SIGGRAPH 2026
Los Angeles 19–23 JUL

KHRONOS GROUP

Vulkan

GFXReconstruct - an Engine for Vulkan Debugging and Profiling

Brad Grantham, LunarG
Director of Technology

Brad Grantham, LunarG, Inc.

Who am I?

- GFXReconstruct Architecture Lead
- Silicon Graphics, AMD, and Arm before joining LunarG in 2019
- Still use “vi” way too often

Agenda

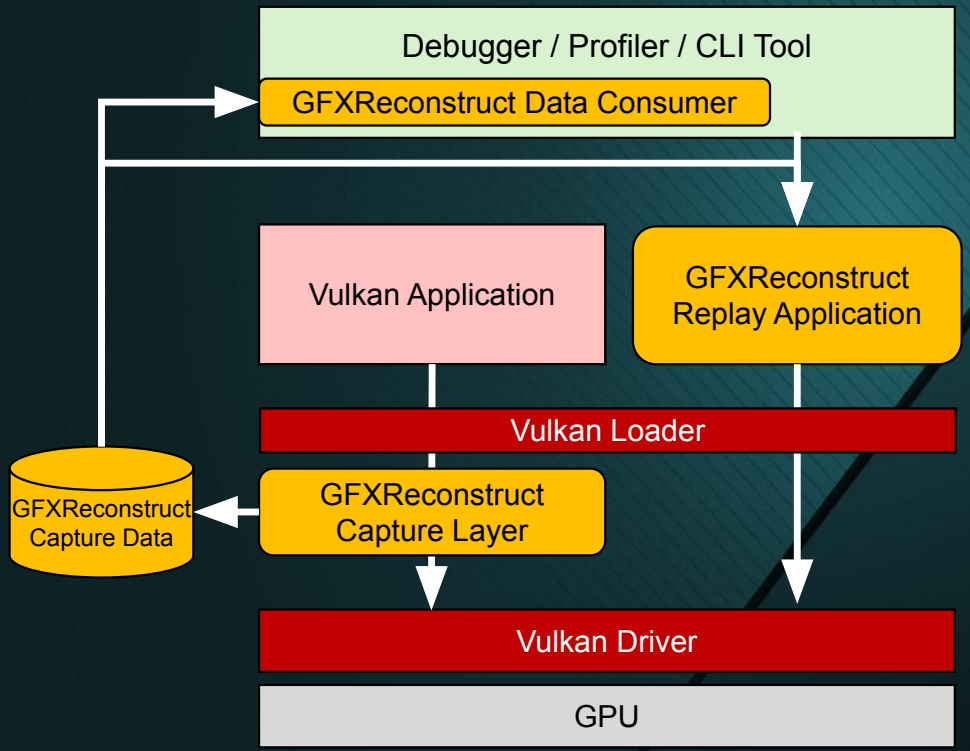
Introduction	–	What is GFXReconstruct and who builds on it?
Pillars	–	Fidelity, Integrity, Portability, Performance
Capturing	–	Produce a Record of the Application's API Usage
Analyzing	–	Examine Transactions at the API Boundary
Replaying	–	Get Results from the GPU

Who Is This Talk For?

And what you should leave with

- You ship Vulkan applications:
  - Capture a bug once – hand off a small reproducer
- You build GPUs, drivers, or platforms:
  - Replay real workloads without the app, across revisions
- You build tools:
  - Capture app contents, analyze the data, replay on a GPU
- The takeaway: capture once then process the capture

What is GFXReconstruct?



What is GFXReconstruct?

GFXReconstruct (aka “GFXR”) lets a developer analyze and replay an application’s captured graphics commands after a program has run.

- Capture an application’s graphics commands to a file
- Analyze and transform those commands
- Replay those commands at any time without the application

LUNARG

Powering the Industry's Top Analysis Tools

SAMSUNG
Sokatoa

arm
Frame Advisor

Google
Android Performance Analyzer

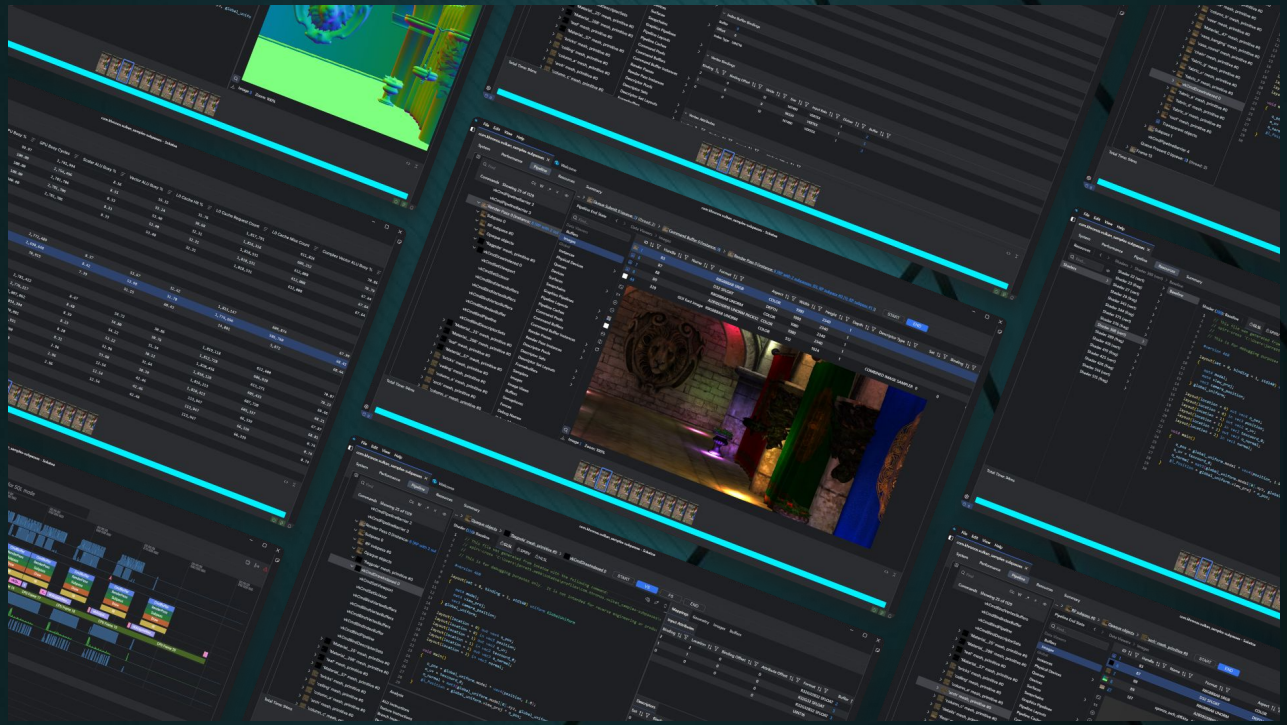
Qualcomm
Snapdragon Profiler

KHRONOS GROUP

LUNARG

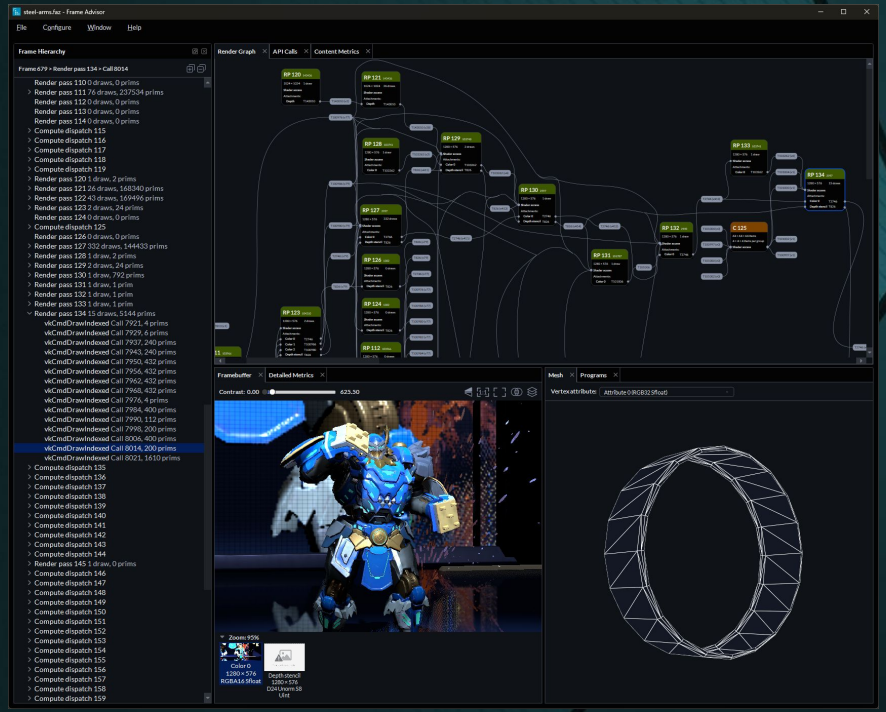
SAMSUNG
Sokatoa

Sokatoa profiler and debugger tool for Android
from Samsung



arm
Frame Advisor

Frame Advisor in Arm Performance Studio

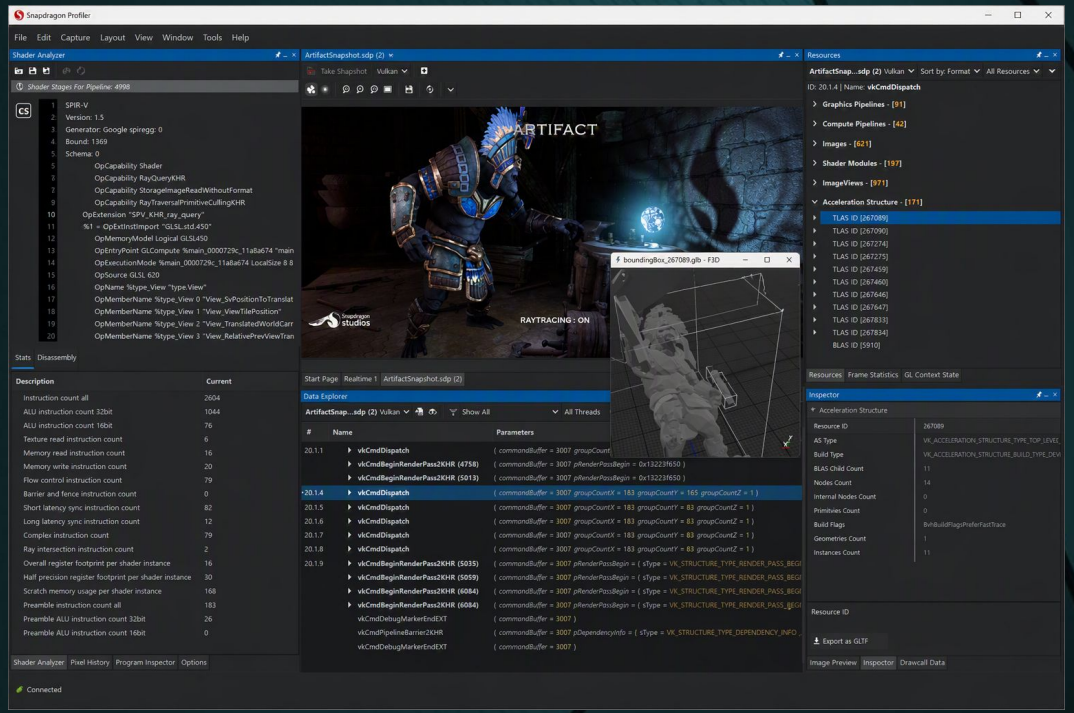


Google
Android Performance Analyzer

Android Performance Analyzer (APA) is Google's new performance tool for Android, co-developed with Samsung's SARC and LunarG.

Qualcomm
Snapdragon Profiler

Qualcomm Snapdragon Profiler



GFXReconstruct Components

Insertable layers for capture produce a “capture file”

- Multiple frames or short frame ranges (maybe just one)

Process captured data using a C++ Consumer implementation

- Subclass `gfxrecon::decode::VulkanConsumer`

Replay captured data on device

- Instrument for performance analysis
- Gather intermediate images and data

Introduction – Demo

Capture Vulkan Program

Replay application graphics workload

Design Principles

FIDELITY	INTEGRITY	PORTABILITY	PERFORMANCE
Capture & playback on the same device, with identical results	Optimizations that stay true to application behavior	Playback across a broad range of devices, with variable fidelity	Deliver the performance required for usability & interactivity

➔ *Trade offs are often required, but they should be evaluated against these principles*

Principles – Fidelity and Integrity

GFXR captures all information crossing the API boundary

(May also capture additional information required for replay)

- API function or method calls : objects, parameters, return values
- CPU writes to API–visible memory at sync points
- Status of API and OS synchronization primitives when those primitives represent forward progress (enforced in replay)

Principles – Fidelity and Integrity

Some things GFXR does *not* capture:

- Timestamps
  - A candidate for annotation for debugging but are unreliable as information about concurrency or synchronization
- Other OS primitives (system calls, library functions)
  - Considered out of scope but another candidate for annotation
- GPU state not returning to CPU across the API
  - Most intermediate GPU buffer results *during capture* (more later)

Principle – Portability

Priority is replay on same implementation

- OS, driver version, GPU model

Replay on other implementations is very valuable

- Capture queries additional information so that portable replay may be possible
- `gfxrecon-replay` offers options to assist with replay on other implementations
- Works to varying degrees – not every capture can made portable

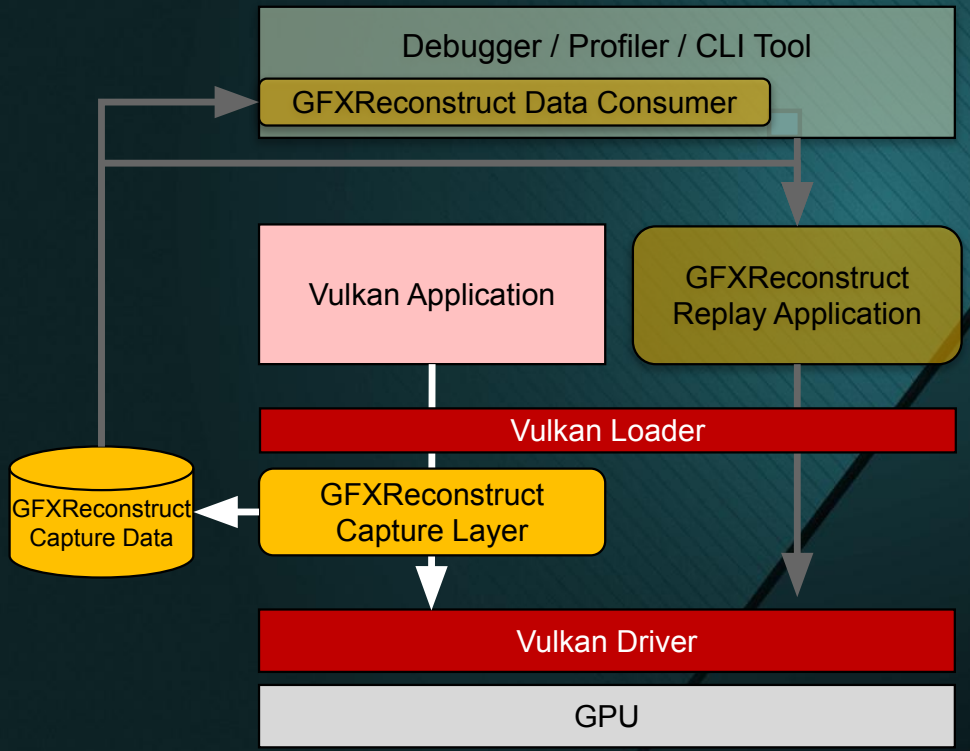
See documentation in our repository for more

Principle – Performance

High performance, versioned binary format

- Backward compatible
- Block-based
- Compressed on write, decompressed just-in-time

Capture Once, Analyze the Capture



Capturing API Content

Vulkan API layer “libVkLayer_gfxreconstruct.so” or .dll

On desktop use the Vulkan Configurator, our Python convenience script, or set the Vulkan environment variables

On Android:

- Need a debuggable app or a userdebug Android build
- Android settings to insert the capture layer
- See documentation in our source code repository

Records all core 1.4 Vulkan function calls and many extensions

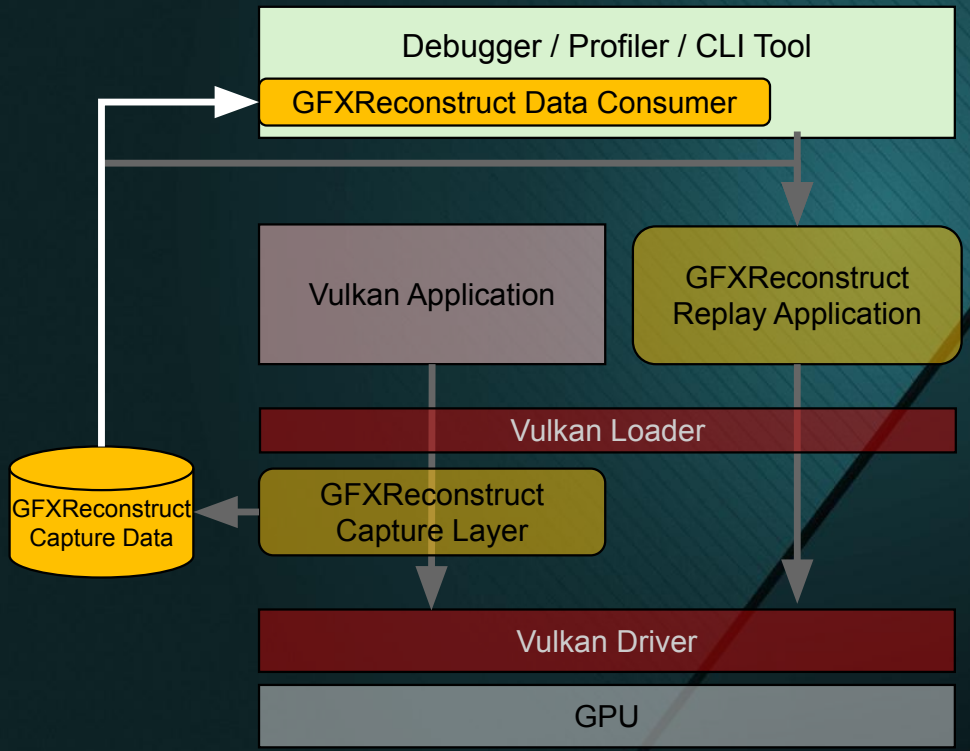
Capturing API Content

Capture can be limited to frame ranges (“trimmed”)

- “State setup” blocks set up the GPU as if prior frames ran
- Specify a static range before start (e.g. for automation)
- Or can trigger during run

Many more options are described in the USAGE guides

Analyzing The Capture



Analyzing The Capture

gfxrecon.py info

Display useful information about a capture

- Compression
- Frames
- App info
- Device info

```
$ gfxrecon.py info dogwalk.gfxr
Exe info:
      Application exe name: ./dogwalk.x86_64
      Application version: 0.0.0.0
      Application Company name:
      Product name:

File info:
      Compression format: LZ4
      Total frames: 1030

Vulkan application info:
      Application name:    DOGWALK
      Application version: 0
      Engine name:         Godot Engine
      Engine version:      16793601
      Target API version:  4202496 1.2.0
      Used resolutions:    1920x1080 1920x1048

Vulkan physical device info:
      Device name:         [...]
```

Analyzing The Capture

```
$ gfxrecon.py info --verbose /d/tmp/dogwalk.gfxr
{
    "detected-apis": [
        "Vulkan"
    ],
    "exe": {
        …
        "name": "./dogwalk.x86_64",
    },
    …
    "general-info": {
        "compression": {
            "format": "LZ4"
        },
        "frames": {
            "actual-count": 663,
            [...]
        }
    },
    …
    "vulkan": {
        "header-version": "1.4.350",
        "instances": [
            { …
```

Analyzing The Capture

gfxrecon.py convert

Produce JSON or JSON Lines for a capture file

- Query with favorite JSON lib or tool
  - Average draws per frame? Histogram of buffer uploads? Missing barriers?
- JSON Lines is smaller and streamable and greppable

One file for the whole capture or a file per frame

Optionally dump out BLOBs (shaders, pipeline caches, CPU writes to mapped memory, push constants…)

Analyzing The Capture

C++ Consumer Implementation

- FileProcessor -> Decoder -> Consumer pipeline
  - Handles file format, compatibility
  - Generated from API specification, updated regularly

- You subclass `gfxrecon::decode::VulkanConsumer`
  - A virtual method per API call, with decoded arguments as parameters
  - Your code sees every transaction at the API boundary
  - Industry's analysis tools use this interface

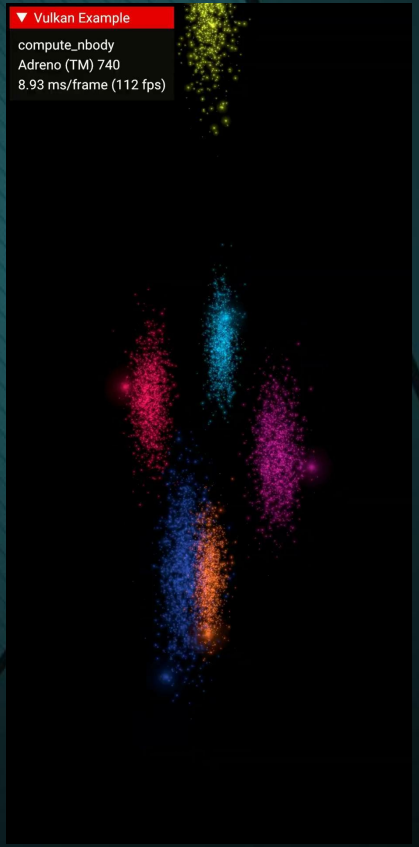
Analyzing The Capture

```
virtual void Process_vkCmdDraw(
    const ApiCallInfo& call_info,
    args::CmdDraw&     args) override;

struct CmdDraw {
    format::HandleId commandBuffer;
    uint32_t vertexCount;
    uint32_t instanceCount;
    uint32_t firstVertex;
    uint32_t firstInstance;

    auto GetTuple() const { return std::tie(...); }
};
```

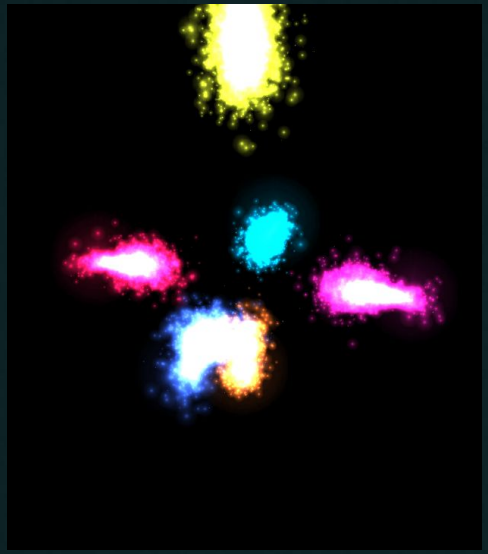
Case Study – Compute Particle System

In automated regression testing screenshots differed by a few %...

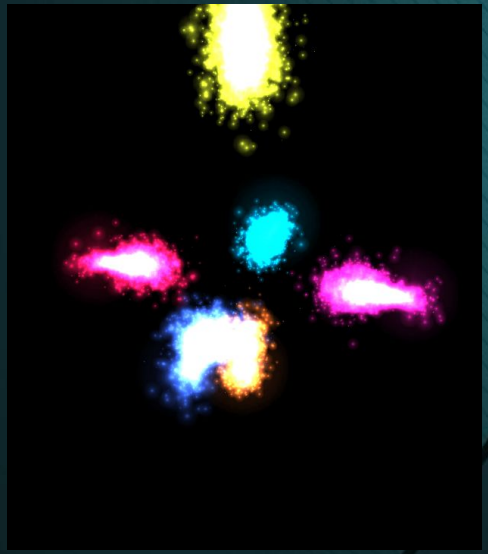
Sometimes passing the threshold into failure!



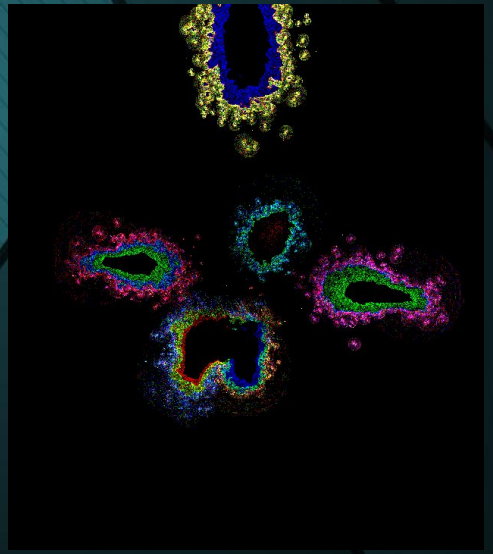
Case Study – Compute Particle System



Replay Frame 40



Next Replay Frame 40



Scaled Difference

Case Study – Compute Particle System

- Different images every replay – shifted particle clouds
- Small and roughly constant across frames
- Does the original app have an issue?

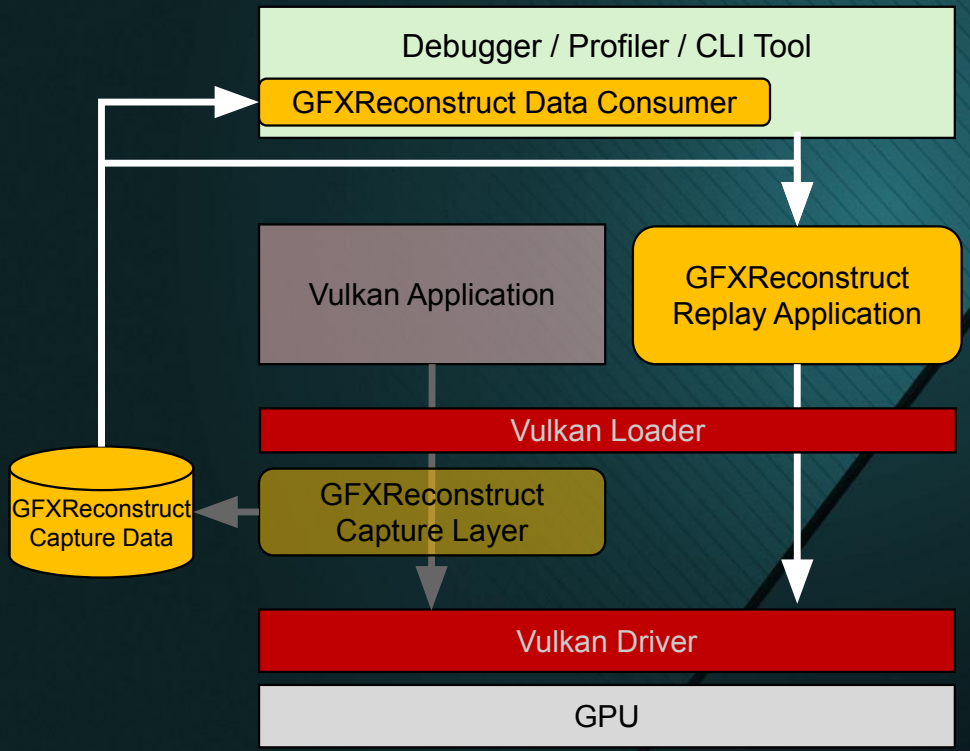
The Capture Contains the Evidence

```
vkQueueWaitIdle()
vkQueueSubmit(compute) -> wait on graphics_sema, signal compute_sema
…
FillMemoryCommand  // host writes deltaT for compute job
vkQueueSubmit(graphics) -> wait on compute_sema, signal graphics_sema+present_sema
vkQueuePresentKHR -> wait on present_sema
```

- `gfxrecon.py replay --sync` -> differences vanish, deterministic
- Verdict: an application host-write race - captured faithfully!

➔ **Nondeterministic replay shows an application issue**

Replaying the Capture



Replaying the Capture

Make a smaller test case by replaying with “trimming”

```
GFXRECON_CAPTURE_FRAMES=1234 gfxrecon.py replay --capture …
```

Output capture contains:

- Collated “setup” commands
- All the commands in frame 1,234

Discard unused resources with `gfxrecon.py optimize`

- To reduce file size and replay time

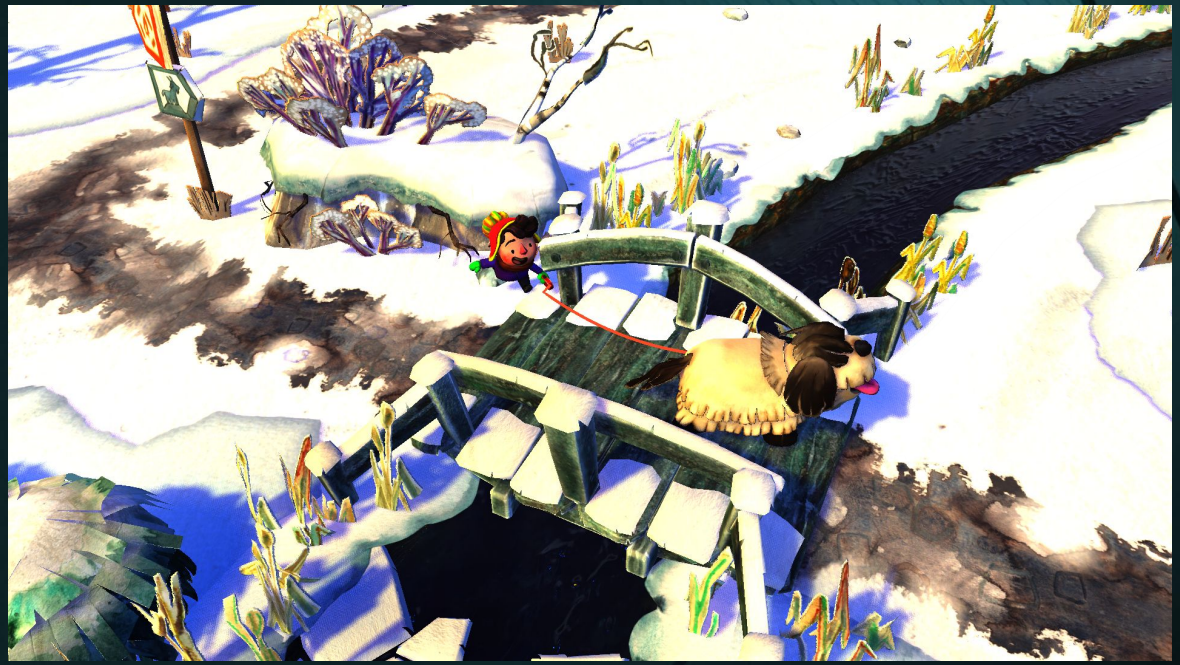
Replaying the Capture

Getting data within a frame: Replaying with “Dump Resources”

- JSON input specifies what to dump
  - Options for dump resources operation
  - Tuples of block numbers identify Draw/Dispatch/TraceRays uniquely
  - *Parse data up front (e.g. with Consumer) to know block indices*
- JSON output describes was was pulled from GPU
  - Each output resource
- Feature is resource intensive in memory and disk and time
  - *Caching and batching and downscaling may be necessary to get thumbnails*

Replaying the Capture

Dump resources: save intermediate images and data between draws



Replaying the Capture

Dump resources: save intermediate images and data between draws

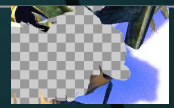
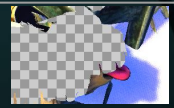
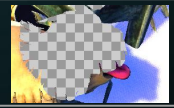
Replaying the Capture

Area of Interest

Replaying the Capture

Input JSON

```
{
    "DumpResourcesOptions": {
        "ImageFormat": "png",
        "Scale": 0.25
    },
    "BeginCommandBuffer": [
        57021
    ],
    "Draw": [
        [
            [...]
            67261,
            67267,
            67271,
            67277,
            67283,
            [...]
        ]
    ],
    "RenderPass": [
        [
            [
                66092,
                66105
            ],
            [...]
        ]
    ],
    "QueueSubmit": [
        67563
    ]
}
```

Output Images	
...	
raw_**67261**_qs_67563_bcb_57021….png	
raw_**67267**_qs_67563_bcb_57021….png	
raw_**67271**_qs_67563_bcb_57021….png	
raw_**67277**_qs_67563_bcb_57021….png	
raw_**67283**_qs_67563_bcb_57021….png	
...	

Replaying the Capture

Before

Replaying the Capture

After

Replaying the Capture

Shader Replacement

- Extract the shaders from a capture
  - `gfxrecon.py extract …`
- Generate source, e.g. `spirv-cross`
- Edit or rewrite shaders
- Replay with the replacements – no re-capture needed
  - `gfxrecon.py replay --replace-shaders …`

Replaying the Capture

Shader Replacement

Captured Shaders

Replaying the Capture

Shader Replacement

```
// vec3 param_103 = directional_lights.data[i_5].color *
   Directional_lights.data[i_5].energy;

vec3 blue = vec3(0.0, 0.0, 1.0);
vec3 param_103 = blue;
```

Replaying the Capture

Shader
Replacement

Replaced Shaders

Replaying the Capture

`gfxrecon-replay.exe` is just another program

- Replay into e.g. RenderDoc!
- Capture once with GFXR, analyze with anything

Summary

GFXReconstruct –

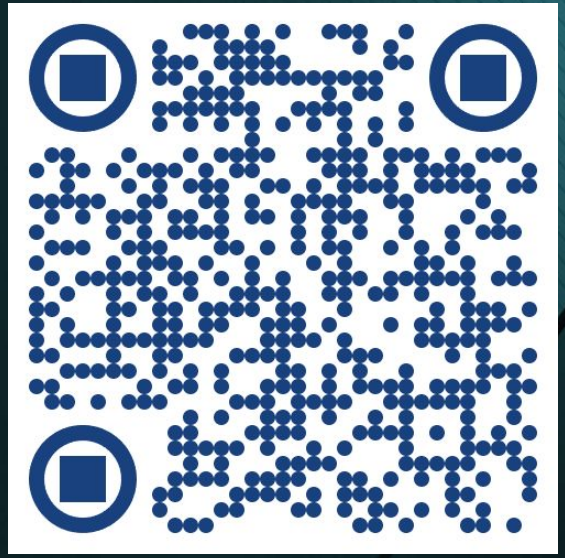
An Engine for Graphics API Debugging and Profiling

- Capture the API boundary faithfully
- Analyze it with your own Consumer
- Replay it to get intermediate data

Where To Get It

Source code:

- https://github.com/LunarG/gfxreconstruct



Thank you – KHRONOS GROUP

LUNARG

More Information

Website
www.lunarg.com

Presentations

Brad Grantham

Spencer Fricke

GFXReconstruct