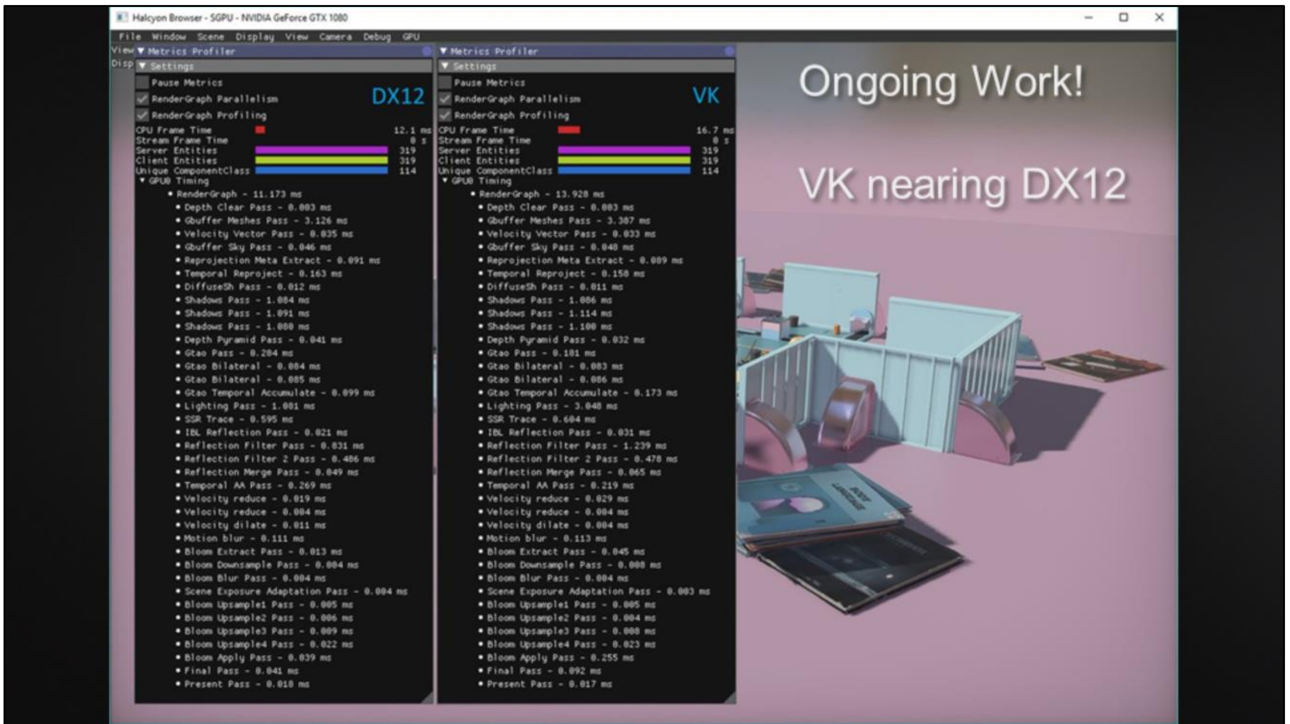
SEED // Halcyon Architecture "Director's Cut"

```
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;
}
```

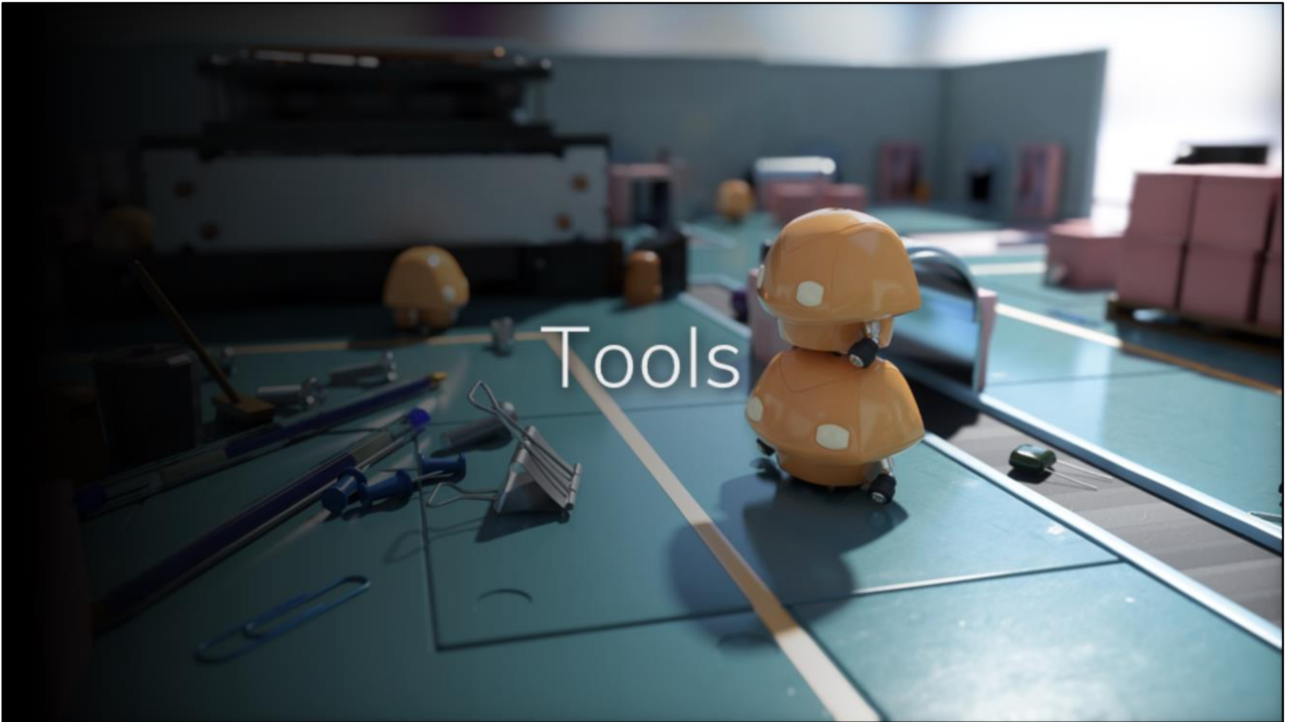
Here is an example of translating a high level begin timing command to Vulkan timestamps

```
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.



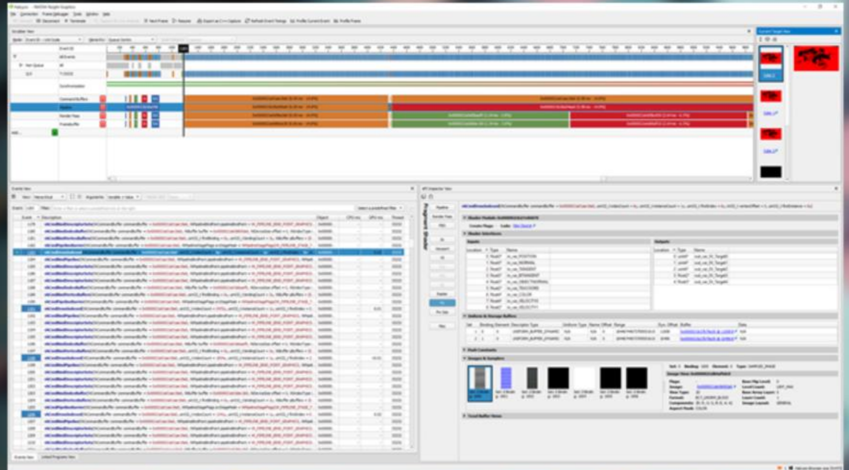
This comparison shows that our Vulkan implementation is nearing the performance of our DirectX 12 version, and is completely usable. There are a number of reasons for the delta, but none of them represent any amount of significant work to resolve.



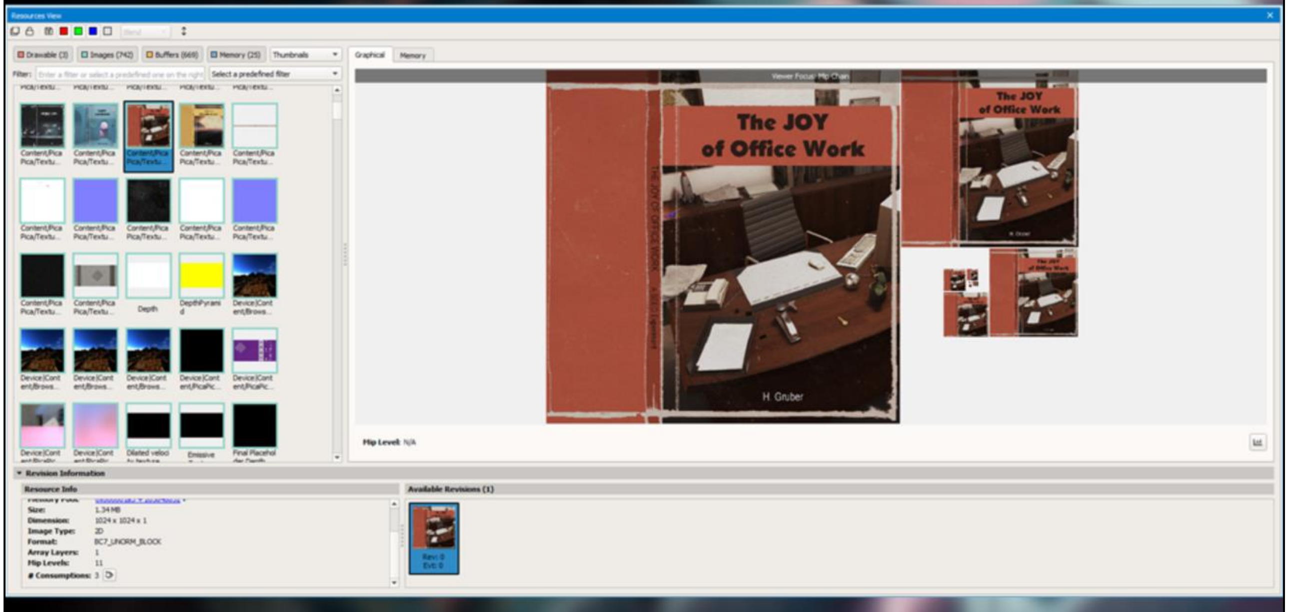
I will briefly mention some useful tools used for the Vulkan implementation

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.

SEED // Halcyon Architecture

Tools

API Statistics View

Summary

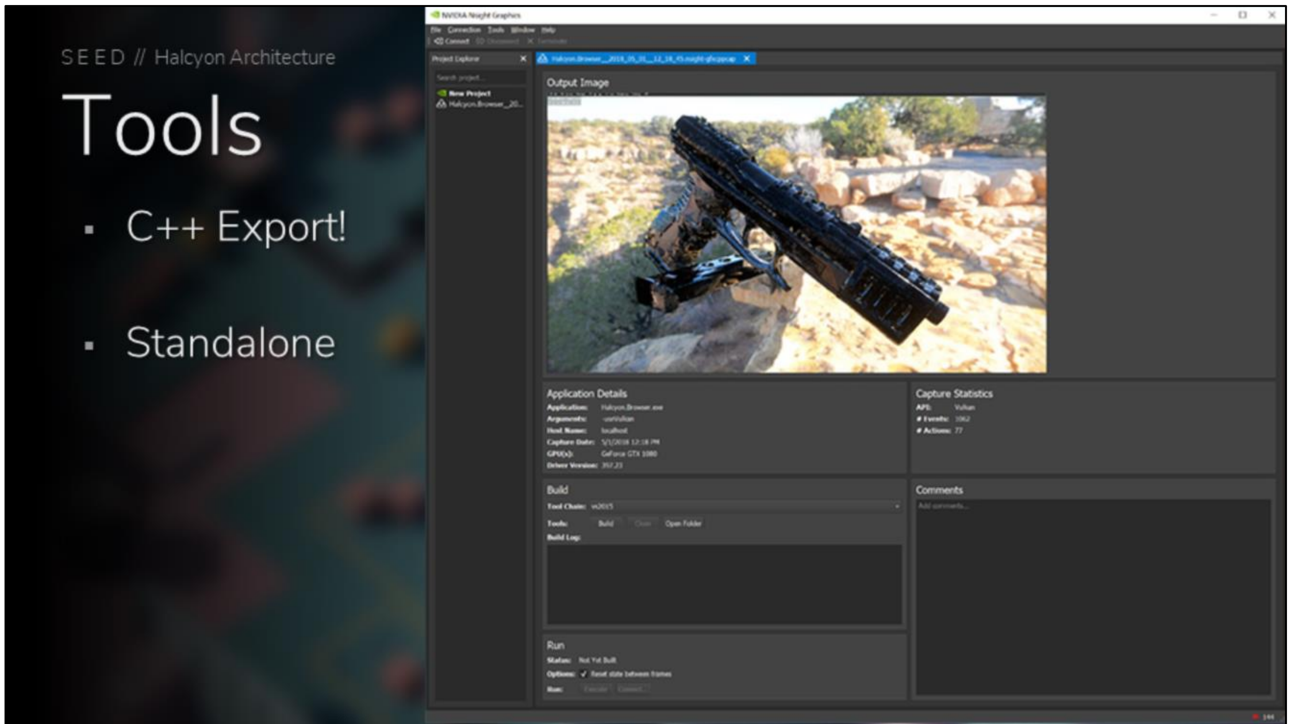
Draws: 1801 **Dispatches:** 38 **Clears:** 27 **Blits:** 1 **Presents:** 1 **Command List Executes** 30 **Signals:** 0 **Waits:** 0 **Misc. Data Update:** 42 **Non-API:** 29 **Other:** 10895 **Total:** 12864

Details

Filter: Enter a filter

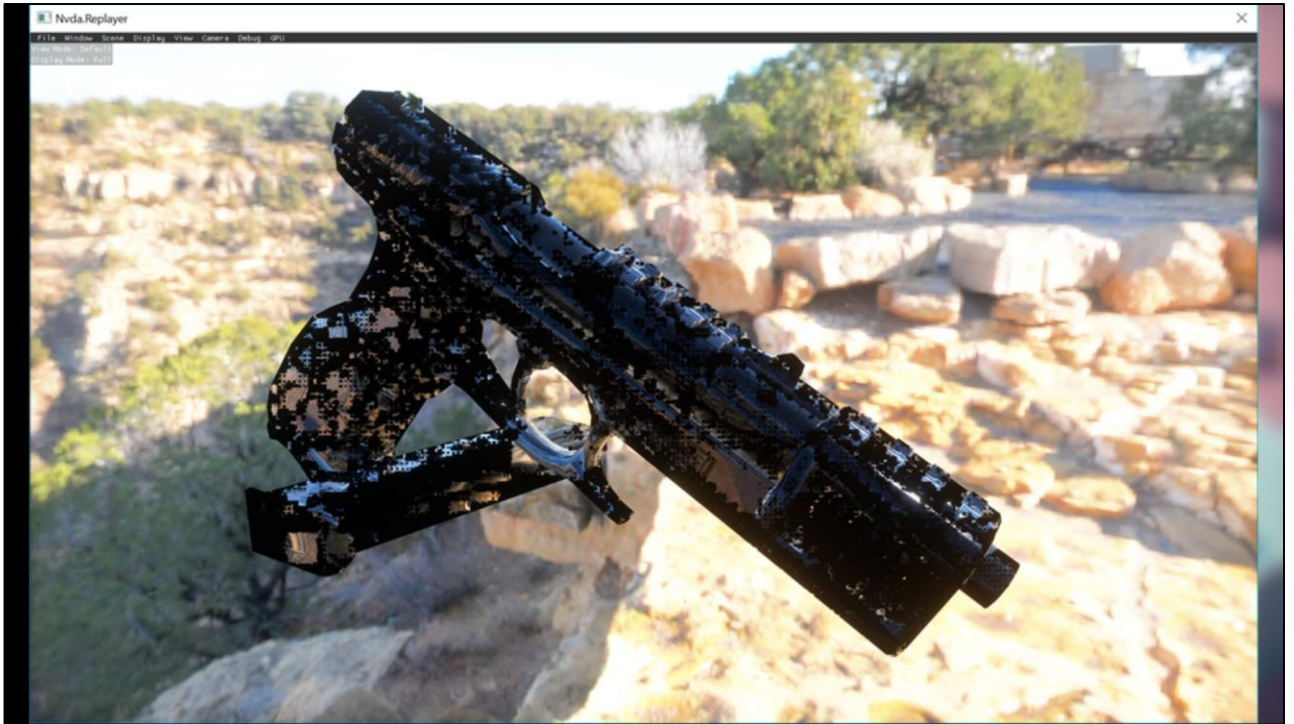
API Call	Count	Avg CPU ms	Σ CPU ms	Avg GPU ms	Σ GPU ms
vkQueueSubmit()	30	0.10	3.00	0.00	0.00
vkCreateBufferView()	4	0.10	0.3[illegible]	0.00	0.00
vkAllocateDescriptorSets()	41	0.06	2.48	0.00	0.00
vkResetCommandPool()	1	0.02	0.02	0.00	0.00
vkCreateImageView()	184	0.02	3.44	0.00	0.00
vkWaitForFences()	2	<0.01	0.01	0.00	0.00
vkCreateRenderPass()	13	<0.01	0.03	0.00	0.00
vkAcquireNextImageKHR()	1	<0.01	<0.01	0.00	0.00
vkResetFences()	2	<0.01	<0.01	0.00	0.00
vkMapMemory()	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
vkUnmapMemory()	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()	184	<0.01	<0.01	0.00	0.00
vkCmdBindDescriptorSets()	4,143	0.00	0.00	0.00	0.00
vkCmdBindPipeline()	1,839	0.00	0.00	0.00	0.00
vkCmdBindIndexBuffer()	1,799	0.00	0.00	0.00	0.00
vkCmdBindVertexBuffers()	1,799	0.00	0.00	0.00	0.00
vkCmdDrawIndexed()	1,799	0.00	0.00	<0.01	11.48
vkCmdPipelineBarrier()	616	0.00	0.00	0.00	0.00
vkCmdUpdateBuffer()	38	0.00	0.00	0.00	0.00
vkCmdDispatch()	38	0.00	0.00	0.19	7.15
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	3.63	7.26
vkCmdWriteTimestamp()	2	0.00	0.00	0.00	0.00
vkCmdCopyImage()	1	0.00	0.00	0.00	0.00
vkCmdCopyQueryPoolResults()	1	0.00	0.00	0.00	0.00
vkCmdResetQueryPool()	1	0.00	0.00	0.00	0.00
vkCmdBlitImage()	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.

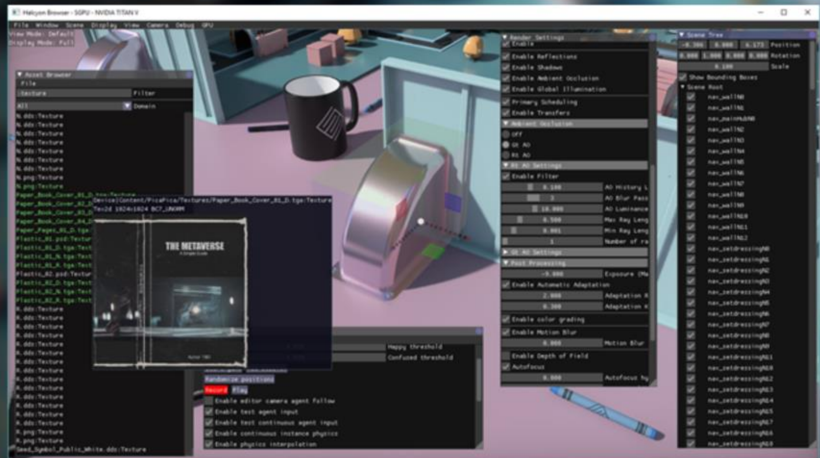
Another awesome feature with Nsight is the ability to export a capture as a standalone C++ application. Nsight will write out binary blobs of your resources in the capture, and write out source code issues your API calls. You can build this app and debug problems in an isolated solution.



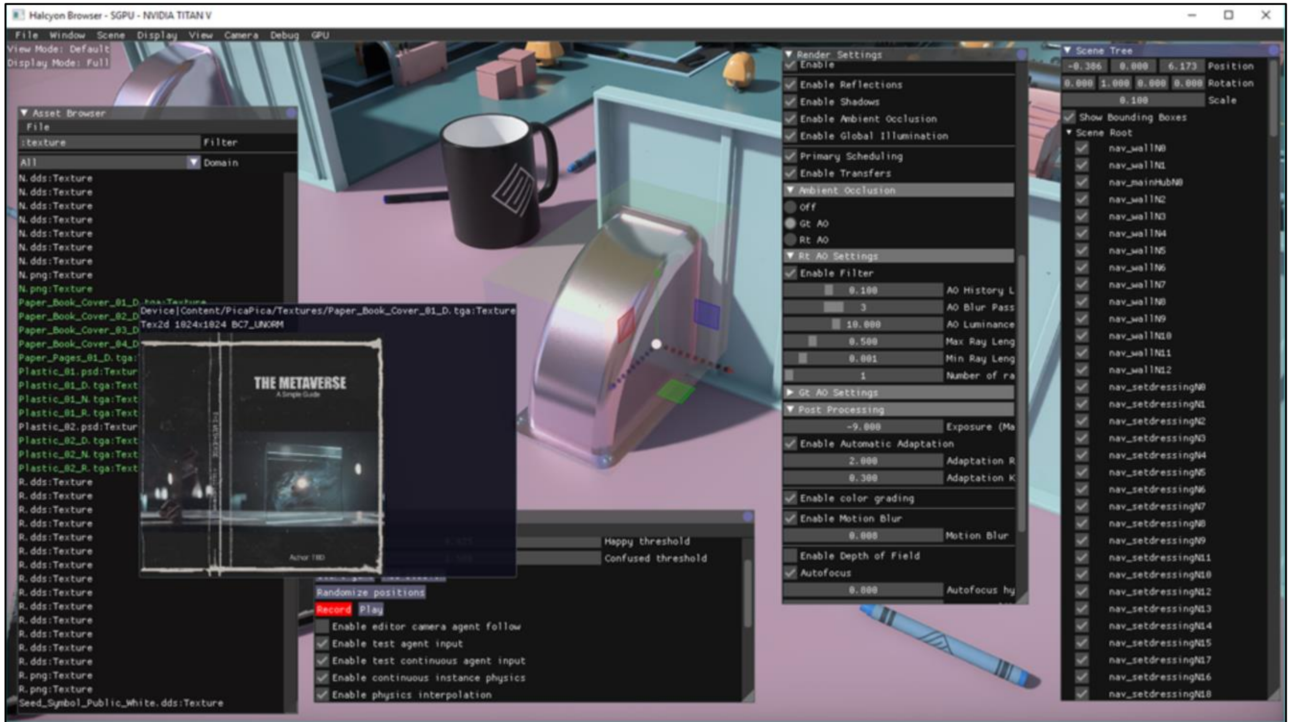
Here is our bug being reproduced in the standalone C++ export

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



Halcyon Browser - SGPU - NVIDIA TITAN V
File Window Scene Display View Camera Debug GPU
View Mode: Default
Display Mode: Full
Asset Browser
File
:texture
Filter
All
Domain
N.dds:Texture
N.png:Texture
Paper_Book_Cover_01_D.tga:Texture
Paper_Pages_01_D.tga
Plastic_01.psd:Textur
Plastic_01_D.tga:Text
Plastic_01_N.tga:Text
Plastic_01_R.tga:Text
Plastic_02.psd:Textur
Plastic_02_D.tga:Text
Plastic_02_N.tga:Text
Plastic_02_R.tga:Text
R.dds:Texture
R.png:Texture
Seed_Symbol_Public_White.dds:Texture
Device|Content/PicaPica/Textures/Paper_Book_Cover_01_D.tga:Texture
Tex2d 1024x1024 BC7_UNORM
THE METAVERSE
A Simple Guide
Happy threshold
Confused threshold
Randomize positions
Record
Play
Enable editor camera agent follow
Enable test agent input
Enable test continuous agent input
Enable continuous instance physics
Enable physics interpolation
Render Settings
Enable
Enable Reflections
Enable Shadows
Enable Ambient Occlusion
Enable Global Illumination
Primary Scheduling
Enable Transfers
Ambient Occlusion
Off
Gt AO
Rt AO
Rt AO Settings
Enable Filter
0.100 AO History L
3 AO Blur Pass
10.000 AO Luminance
0.500 Max Ray Leng
0.001 Min Ray Leng
1 Number of ra
Gt AO Settings
Post Processing
-9.000 Exposure (Ma
Enable Automatic Adaptation
2.000 Adaptation R
0.300 Adaptation K
Enable color grading
Enable Motion Blur
0.000 Motion Blur
Enable Depth of Field
Autofocus
0.000 Autofocus hy
Scene Tree
-0.386 0.000 6.173 Position
0.000 1.000 0.000 0.000 Rotation
0.100 Scale
Show Bounding Boxes
Scene Root
nav_wallN0
nav_wallN1
nav_mainHubN0
nav_wallN2
nav_wallN3
nav_wallN4
nav_wallN5
nav_wallN6
nav_wallN7
nav_wallN8
nav_wallN9
nav_wallN10
nav_wallN11
nav_wallN12
nav_setdressingN0
nav_setdressingN1
nav_setdressingN2
nav_setdressingN3
nav_setdressingN4
nav_setdressingN5
nav_setdressingN6
nav_setdressingN7
nav_setdressingN8
nav_setdressingN9
nav_setdressingN11
nav_setdressingN10
nav_setdressingN12
nav_setdressingN13
nav_setdressingN14
nav_setdressingN15
nav_setdressingN17
nav_setdressingN16
nav_setdressingN18

References

- **[Wihlidal 2018]** Graham Wihlidal, Colin Barré-Brisebois. "Modern Graphics Abstractions & Real-Time Ray Tracing" .
  available online
- **[Wihlidal 2018]** Graham Wihlidal. "Halcyon + Vulkan".
  available online
- **[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 Sjöö 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

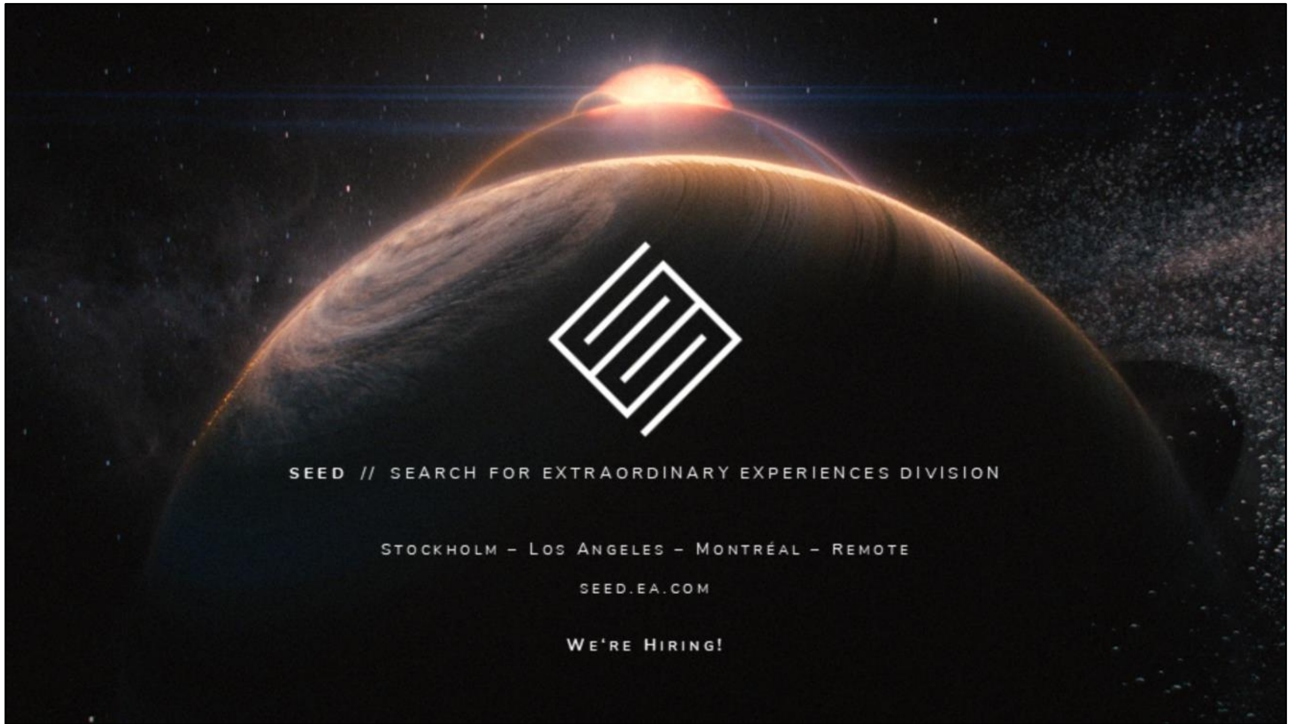
Thanks

- 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
- Tomasz Stachowiak

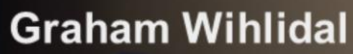
Thanks

- 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 Sjöö
  - 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 us a shout!



Graham Wihlidal
graham@ea.com
@gwihlidal

Questions?