Mastering GFXReconstruct

Parts 1–4

January 15, 2026

LunarG, Inc.

Introduction

This document contains formatted versions of LunarG blog posts describing how to use the popular GFXReconstruct tool that were published on LunarG's website starting in September 2025. This multi-part series was designed to give developers a comprehensive understanding of how to use GFXReconstruct and appreciate the value it brings to graphics development workflows.

Chapter 1:

A Powerful Open-SourceTool for Graphics API Capture and Replay

Welcome to our comprehensive overview of GFXReconstruct, an open-source tool designed to capture and replay graphics API calls. Whether you're a graphics developer debugging a complex rendering issue or optimizing performance for a new platform, GFXReconstruct is a powerful ally in your toolkit. In this chapter, we'll explore what GFXReconstruct is, its evolution, its licensing, and its core value propositions for developers. We'll also dive into its primary use cases and the APIs and operating systems it currently supports. By the end of this chapter, you'll have a clear understanding of how GFXReconstruct can enhance your graphics development workflow and why it's a critical tool in the graphics ecosystem.

Background

The concept of capturing and replaying graphics workloads has roots stretching back to the early days of interactive graphics. In the 1980s and '90s, Silicon Graphics (SGI) provided tools like GLdebug for its Iris Workstations running Iris GL. GLdebug could log every API call—along with parameters and warnings—and even generate replayable C code from a captured session.[1] Around the same time, the X Window System introduced the RECORD extension, enabling low-level capture and replay of X11 protocol events, including drawing commands.[2]

As real-time 3D graphics evolved, more sophisticated tools emerged: GLIntercept (~2002) provided the first open-source OpenGL call interceptor with basic replay[3], while PIX from Microsoft introduced robust Direct3D capture and replay on Xbox and Windows platforms. [4]

Graphics APIs like Vulkan and DirectX 12 are now the backbone of modern, high-performance graphics applications, enabling developers to harness the full power of GPUs. Debugging and optimizing these applications can be challenging due to the complexity of API calls and their interactions with hardware. Previous capture/replay tools successfully laid the groundwork for modern, cross-platform tools like RenderDoc and GFXReconstruct, which bring precision, automation, and open-source, cross-platform flexibility to graphics debugging and performance analysis.

GFXReconstruct is an open-source tool for capturing and replaying graphics API calls—primarily Vulkan and DirectX 12 (but support for other graphics APIs has already been demonstrated). It gives developers the ability to isolate, examine, and replay rendering workloads, enabling deep

insights into how an application uses the graphics stack. Under the permissive MIT License, it's freely available for modification and integration into your own tools and workflows.

GFXReconstruct's Place in the Ecosystem

Developed by LunarG in collaboration with partners like AMD's GPUOpen team, GFXReconstruct began as a Vulkan-focused tool aimed at improving the quality of Vulkan applications. Initially integrated into the Vulkan SDK in 2020 (version 1.2.141), it replaced earlier tools like Vktrace/Vkreplay, offering enhanced capture and replay capabilities. Over time, GFXReconstruct evolved to support DirectX 12 (D3D12) and DirectX Raytracing (DXR), announced in January 2023, reflecting its growing relevance in the Windows gaming ecosystem, where D3D12 is a dominant standard. This expansion demonstrates GFXReconstruct's adaptability to meet the needs of developers working across multiple graphics APIs.

Licensed under the MIT License, GFXReconstruct is freely available for use, modification, and contribution, fostering a collaborative community of developers who can enhance its functionality or tailor it to specific needs. Its open-source nature ensures accessibility and encourages contributions, such as bug fixes or support for additional APIs, making it a living project that evolves with the graphics industry. As part of the broader Vulkan ecosystem, with sponsorship from Valve and with contributions from industry leaders like AMD, GFXReconstruct plays a pivotal role in providing cross-platform, high-efficiency tools for graphics development.

Looking forward, while currently focused on Vulkan and DirectX 12, GFXReconstruct's API-agnostic container format can support additional APIs in the future, such as OpenXR or Metal, as community or industry needs arise. This extensibility positions GFXReconstruct as a versatile tool for the evolving graphics landscape.

Unlike full-featured GPU debuggers like RenderDoc or Nsight, GFXReconstruct is laser-focused on capture and replay—making it especially useful for automation and integration into regression testing systems, CI pipelines, and cross-platform validation.
Its modular CLI toolset allows for:

- **Capturing** workloads using Vulkan or DirectX.
- **Replaying** .gfxr files to reproduce visual results.
- **Converting** or **trimming** captures for performance and debugging.
- **Inspecting** captured data for analysis or tooling.

Understanding API Capture and Replay

API capture involves intercepting and logging the sequence of graphics API calls made by an application, creating a capture file that records the application's rendering commands and state.

Replay, on the other hand, allows developers to re-execute these captured commands on the same or different hardware, enabling analysis, debugging, or optimization without needing to run the original application.

Programming modern GPUs is inherently complex, especially when trying to understand what happens between your application's API calls and what eventually shows up on screen. Whether you're diagnosing a rendering bug, optimizing performance, or validating behavior across hardware, you often need to freeze time and replay a precise sequence of GPU commands.

At its core, GFXReconstruct operates like a graphics black box recorder. It intercepts API calls—such as Vulkan commands or DirectX 12 draw calls—and logs them into a capture file. That file can then be replayed independently of the original application. This provides a snapshot of your application's graphics behavior at a specific moment in time.

GFXReconstruct excels at this process, providing a robust framework for capturing and replaying API calls with precision, making it easier to diagnose issues, optimize performance, or test compatibility across platforms.

The benefit? Developers can:

- **Reproduce** rendering issues even without access to the full application.
- **Analyze** the API usage offline.
- **Test** across hardware platforms.
- **Profile** GPU workloads in isolation.

Capture and replay workflows are foundational in modern graphics debugging, and GFXReconstruct delivers a clean, deterministic approach to doing just that.

Use Cases for Developers

GFXReconstruct is a cornerstone tool for graphics developers, performance engineers, QA teams, and platform validation labs, offering powerful capabilities for debugging, optimization, and testing. Its versatility stems from four key design goals, each supporting critical use cases in GPU software development:

- **Fidelity**: Capture and playback on same device with identical results
- **Integrity**: Optimizations that stay true to application behavior
- **Portability**: Playback across a broad range of devices with variable fidelity
- **Performance**: Deliver the performance required for usability and interactivity.

Important GFXReconstruct use cases include:

- **Defect Reproduction and Debugging:** Debugging visual artifacts, crashes, or incorrect rendering in graphics applications can be challenging, especially when issues are intermittent or hardware-specific. GFXReconstruct captures a complete sequence of API calls, creating a reproducible test case that can be replayed on the same or different hardware. This enables developers to isolate defects by analyzing the exact application state at the point of failure without needing to recreate the issue in the live application. For bug reporting, capture files provide a standardized, reproducible format that improves communication with driver vendors or API maintainers, accelerating resolution times.
- **API Usage Analysis**: Understanding how an application interacts with graphics APIs like Vulkan or DirectX 12 is essential for ensuring correct and efficient implementations. GFXReconstruct enables developers to inspect API call frequencies, state changes, resource usage, and potential misuses. This analysis helps identify inefficiencies or errors, making it a powerful tool for optimizing API usage and ensuring compliance with best practices.
- **Performance Profiling and Optimization:** Optimizing graphics applications requires detailed insight into rendering performance. GFXReconstruct supports performance profiling by capturing API call traces that can be analyzed to pinpoint bottlenecks or inefficient resource allocation. Developers can use these insights to optimize critical code sections, ensuring better performance on specific hardware.
- **Regression Testing and Platform Bringup:** As graphics applications and drivers evolve, ensuring that updates don't introduce regressions is critical. GFXReconstruct supports regression testing by enabling developers to capture known-good rendering outputs and replay them after code or driver updates to verify consistent behavior. This is particularly valuable for platform bringup, where developers validate API compatibility and performance when porting applications to new hardware or operating systems. By replaying captures across diverse environments, GFXReconstruct ensures smooth operation and reduces the risk of regressions, making it indispensable for driver development and application stability.

As you can see, GFXReconstruct serves as a versatile tool for developers working on Vulkan and DirectX 12 applications, enabling them to build robust, high-performance software across Windows, Linux, and Android environments.

Current OS and API Support

GFXReconstruct is designed for cross-platform compatibility, supporting the following operating systems and graphics APIs as of its latest releases:

- **Operating Systems:**

  - **Windows**: Fully supported for both Vulkan and DirectX 12, leveraging the Windows 10 SDK (version 10.0.20348.0 for D3D12).
  - **Linux**: Robust support for Vulkan, with tools for capturing and replaying API calls.
  - **Android**: Supports Vulkan capture and replay, with specific configurations for Android 10 and newer, including permissions for external storage access.
  - **macOS**: Supports Vulkan via KosmicKrisp, a Vulkan-on-Metal driver developed by LunarG
- **Graphics APIs:**
  - **Vulkan**: Comprehensive capture and replay support, including experimental OpenXR support for developer evaluation.
  - **DirectX 12 (D3D12)**: Full support for capturing and replaying D3D12 applications, including DirectX Raytracing (DXR), introduced in 2023.
  - **OpenXR**: Initial support for OpenXR was demonstrated by LunarG at the AWE 2025 event.

While Vulkan and DirectX 12 are the primary focus, the API-agnostic design of GFXReconstruct's capture file format lays the groundwork for potential future support of other APIs, such as OpenXR or Metal, depending on community contributions and industry demand.

GFXReconstruct's open-source nature, MIT License, and support for Vulkan and DirectX 12 make it a cornerstone tool for modern graphics development. Its ability to reproduce defects, analyze API usage, profile performance, and support regression testing empowers developers to build robust, high-performance applications across Windows, Linux, and Android.

References

1. Silicon Graphics Inc. GLdebug Debugger User's Guide, 007-1489-030. SGI, 1993. https://irix7.com/techpubs/007-1489-030.pdf

2. X.Org Foundation. RECORD Extension Protocol Specification, X11R7.6. https://www.x.org/releases/X11R7.6/doc/recordproto/record.html

3. Damian Trebilco. GLIntercept: OpenGL Function Interceptor. SourceForge, 2002–2020. https://github.com/dtrebilco/glintercept

4. Microsoft. PIX on Windows – Performance Tuning and Debugging for Games. https://devblogs.microsoft.com/pix/

Chapter 2:

Getting Started with Installation and Setup

In this chapter, we'll focus on getting you up and running with GFXReconstruct for Vulkan on Windows and Linux, as well as DirectX 12 on Windows. The aim is to make onboarding straightforward, so you can start capturing and replaying API calls without unnecessary hurdles. Note that this chapter doesn't cover Android or macOS setups—those will be addressed in future chapters.

GFXReconstruct provides a Vulkan capture layer (**VK_LAYER_LUNARG_gfxreconstruct**), helper capture scripts like **gfxrecon-capture-vulkan.py**, and a suite of processing tools including:

- **gfxrecon-replay** (replay)
- **gfxrecon-info** (introspection)
- **gfxrecon-compress** (compression)
- **gfxrecon-convert** (JSON Lines conversion)
- **gfxrecon-extract** (SPIR-V extraction)
- **gfxrecon-optimize** (performance optimization), and
- **gfxrecon-tocpp** (experimental code generation).

We'll touch on how these fit into the setup process.

Prerequisites for Vulkan and DirectX

For Vulkan (Windows and Linux)

- **Operating System:** Windows 10/11 (64-bit) or a modern Linux distribution (e.g., Ubuntu 20.04+).
- **Vulkan SDK:** Download and install the latest Vulkan SDK from LunarG's website. This includes the Vulkan headers, loader, and GFXReconstruct binaries. Minimum version: SDK 1.3.243 or later for full features.
- **Python 3**: Version 3.6 or higher, required for capture scripts like **gfxrecon-capture-vulkan.py.**

- **Hardware**: A Vulkan-compatible GPU (e.g., NVIDIA GTX 10-series or later, AMD RX 500-series or later).
- **Additional for Linux:** GCC 7+ and CMake 3.24+ if building from source.

For DirectX 12 (Windows Only)

- **Operating System:** Windows 10/11 (64-bit) with DirectX 12 support enabled.
- **GFXReconstruct Binaries**: To get prebuilt binaries that include DirectX 12 support, download the latest GFXReconstruct release from https://github.com/LunarG/gfxreconstruct/releases. Note that DirectX 12 support is not included in the Vulkan SDK build of GFXReconstruct.
- **Windows SDK**: Version 10.0.26100 or later, installed via Visual Studio or standalone download from Microsoft.
- **Hardware**: A DirectX 12-compatible GPU (e.g., NVIDIA GTX 900-series or later, AMD RX 400-series or later).
- **Python 3**: As above, for scripts.

Once installed, verify your Vulkan setup by running **vulkaninfo** from the SDK (it should list your GPU and supported features). For DirectX 12, use the DirectX Diagnostic Tool (dxdiag) to confirm DX12 support.

Building from Source vs. Prebuilt Binaries

Getting GFXReconstruct up and running is pretty straightforward, whether you grab the ready-to-go binaries or roll up your sleeves and build it from source. For most folks, prebuilt binaries are the way to go for a quick start. But if you're itching to tweak things or play with experimental features, go ahead and build from source. Here's the lowdown, made simple.

Prebuilt Binaries

Want to hit the ground running? Snag GFXReconstruct through the Vulkan SDK or GitHub releases.

- Vulkan SDK Route:
    - Head to vulkan.lunarg.com and download the installer.
    - Run it, stick with the default settings, and you're set. On Windows, it adds tools like **gfxrecon-replay** to your **PATH**. On Linux, it sorts out the layers automatically.

    - Find the tools in the SDK's Bin folder—ready to roll!
    - *Heads-up*: DirectX 12 support is not included in the Vulkan SDK build of GFXReconstruct

- GitHub Releases for DirectX 12:
    - Need DX12? Grab the latest prebuilt binaries from https://github.com/LunarG/gfxreconstruct/releases.
    - These are stable, compatible, and ready to use.

This is the hassle-free option—perfect for jumping straight into capturing and replaying Vulkan or DX12 workloads.

Building from Source

If you need the latest development changes, want to enable experimental features (e.g. OpenXR), or customize build options, build from source instead of using the SDK.

Step 1: Grab the Code

Clone the repo and set up submodules:

```shell
git clone https://github.com/LunarG/gfxreconstruct.git --branch main
cd gfxreconstruct
git submodule update --init
```

If you want bleeding-edge features or want to extend GFXReconstruct and make a Pull Request, check out the dev branch.

Step 2: Get Your Tools Ready

Check **BUILD.md** for the latest, but here's the overview:

- CMake: 3.24+ is recommended.
- Python: 3.6+ for build script and code generation.
- Compiler:
    - Windows: Visual Studio 2019 or 2022 with the Desktop C++ workload.
    - Linux: GCC 9+ (or Clang) with C++17 support.

- Windows SDK: Use 10.0.26100.0 for DX12 (set with -DCMAKE_SYSTEM_VERSION=10.0.26100.0 if you've got multiple SDKs).
- Vulkan SDK: Needed for headers and loader during dev (or grab Vulkan headers via system packages).
- Linux Extras: Install X11/XCB and/or Wayland dev packages x (check BUILD.md for specifics).

Step 3: Easy Mode—Use the Build Script

The included Python script does the heavy lifting:

```shell
python scripts/build.py --skip-check-code-style --skip-tests -c Release
```

Handy flags to know:

- **–skip-d3d12-support**: Skip DX12 to keep things light.
- **–skip-tests**: Speed things up by skipping test apps.
- **-a arm64**: Build for Windows ARM64 (if you're on an ARM host).

Step 4: Manual CMake (If You Want Control)

Prefer doing it by hand? Here's how:

- Linux/macOS (Single-config, e.g., Ninja or Makefiles):

```shell
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
```

- Windows (Multi-config, e.g., Visual Studio):

```shell
cmake -S . -B build -G "Visual Studio 17 2022" -A x64 -DCMAKE_SYSTEM_VERSION=10.0.26100.0
cmake --build build --config Release
```

Useful CMake options:

- **-DD3D12_SUPPORT=OFF**: skip DX12 components
- **-DGFXRECON_ENABLE_OPENXR=OFF**: Disable OpenXR
- **-DUSE_CCACHE=On**: Faster rebuilds with ccache

Step 5: Install (Optional)

Want a clean setup like the SDK? Install the artifacts:

```
Shell
cmake --install build --config Release --prefix ./install
```

This puts tools like **gfxrecon-replay** in **install/bin/**. Skip this, and you'll find outputs scattered in the build folder (especially messy with Visual Studio's multi-config setup).

Step 6: Check Your Work

List the tools produced with the following command:

```
Shell
ls install/bin
```

Depending on the options you chose, you should see:

- **gfxrecon-replay**
- **gfxrecon-info**
- **gfxrecon-compress**
- **gfxrecon-convert**
- **gfxrecon-extract**
- **gfxrecon-optimize**
- **gfxrecon-tocpp**
- helper scripts like **gfxrecon-capture-vulkan.py**

Which Should You Choose?

Prebuilt binaries are a no-brainer for quick setup and guaranteed stability. Building from source is perfect if you're tweaking, experimenting, or chasing the latest features. Either way, you're now set to capture and replay like a pro.

Checking Your Setup

Once you've got GFXReconstruct installed (whether through the Vulkan SDK, a source build, or a quick install), let's do a fast test to make sure everything's working for Vulkan and DirectX 12. This section breaks it down into simple, repeatable steps, straight from the official docs.

Vulkan Test (Desktop)

Here are some steps to make sure your Vulkan setup is ready to roll:

Step 1: Make the layer discoverable

- If you used the Vulkan SDK or ran **cmake --install**, you should be good to go.
- If you built GFXReconstruct locally and didn't install it, tell Vulkan where to find the layer by setting **VK_LAYER_PATH**. For example:
    - Windows (command prompt):

```shell
set VK_LAYER_PATH=C:\gfxreconstruct\build\layer\Debug;%VK_LAYER_PATH%
```

    - Linux/macOS (terminal):

```shell
export VK_LAYER_PATH=/path/to/gfxreconstruct/build/layer:$VK_LAYER_PATH
```

Step 2: Enable the layer

Pick one of these options:

- **Option 1**: Vulkan Configurator (**vkconfig**) GUI:
    - Launch **vkconfig** from the Vulkan SDK.
    - Create or select a configuration and enable the "GFXReconstruct" layer.
    - Add your application (or set the configuration as active system-wide), then run from **vkconfig**.

    - If you built locally without install, add your build's layer manifest folder to **vkconfig**'s Layer Paths so it can discover the layer.
    - **vkconfig** applies the needed overrides (**VK_INSTANCE_LAYERS** and, if necessary, **VK_LAYER_PATH**) per-app—no manual env vars required.

- **Option 2**: Environment variable
    - Windows:

```
Shell
set VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_gfxreconstruct
```

    - Linux/macOS:

```
Shell
export VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_gfxreconstruct
```

Step 3: Capture a Test

Run a quick capture using the helper script:

```
Shell
gfxrecon-capture-vulkan.py -o test_vulkan.gfxr vkcube
```

- If **vkcube** isn't in your PATH, use the full path (find it in the Vulkan SDK samples).
- Or, just run your own Vulkan app with the layer enabled. The capture file will be **gfxrecon_capture.gfxr** unless you set **GFXRECON_CAPTURE_FILE** to something else.

Step 4: Replay and check

- Replay the capture:

```
Shell
gfxrecon-replay test_vulkan.gfxr
```

- Peek at the details:


```shell
gfxrecon-info test_vulkan.gfxr
```

What's a Win?

- The replay window (or headless mode) shows the frames you expect.
- No nasty errors in the log. If something's funky, crank up the debug info with:

```shell
set GFXRECON_LOG_LEVEL=debug  # Windows
export GFXRECON_LOG_LEVEL=debug  # Linux/macOS
```

DirectX 12 Test (Windows Only)

Let's get your DirectX 12 setup humming.

Step 1: Set up capture libraries

Drop these files next to your app's executable (e.g., **my_d3d12_app.exe**):

- **d3d12.dll**
- **dxgi.dll**
- **d3d12_capture.dll**

(Grab these from the SDK or your build's install folder. Delete them after testing to stop capturing.)

Step 2: Keep It Small (Optional)

To avoid a massive capture file, limit the frame range. One way to do this is to set a key for toggling capture on and off.

```shell
set GFXRECON_CAPTURE_TRIGGER=F3
```

Want a custom file name? Set:


```shell
set GFXRECON_CAPTURE_FILE=dx12_test.gfxr
```

Step 3: Run Your App

Launch your app from the same Command Prompt (so the environment variables stick):

```shell
my_d3d12_app.exe
```

Play through your scene until you reach the desired starting point, then press the hotkey (e.g., F3) to begin recording. Advance to the end of your target frame range and press the hotkey again to stop. GFXReconstruct will automatically generate a .gfxr file (**dx12_test.gfxr** or **gfxrecon_capture.gfxr**) containing only those frames, plus any prerequisite state needed for replay.

Step 4: Replay and Inspect

Replay the capture:

```shell
gfxrecon-replay dx12_test.gfxr
```

Check the details:

```shell
gfxrecon-info dx12_test.gfxr
```

Optional—Optimize for performance:

```shell
gfxrecon-optimize -o dx12_test_opt.gfxr dx12_test.gfxr
```


What's a Win?

- Replay finishes without "device removed" errors.
- If there's visual output, it looks like what you expected.

Why This Matters

This quick test ensures your GFXReconstruct setup is solid, so you can capture and replay Vulkan or DirectX 12 workloads without a hitch. It's like a handshake with your GPU—make sure it's firm!

Common Optional Checks

Goal	Command/Action
Get more detailed output during capture	**set/export GFXRECON_LOG_LEVEL=debug**
Get more detailed output during replay	**gfxrecon-replay --log-level debug <file.gfxr>**
Convert to JSON (inspection)	**gfxrecon-convert <file.gfxr>** # Creates a new file with the same name # but replaces .gfxr with .json

Troubleshooting

Common issues and fixes:

- Layer Not Found: Confirm **VK_INSTANCE_LAYERS** is visible in the app's environment and that the layer JSON is reachable (try setting **VK_LAYER_PATH**).
- Capture File Not Generated: Check if your app calls **vkCreateInstance/vkDestroyInstance** (Vulkan) or loads D3D12/DXGI (DirectX). Verify write permissions for the capture path.
- Memory Tracking Errors: Switch modes with **GFXRECON_MEMORY_TRACKING_MODE=unassisted** if page_guard conflicts (e.g., with debuggers).
- Replay Fails on DirectX 12: Ensure the Agility SDK is in a D3D12 folder next to **gfxrecon-replay.exe**. Optimize captures with **gfxrecon-optimize** for better performance.

- Hotkey Not Working: Confirm supported keys (F1-F12, TAB, CTRL) and no conflicts with app input.
- General Errors: Set **GFXRECON_LOG_LEVEL=debug** for detailed logs. Check the USAGE docs for API-specific tips.
- Debugger breaks caused by layer memory tracking: disable **SIGSEGV** breaks in your debugger for the target app. Check **USAGE_desktop_Vulkan.md** for the LLDB/GDB commands.
- Very large files: Use a hotkey trigger to capture only the frames you need (set **GFXRECON_CAPTURE_TRIGGER**, e.g., F3, to create a trimmed capture), then optimize the resulting .gfxr to remove unused initial-state data: **gfxrecon-optimize <input.gfxr> <output_opt.gfxr>** Note: Optimization operates on trimmed captures (created via trigger or frame selection) and produces a smaller, faster-to-replay file.

If issues persist, consult the GFXReconstruct GitHub issues list.

With GFXReconstruct set up, you're ready to capture and debug your graphics apps efficiently. Upcoming chapters will discuss advanced usage, Android/macOS support, and more. Happy developing!

Chapter 3:

Capturing Your First Frames

Now that we've covered installation, environment setup, and validation layer integration we'll focus on capturing our first workload and generating a **.gfxr** trace file—the foundation for debugging, performance analysis, and replay testing. By the end of this chapter, you'll have recorded real GPU commands and verified your capture—all within minutes.

Prerequisites Recap

Before proceeding, ensure:

- GFXReconstruct is installed (see Chapter 2).
- Vulkan SDK ≥ 1.3.268 is available.
- Your target application uses Vulkan (or supports Vulkan via compatibility layers).

Except for environment variable syntax and executable names, the commands for utilizing GFXReconstruct are the same on Linux, macOS, and Windows. The **gfxrecon-capture-vulkan.py** convenience script can further simplify usage. As we'll see in examples below, it removes OS-specific differences by specifying GFXReconstruct values as command line arguments instead of as environment variables.

Launching an Application with Capture Enabled

In order to begin capturing, you need to inject the capture layer at runtime. This method requires no code changes to your application.

Set the Capture Layer

Linux/macOS:

```shell
export VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_gfxreconstruct
```


Windows:

```shell
set VK_INSTANCE_LAYERS=VK_LAYER_LUNARG_gfxreconstruct
```

Launch Your App with Capture Settings

Option A: Use environment variables and run your app

Linux/macOS:

```shell
export GFXRECON_CAPTURE_FILE=my_first_capture.gfxr
export GFXRECON_CAPTURE_FRAMES=1-100
./your_vulkan_app
```

Windows:

```shell
set GFXRECON_CAPTURE_FILE=my_first_capture.gfxr
set GFXRECON_CAPTURE_FRAMES=1-100
your_vulkan_app.exe
```

Option B: Use the convenience script

```shell
gfxrecon-capture-vulkan.py -o my_first_capture.gfxr -f 1-100 ./your_vulkan_app
```

(Use **your_vulkan_app.exe** on Windows.)

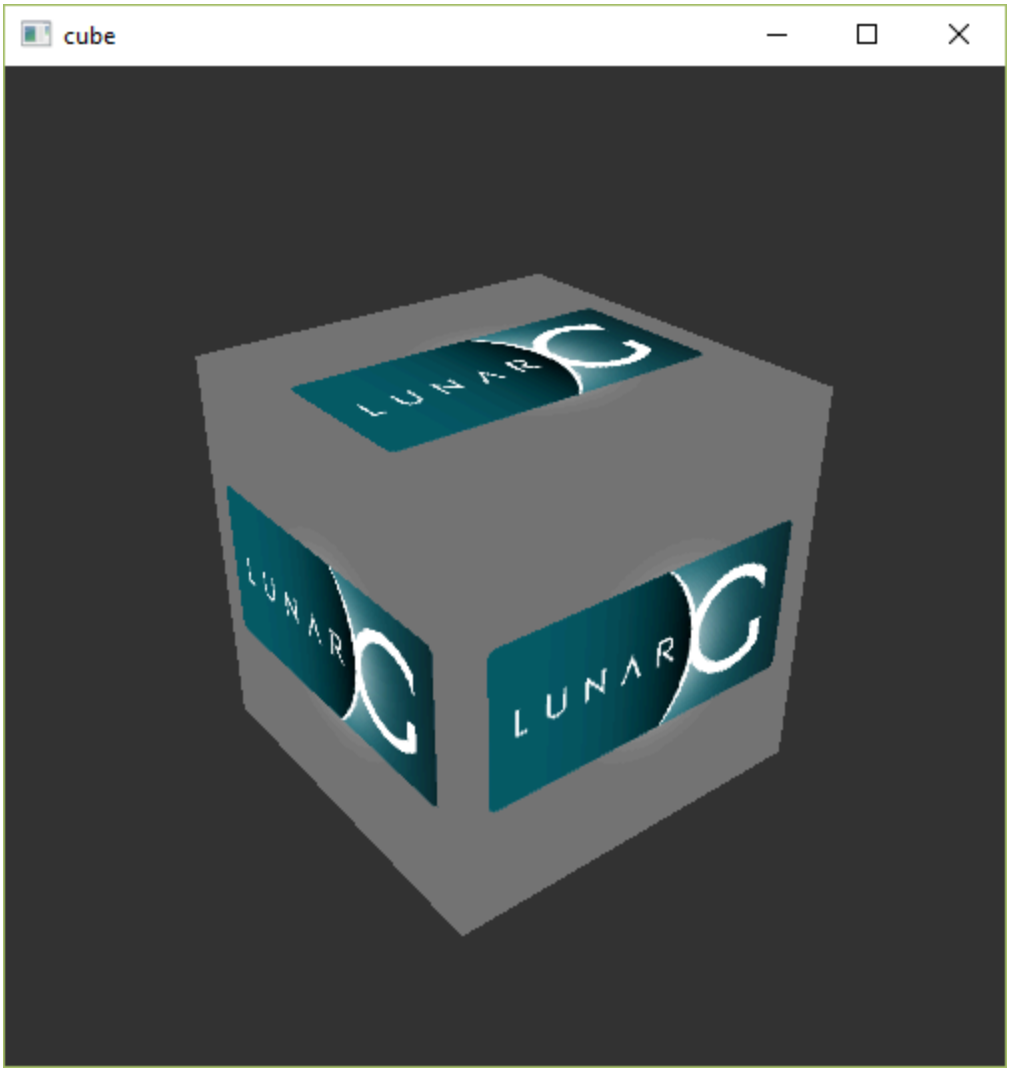

Figure 1. *vkcube* is a Khronos Vulkan sample that can be used to experiment with GFXReconstruct.

Key capture options (environment variables)

Variable	Description
GFXRECON_CAPTURE_FRAMES	Frame ranges to record (1-100, 50, or 10-20,40-60).
GFXRECON_CAPTURE_FILE	Output .gfxr file. Default: **gfxrecon_capture_.gfxr**.
G**FXRECON_MEMORY_TRACKING_MODE**	**page_guard** (default), **userfaultfd** (Linux/Android), **assisted**, **unassisted**.

GFXReconstruct Memory Tracking Modes

Memory tracking mode	Description
page_guard (default)	• Watches memory for changes by protecting it. • Fast and keeps files small. • Can clash with apps that install their own crash/signal handlers
userfaultfd (Linux/Android)	• Uses a kernel feature to catch page changes without signals. • Good fallback if **page_guard** clashes. • A bit slower.
assisted	• Only saves ranges after the app calls **vkFlushMappedMemoryRanges**. • Very efficient, but only if the app does this correctly.
unassisted	• Saves all mapped memory when unmapping or submitting work. • Easiest to reason about, but slow and can create huge files.

Filtering Captures for Precision

Large applications generate massive traces. Use frame range filtering and hotkeys to capture only what matters.

Frame Range Filtering

With the Vulkan API, the end of one frame and the start of the next one is defined by a call to **vkQueuePresentKHR()**. GFXReconstruct keeps track of the number of calls to this function and lets you specify which frames to record. In this example, only 6 frames of application data are captured, starting at frame 50:

```shell
gfxrecon-capture-vulkan.py -o focused_scene.gfxr -f 50-55 ./your_vulkan_app
```

(Use **your_vulkan_app.exe** on Windows.)

Applications that perform offscreen rendering or general computation and never call **vkQueuePresentKHR()** are not able to use this method.

Hotkey Triggered Capture (Interactive Mode)

To interactively capture a range of frames, start your app normally, then press the configured hotkey during runtime to begin capture, and again to stop. The file is written per start/stop pair. Enable hotkeys with GFXRECON_CAPTURE_TRIGGER and the key you want to use as the trigger (F3 in the example below):

Linux/macOS:

```shell
export GFXRECON_CAPTURE_TRIGGER=F3
```

Windows:

```shell
set GFXRECON_CAPTURE_TRIGGER=F3
```

Notes:

- Hotkeys require the capture layer to be loaded and the app to have keyboard focus.
- Valid triggers include F1–F12, TAB, and CTRL.

Interpreting Capture Output

After execution, you'll find a .gfxr file in your working directory. Let's verify and explore it.

Step 1: Inspect capture metadata

```shell
gfxrecon-info my_first_capture.gfxr
```


Sample output:

```
Shell
Exe info:
        Application exe name:
        Application version: 0.0.0.0
        Application Company name:
        Product name:

File info:
        Compression format: LZ4
        Total frames: 198

Vulkan application info:
        Application name: vkcube
        Application version: 0
        Engine name: vkcube
        Engine version: 0
        Target API version: 4194304 (1.0.0)
        Used resolutions: 800x600
```

Step 2: Convert to JSON for scriptable inspection

```
Shell
gfxrecon-convert my_first_capture.gfxr --output my_first_capture.jsonl
```

Inspect draw calls, pipeline state, and descriptor bindings in human-readable form, or filter or pretty-print with tools like https://jqlang.org/.

Step 3: Replay the Capture (Validation)

```
Shell
gfxrecon-replay my_first_capture.gfxr
```

Watch your app replay, frame-by-frame. Any deviation indicates a driver or application bug. Expect near pixel-perfect replay; minor differences can appear due to driver or timing variance.

Quick Start Summary: Your First Capture

Linux/macOS:

```shell
gfxrecon-capture-vulkan.py -o demo.gfxr -f 1-10 ./your_vulkan_app
gfxrecon-info demo.gfxr
```

Windows:

```shell
gfxrecon-capture-vulkan.py -o demo.gfxr -f 1-10 your_vulkan_app.exe
gfxrecon-info demo.gfxr
```

That's it — you've recorded GPU commands into a portable .gfxr file!

Next Steps

Now that you can capture reliably:

- Use **gfxrecon-optimize** to remove unused resource initialization from trimmed captures.
- Integrate capture/replay into CI using **gfxrecon-capture-vulkan.py** and **gfxrecon-replay**.
- Explore resource inspection with **vulkan_dump_resources.md** and replace shaders at replay with **gfxrecon-replay –replace-shaders**.

Chapter 4:

Understanding the .gfxr File

What GFXReconstruct Actually Captures and Replays

If you made it this far, you've learned how to capture your first Vulkan frames (Chapter 1), how to trigger captures precisely when you need them (Chapter 2), and how to replay and trim those captures (Chapter 3). In this post, we step back and look at the central artifact in all of this: the .gfxr capture file.

This chapter explains what a .gfxr file contains, how it's structured, what gets captured (and what does not), and why replay is typically deterministic and visually identical to the original run. Future chapters will dive deeper into topics like memory snapshots, optimization, and debugging mismatches during replay.

What Is a .gfxr File?

A .gfxr file is a **binary, frame-accurate** record of API usage created between the start and end of a capture session. It is produced by the **GFXReconstruct capture layer** and consumed by the **gfxrecon-replay** tool.

Think of it as a GPU API *tape recorder*: it tracks the exact sequence of API calls, the parameters passed to those calls, and—when needed—the relevant GPU memory contents. The intent is that the replay tool can recreate the same command stream on another machine, driver, or even GPU vendor, as long as the Vulkan features and extensions match.

In practice, cross-driver or cross-vendor replay can sometimes expose differences (or even reveal bugs). When this happens, the .gfxr file is often the key to diagnosing what's going wrong.

This article focuses on .gfxr files produced by Vulkan applications on Linux, macOS, and Windows. Android usage will be covered in a future post. In addition to Vulkan, GFXReconstruct can also capture OpenXR and Direct3D API calls.

Structure of a Capture File

A .gfxr file is a single, self-contained binary. You can think of this file as a header that represents the format version and a few options (e.g. compression type, etc.) followed by a stream of blocks, most of which are captured API calls. For up-to-the-minute detailed information about the .gfxr file format, consult the GFXReconstruct documentation.

GFXReconstruct's capture file format is designed with **forward compatibility** in mind. This means that capture files created with any given version of GFXReconstruct should be playable (replayable) on **all future versions** of the tool

However, the reverse is not guaranteed: capture files generated by a **newer** version of GFXReconstruct may not be compatible with older builds. For instance, if a newer capture includes Vulkan API calls, extensions, or features introduced after an older GFXReconstruct version was released, that older build will typically be unable to process or replay the file correctly.

What Gets Captured

A .gfxr file captures everything required to reconstruct the original GPU workload:

Captured Content

- All commands issued on any queue from an API supported by GFXRecontruct (Vulkan, Direct3D, OpenXR)
- Command buffer contents
- Pipeline, descriptor set, and shader module state
- Buffer and image memory contents necessary to correctly replay the captured API commands
- Swapchain creation and presentation calls
- Debug markers and object names (if VK_EXT_debug_utils is active)
- Extension-specific commands supported by GFXReconstruct (ray tracing, mesh shading, video, etc.)

What Isn't Captured (by design)

GFXReconstruct records only what is necessary to reproduce GPU behavior. It deliberately does not capture:

- Application CPU logic
- Any window-system events (X11, Wayland, Win32, Cocoa)
- API calls other than Vulkan, Direct3D, and OpenXR (e.g., OpenGL, Metal, CUDA, etc.)

This means a replay focuses strictly on GPU-visible work, not application flow.

Achieving Determinism — Why Replay Is Usually Visually Indistinguishable

GFXReconstruct takes several steps to ensure deterministic results:

- **Memory snapshots** capture the initial contents of buffers and images precisely when first used by the GPU.
- **Command ordering** with respect to synchronization.
- **Timestamps and fences** follow the original logical sequence.
- **Swapchain operations** are re-acquired and presented in the same order.

The result: **On the same OS, driver version, and GPU hardware, replay is typically visually identical**. Across vendors and drivers, replay is still usually visually indistinguishable, with deviations generally caused by driver differences, undefined behavior, or unsupported extensions.

Tools You'll Use Every Day

GFXReconstruct ships several tools that work consistently across Linux, macOS, and Windows:

Tool	Purpose
gfxrecon-info	Summarizes a .gfxr file (frame count, duration, API version, extensions used)
gfxrecon-convert	Converts a .gfxr file to json
gfxrecon-replay	Replays a .gfxr file
gfxrecon-optimize	Reduces file size by eliminating unused memory snapshots (advanced)

The Vulkan SDK is not required if you are capturing OpenXR or Direct3D calls. Platform differences are minimal: just ensure that the Vulkan SDK or standalone GFXReconstruct build you are using matches your OS and architecture.

Portability (and Non-Portability) of .gfxr Files

Here is what you can expect when replaying a capture across different environments:

Scenario	Expected Result
Same OS, same driver version	Visually indistinguishable
Same vendor, different driver version	Rare visual differences possible
Different GPU vendor (NVIDIA ↔ AMD ↔ Intel)	High success rate if required extensions exist
Linux → Windows or macOS	Works as long as API features match
Windows → Linux or macOS	Works as long as API features match

Coming Up Next

Future chapters will explore:

- How memory snapshots work (and how to reduce capture size)
- Using gfxrecon-optimize and trimming more effectively
- Automating capture and replay in CI pipelines

For now, grab the latest version of GFXReconstruct from the LunarG GitHub releases page or the Vulkan SDK, and start exploring your .gfxr files with **gfxrecon-info**. You may be surprised how much insight a single capture file reveals.

