Halcyon
Rapid Innovation using Modern Graphics

Graham Wihlidal
SEED – Electronic Arts
Disruptive Technology?

- Technology that significantly alters the way your business operates.

- Often forces companies to change their business for fear of losing market share or becoming irrelevant.
Embrace Disruption

- Maintain existing business
  - Safe bets, manage risk
  - Also important!

- Avoid “horse blinders”
  - Evolving market
  - Evolving users
  - Evolving technology
Embrace Disruption

- Innovation is important!
- Don’t be Kodak!
  - Invent the digital camera
  - Fear of disruption
  "That’s cute - but don’t tell anyone about it." [Mui 2012]
- Competition wins
Game Development

- Typically, **large engines** have...
  - Complex build systems
  - Inter-connected dependencies
  - Opinionated and rigid APIs
  - Specialized systems
  - Steep learning curve
Why?

- For good reason!
- Ship great games
- Performance > flexibility
But...

- Constrains agile prototyping
- Hard to rapidly pivot architecture
  - New rendering engine?
  - No more triangles?
  - Emerging platforms?
    - Cloud, VR, AR, MR, XR, Mobile, Social, ...
  - ...
- High risk, requiring large investment
Solution!

- Agile R&D engine
- Prove out technology
  - Scout ahead
  - Share findings
  - Mitigate risk
  - Guide adoption
Comparison

Game Engine

R&D Engine
Comparison

Game Engine

R&D Engine
Halcyon

- Rapid prototyping engine
- Windows, Linux, macOS
- Different purpose than Frostbite
  - Fast experimentation vs. AAA games
“PICA PICA”

- **Exploratory mini-game & world**
  [Andersson 2018] [Harmer 2018] [Opara 2018]

- **Built with Halcyon**

- **Goals**
  - Hybrid rendering
  - Clean and consistent visuals
  - Self-learning AI agents
  - Procedural worlds
  - No precomputation
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
Halcyon Goals

- Only target modern APIs
  - Vulkan 1.1
  - Direct3D 12
  - Metal 2

- Multi-GPU
  - Explicit heterogeneous mGPU
  - No AFR / No linked adapters
Halcyon Goals

- **Scalable computation**
  - All cores in a workstation
  - Multiple graphics adapters (mGPU)
  - Local cluster
  - Google Cloud Platform

- Under a **single abstraction**
  - Easy, right?
Halcyon Goals

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

Deferred Shading
(prompt: raster)

Direct Shadows
(prompt: ray trace or raster)

Direct Lighting
(prompt: compute)

Reflections
(prompt: ray trace or compute)

Global Illumination
(prompt: ray trace)

Ambient Occlusion
(prompt: ray trace or compute)

Transparency & Translucency
(prompt: ray trace)

Post Processing
(prompt: compute)
Hybrid Rendering
Guiding Principles

- No concept of a classic “frame”
- Rendering occurs at variable frequency
- No promise that data is locally resident
  - Design for latency
  - Design for massive data
Guiding Principles

▪ “Separation of Concerns”
  ▪ Design for a single purpose
  ▪ No monolithic systems

▪ Verbosity hidden with layers
  ▪ Explicit low-level API
  ▪ Convenience APIs on top
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 Backend

- Provides debugging and profiling
  - RenderDoc integration, validation layers, ...

- **Create and destroy render devices**
Render Backend

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

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

S E E D // Halcyon: Rapid Innovation using Modern Graphics
Render Backend

- **Vulkan 1.1**
  - Sub-groups
  - Descriptor indexing
  - External memory
  - Multi-draw indirect
  - NV Ray tracing
  - ...

...
Render Backend

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

SEED // Halcyon: Rapid Innovation using Modern Graphics
Render Backend

- Proxy
  - Discussed later in the presentation
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*
Render Device

- **Abstraction of a logical GPU adapter**
  - e.g. VkDevice, ID3D12Device, ...

- **Provides interface to GPU queues**

- **Command list submission**
Render Device

- **Ownership of GPU resources**
  - Create & Destroy

- **Lifetime tracking** of resources

- Mapping render handles \(\rightarrow\) device resources
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
### Render Handles

- Handles allow **one-to-many cardinality** \([\text{handle} \rightarrow \text{devices}]\)
- Each device can have a unique representation of the handle
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
Render Handles

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

```
ID3D12Resource
```

```
DX12: Adapter 0
```
```
DX12: Adapter 1
```
```
DX12: Adapter 2
```
```
DX12: Adapter 3
```
Render Handles

- Can also **mix and match backends in the same process!**
  - Makes debugging new backends much easier
  - **DX12 on left half of screen, VK on right half of screen**
Render Commands
## Render Commands

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

---

"SEED // Halcyon: Rapid Innovation using Modern Graphics"
Render Commands

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

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

    uint32 dispatchX = 0;
    uint32 dispatchY = 0;
    uint32 dispatchZ = 0;
};
```
Render Command List

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

- Translate commands
- Read command list
- Write Vulkan API
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;
}
bool RenderCompileContextVulkan::compileCommand(const RenderCommandBeginTiming& command)
{
    auto timingHeap = 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;
}
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;
}
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
  - ...

S E E D // Halcyon: Rapid Innovation using Modern Graphics
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
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

- Composition of multiple graphs at varying frequencies
  - e.g. translucency, refraction, 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
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);
    };
});
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
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
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
Render Graph

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

```cpp
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
);
```
Lots of fun coordinate snapping bugs

i.e. 3 GPUs partitioned to 0.33333...
Lots of fun coordinate snapping bugs

16 GPUs! (because, why not?)
Render Graph

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

Requested Per Pass $\rightarrow$

```c
void setSchedule(RenderGraphSchedule passSchedule);
void setTransfer(
    RenderGraphResource resource,
    RenderTransferPartition src,
    RenderTransferPartition dst,
    RenderTransferFilter dstFilter);
```
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 →

```c
void setSchedule(RenderGraphSchedule passSchedule);
void setTransfer(
  RenderGraphResource resource,
  RenderTransferPartition src,
  RenderTransferPartition dst,
  RenderTransferFilter dstFilter);
```
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*)
```cpp
renderGraph.addPassCallback("Shadow Mask Pack", [&](RenderGraphBuilder& 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.createServerShaderViews(srvs, uavs)) },
            dispatchWidth, desc.height
        );
    });
});
```
Some bugs were obvious
Some bugs were obvious
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
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
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

```c
struct RenderGraphAreaLight {
  RenderGraphResource triangleLightList;
  uint32 triangleCount;
};
```
Render Graph

- Lookup by **type**
  - `scope.get<T>() -> &T`

- Parameters in “plain old data” structs
  - `RenderGraphResource, RenderHandle`
  - `float, int, mat4, etc.`

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

```cpp
{  
gbuffer <- render_opaque()  
gbuffer <- render_decals(gbuffer)  
{  
    gbuffer <- render_opaque()  
    render_lighting(gbuffer)  
} -> envmap  
apply_envmap(gbuffer, envmap)  
}
```
Render Graph DSL

- Experimental
- Macro Magic
Render Graph

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

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

- Build **device indirection table**
- Virtual device index $\rightarrow$ adapter index
Virtual Multi-GPU

- Create multiple instances of a device
- Virtual GPUs execute sequentially (WDDM)
Virtual Multi-GPU

- Increases overall wall time (don’t use for profiling)
- Amazing for development and testing!
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!
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.. 😊)
Render Proxy
Render Proxy

- Remote render backend
- Any API / 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
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. NV ray tracing on a MacBook 😊
Render Proxy

- The possibilities are endless!
Machine Learning

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

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

100% Rust!
Asset Pipelines

- Everything is **content addressable**
  - Hash of data is the identity
  - Sha256

- **Merkle trees!**
  - Dependency evaluation
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
Asset Pipelines

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

S E E D // Halcyon: Rapid Innovation using Modern Graphics

Objects with Multi-Layered Materials
Shaders

- Exclusively HLSL
  - Shader Model 6.X
- Majority are compute shaders

- Performance is critical
  - Group shared memory
  - Wave-ops / Sub-groups
Shaders

- No reflection
  - Avoid costly lookups
  - Only explicit bindings
  - ... except for validation

- Extensive use of HLSL spaces
  - Updates at varying frequency

- Bindless
Shaders

- HLSL
  - DXC
    - DXIL
      - Direct3D 12
    - SPIR-V
      - Vulkan 1.1
  - SPIRV-CROSS
    - MSL
      - Metal 2
    - ISPC
      - AVX2, ...
Shader Arguments

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

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

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

- Each argument contains:
  - “ShaderViews” handle
  - Constant buffer handle and offset
- “ShaderViews”
  - Collection of SRV and UAV handles
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
Texture2D<float4> g_albedo
Texture2D<float4> g_normal
Texture2D<float4> g_roughness
Texture2D<float4> g_metalness
Texture2D<float4> g_ao
Texture2D<float4> g_emissive
Texture2DArray<float4> g_translucency

StructuredBuffer<uint3> g_indexBuffer
StructuredBuffer<VtxInputPosition> g_vbPositions
StructuredBuffer<VtxInputTangentSpace> g_vbTangentSpace
StructuredBuffer<VtxInputTexCoordAndColor> g_vbTexCoordAndColor

ConstantBuffer<GeometryConstants> g_geometry
ConstantBuffer<MaterialConstants> g_material

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

- Shader compilation (HLSL → SPIR-V)
  - Patch SPIR-V to match DX12
  - Using spirv-reflect from Hai and Cort

- `spvReflectCreateShaderModule`
- `spvReflectEnumerateDescriptorSets`
- `spvReflectChangeDescriptorBindingNumbers`
- `spvReflectGetCodeSize / spvReflectGetCode`
- `spvReflectDestroyShaderModule`
SPIR-V Patching

- **SPV_REFLECTRESOURCE_FLAG_SRV**
  - Offset += 1000

- **SPV_REFLECTRESOURCE_FLAG_SAMPLER**
  - Offset += 2000

- **SPV_REFLECTRESOURCE_FLAG_UAV**
  - Offset += 3000
SPIR-V Patching

- **SPV_REFLECTRESOURCE_FLAG_CBV**
  - Offset Unchanged: 0
  - Descriptor Set += MAX_SHADER_ARGUMENTS

- CBVs move to their own descriptor sets
  - ShaderViews become persistent and immutable
SPIR-V Patching

- If 2 of 4 HLSL spaces in use:

<table>
<thead>
<tr>
<th>Set 0</th>
<th>SRVs (&gt;=1000)</th>
<th>Samplers (&gt;=2000)</th>
<th>UAVs (&gt;=3000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>SRVs (&gt;=1000)</td>
<td>Samplers (&gt;=2000)</td>
<td>UAVs (&gt;=3000)</td>
</tr>
<tr>
<td>Set 2</td>
<td>Unbound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>Unbound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 4</td>
<td></td>
<td>Dynamic Constant Buffer (Offset: 0)</td>
<td></td>
</tr>
<tr>
<td>Set 5</td>
<td></td>
<td>Dynamic Constant Buffer (Offset: 0)</td>
<td></td>
</tr>
</tbody>
</table>
Tools
Tools

- RenderDoc
- NV Nsight
- AMD RGP
## Tools

![API Statistics View](image)

### Summary

- **Draws**: 1801
- **Dispatches**: 38
- **Clears**: 27
- **Bits**: 1
- **Presents**: 1
- **Command List Executes**: 30
- **Signals**: 0
- **Waits**: 0
- **Misc. Data Updates**: 2
- **Non-API**: 29
- **Other**: 10995
- **Total**: 12684

### Details

<table>
<thead>
<tr>
<th>API Call</th>
<th>Count</th>
<th>Avg CPU ms</th>
<th>I CPU ms</th>
<th>Avg GPU ms</th>
<th>I GPU ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>vkCreateSubmmit</td>
<td>30</td>
<td>0.00</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateBufferView</td>
<td>4</td>
<td>0.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkAllocaterDescriptorSets</td>
<td>41</td>
<td>0.00</td>
<td>0.30</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkResetCommandPool</td>
<td>1</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateImageView</td>
<td>1648</td>
<td>0.00</td>
<td>0.49</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkFlushForFence</td>
<td>2</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateRenderPass</td>
<td>13</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkAcquireNextImageMemory</td>
<td>1</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkResetFences</td>
<td>2</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateDescriptorPool</td>
<td>41</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkUploadDescriptorSets</td>
<td>41</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateFramebuffer</td>
<td>13</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCreateMemory</td>
<td>1</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkDestroyRenderPass</td>
<td>13</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkDestroyFramebuffer</td>
<td>13</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkDestroyDescriptorPool</td>
<td>41</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkDestroyImageView</td>
<td>1048</td>
<td>-0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBindDescriptorSets</td>
<td>5143</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBindPipeline</td>
<td>1.030</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBindIndexBuffer</td>
<td>1.050</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBindVertexBuffers</td>
<td>1.050</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdDrawIndexed</td>
<td>1.793</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdPipeLineBarrier</td>
<td>61</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdDispatch</td>
<td>38</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdClearBuffer</td>
<td>28</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBeginRenderPass</td>
<td>27</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdEndRenderPass</td>
<td>27</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdSetScissor</td>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdSetTexture</td>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdSetVertexBuffer</td>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdDraw</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdWriteTimestamp</td>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdCopyImage</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdCopyImageToTex</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdCopyQueryPoolResults</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdCopyQueryPool</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkCmdBlitImage</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vkQueuePresentKHR</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Tools

- C++ Export!
- Standalone
Dear ImGui + ImGuizmo

- Live tweaking
- Very useful!
References

Thank You

- Reboot Develop Blue & Khronos
- Halcyon Team @ SEED
  - Andrew Lauritzen
  - Colin Barré-Brisebois
  - Jasper Bekkers
  - Henrik Halén
  - Carlos Gonzalez-Ochoa
  - Graham Wihlidal