1.3 Vulkan Loader Improvements

Mark Young, LunarG
Charles Giessen, LunarG

March 2022

Introduction

The Khronos Vulkan Loader is a crucial component for desktop Vulkan implementations. With the recent release of Vulkan 1.3, LunarG updated the Vulkan loader to include changes to support the new API version as well as several other important changes made in the last six months. We hope these changes help developers to better understand the loader and assist them in resolving difficult issues more quickly.

Documentation Changes

Software products are only as good as their supporting documentation. Bad docs can hurt just as much as great docs help. Because of this, we went through and cleaned up the language across the loader documentation. Many things that were confusing before have been simplified, and some of the more messy language has been clarified.

New Documentation Location

All documentation now lives in a new “docs” folder which is a top level folder in the source tree (https://github.com/KhronosGroup/Vulkan-Loader). This location makes it easier to find and browse available loader documentation.

New Organization

We also decided to split up the loader documentation. Previously, opening up the loader documentation resulted in a colossal document that had a lot of irrelevant information for the average reader, which made finding information relevant to your situation difficult.

To remedy this situation, we have split the loader documentation into four new files:

- LoaderInterfaceArchitecture.md
- LoaderApplicationInterface.md
- LoaderLayerInterface.md
- LoaderDriverInterface.md

For clarity the top-level documentation file has been renamed to "LoaderInterfaceArchitecture.md." The old file with the name "LoaderAndLayerInterface.md" still exists in its former location next to the loader source, but now simply points to this new file. "LoaderInterfaceArchitecture.md" is an overview of the loader and its high-level interactions with other components. It includes links to portions of the other documentation relevant for applications, layers, and drivers. Most users will browse through this file only a few times and spend most of their time in one of the other files, but it does contain a reference table of environment variables and glossary of terms used throughout the documentation.

The application-specific information has been pulled into its own document now called "LoaderApplicationInterface.md". This document now contains information most application developers and general users of the Vulkan loader will find useful. This information includes topics like: how to link the loader, how to debug loader issues, and how layers are organized. The intent is that general users should not need to look at any other document.

Layer-specific and driver-specific information have been moved into "LoaderLayerInterface.md" and "LoaderDriverInterface.md" respectively. Each of these is now focused solely on the developers for those components of the Vulkan ecosystem. Some users may still find the layer information useful if they interact with layers frequently.

VkConfig Interaction

VkConfig continues to be an extremely useful tool for developers and experienced end-users. Unfortunately, previous loader documentation only sparsely mentioned VkConfig. We have expanded the information on how the loader interacts with VkConfig as well as added its interaction with various top-level diagrams. The new sections can be found in the LoaderInterfaceArchitecture.md document under the "VkConfig" heading as well as various mentions in the LoaderApplicationInterface.md document.

Clarify Behaviors for Elevated Privilege Applications

Some developers were pointing out that the behavior of the loader when running elevated applications wasn't clear. Therefore, we have added a new "Elevated Privilege Caveats" section to the LoaderInterfaceArchitecture.md document as well as specific information to the LoaderLayerInterface.md and LoaderDriverInterface.md documents detailing what functionality of the loader is unavailable when running an elevated application and why.

Document Loader Search Paths

We realized the documentation around loader search paths on Linux and Apple, specifically for layer and driver manifest files, was out of date. Because of this, we went back and clarified the paths and cleaned up the documentation around the search process. The Linux layer and driver search paths are detailed in the LoaderLayerInterface.md and LoaderDriverInterface.md documents.

Additionally, we have added brief details on how layer and driver discovery is performed on the Fuchsia platform in those same documents as well.

Debugging Improvements

Over the last year, we've been asked by several developers to assist in debugging layer and driver issues. With the logging that was originally in place, it was very difficult to track down these issues without further instrumenting the Vulkan loader. Because of this, we have undertaken an effort to expand the logging to assist in these types of issues.

Loader Identification

Since some developers may have multiple loaders on their system, we added functionality to identify the loader version being used and where it was built in terms of the Loader Git history. This information is output when you set:

VK_LOADER_DEBUG=info ('all' will also reveal this information)

This information is typically output near the beginning when the loader is first triggered.

For example, the information might look like the following:

```
INFO: Vulkan Loader Version 1.2.198
```

This tells you the loader that is being used supports Vulkan API version 1.2 and it was built against patch 198 of the specification, so it should include all instance extension entry-points.

A loader built from the loader Git repository will have additional information which includes tag and branch information used during the build. This will appear as a second line that looks like the following:

```
INFO: [Git - Tag: v1.2.199-2-gb21acf16e, Branch/Commit: master]
```

This shows that it was built off of the “master” branch after the “v1.2.199” tag. While not that useful to end-users, developers can use it to determine if the loader being used is the version they desire which may include bug fixes or features they require.

Layer Debugging

An issue that has been brought up multiple times is how to debug layer issues. Even something as simple as attempting to determine what layers are actually active can be extremely difficult. To that end, we have added enhanced layer debugging to the latest loader.

To enable this, simply set:

VK_LOADER_DEBUG=layer (‘all’ will also reveal this information)

This action will output a list of which layers are enabled, what type of layers they are (explicit versus implicit), where the corresponding libraries are located, and other useful information.

Layer Searching Example:

The following shows example output for when the loader is in the process of searching for implicit layer manifest files:

```
LAYER: Searching for layer manifest files
LAYER:  In following folders:
LAYER:   /home/${USER}/.config/vulkan/implicit_layer.d
LAYER:   /etc/xdg/vulkan/implicit_layer.d
LAYER:   /usr/local/etc/vulkan/implicit_layer.d
LAYER:   /etc/vulkan/implicit_layer.d
```

```
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d
LAYER:   /home/${USER}/.local/share/flatpak/exports/share/vulkan/implicit_layer.d
LAYER:   /var/lib/flatpak/exports/share/vulkan/implicit_layer.d
LAYER:   /usr/local/share/vulkan/implicit_layer.d
LAYER:   /usr/share/vulkan/implicit_layer.d
LAYER:  Found the following files:
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d/renderdoc_capture.json
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d/steamfossilize_i386.json
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d/steamfossilize_x86_64.json
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d/steamoverlay_i386.json
LAYER:   /home/${USER}/.local/share/vulkan/implicit_layer.d/steamoverlay_x86_64.json
LAYER:   /usr/share/vulkan/implicit_layer.d/nvidia_layers.json
LAYER:   /usr/share/vulkan/implicit_layer.d/VkLayer_MESA_device_select.json
```

In the above scenario, seven implicit layers were discovered in two different folders. Just because they were found does not mean that they will be loaded, but this information can be used to make sure a layer JSON file was properly discovered.

Loading Layer Example:

Later on, when layers are actually loaded, you might see output similar to the following:

```
LAYER | DEBUG: Loading layer library libVkLayer_khronos_validation.so
LAYER | INFO: Insert instance layer VK_LAYER_KHRONOS_validation
(libVkLayer_khronos_validation.so)
LAYER | DEBUG: Loading layer library libVkLayer_MESA_device_select.so
LAYER | INFO: Insert instance layer VK_LAYER_MESA_device_select
(libVkLayer_MESA_device_select.so)
```

Here you can see the loader is loading and inserting 2 layers.

vkCreateInstance Layer Call Chain Example:

Finally, this is all put together when *vkCreateInstance* is called and the instance call chain is developed. The following shows an example log output when the loader has established this call-chain.

```
LAYER: vkCreateInstance layer callstack setup to:
LAYER:  <Application>
LAYER:    ||
LAYER:  <Loader>
LAYER:    ||
LAYER:  VK_LAYER_MESA_device_select
LAYER:      Type: Implicit
LAYER:         Disable Env Var:  NODEVICE_SELECT
LAYER:      Manifest: /usr/share/vulkan/implicit_layer.d/VkLayer_MESA_device_select.json
LAYER:      Library:  libVkLayer_MESA_device_select.so
LAYER:    ||
LAYER:  VK_LAYER_KHRONOS_validation
LAYER:      Type: Explicit
LAYER:      Manifest: /usr/share/vulkan/explicit_layer.d/VkLayer_khronos_validation.json
LAYER:      Library:  libVkLayer_khronos_validation.so
LAYER:    ||
LAYER:  <Drivers>
```

In this scenario, two layers were used (the same two that were loaded earlier):

- VK_LAYER_MESA_device_select
- VK_LAYER_KHRONOS_validation

This information now shows us that the `VK_LAYER_MESA_device_select` is loaded first, followed by `VK_LAYER_KHRONOS_validation` which will then continue into any available drivers.

It also shows that `VK_LAYER_MESA_device_select` is an implicit layer which indicates that it wasn't directly enabled by the application. On the other hand, `VK_LAYER_KHRONOS_validation` is shown as an explicit layer which indicates that it was likely enabled by the application or by environment settings.

Furthermore, it shows that `VK_LAYER_MESA_device_select` can be disabled by setting the "disable" environment variable "NODEVICE_SELECT" to a non-0 value.

Handle Validation

The loader bears the responsibility for being first in the call chain for all instance and physical devices functions. Unfortunately, there was no validation of the incoming handles which could lead to some bizarre behavior or crashes if an invalid handle was passed in. Now, most VkInstance and VkPhysicalDevice handles are validated upon entering the loader's trampoline function. If an issue is detected with the incoming handle, the loader will log an error message and then cleanly abort. This should result in more quickly being able to debug issues of this nature.

Testing Framework

In order to improve the long-term quality of the loader, we have finished an effort to completely rewrite the existing loader tests. To that end, we have created a new test writing framework that supports creating mock layers and mock drivers, which are useful to simulate various scenarios.

This framework allows us to test behaviors and corner-cases we weren't able to validate previously. Test cases now covered thanks to the new framework include:

- Driver & layer interface version requirements
- Unknown function enumeration

- Multiple physical device & device groups
- Complex layer behavior tests
- Enable Address Sanitizier in tests

The main core of the test framework can now be found under the “tests/framework” folder.

Moving forward, if you submit a PR, you will need to write the associated test, or correct an existing test as part of the submission process.

Changes Contributed by Vulkan Community Members

Some impactful changes during the second half of 2021 have come from the broader Vulkan community. We’re extremely grateful to developers who take the time to assist us in making this important product even better. The changes called out below are just a sample of those helping to support the Khronos Vulkan loader.

AARCH64 Unknown Physical Device Extensions

Eric Sullivan of NVIDIA added support for unknown physical device extensions on the ARM64 architecture. This functionality will greatly help those using the Vulkan loader on platforms using ARM processors. The functionality replaces C trampoline functions with dynamic assembly behavior when necessary, improving performance and reliability.

Cleanup XDG Paths

Simon McVittie of Collabora helped clarify and clean up the XDG path usage in the Vulkan loader and the supporting documentation. He revealed that we were using the paths incorrectly and this could have led to the Vulkan loader loading layers or drivers in the wrong order or ignoring properly installed components.

BSD Support

Eleni Stea has supplied multiple changes towards supporting the Vulkan loader on BSD. While not actively being worked on by LunarG, this support should open up using Vulkan and layers on BSD-based operating systems.

Available Now in GitHub

Some additional features are available now in GitHub, but will also be available in the next SDK with a Vulkan header version greater than 1.3.204. These include:

Linux Consistent Device Ordering

On the Linux platform, the order of Vulkan physical devices returned from the loader's vkEnumeratePhysicalDevices terminator can vary from run to run. This behavior is inherent to how the loader searches folders for Vulkan drivers and the devices available to those drivers. Because this order can vary, applications might select different devices causing performance or quality bugs, which seem to appear randomly. We have updated the loader to generate a consistent order of devices out of the terminator (prior to any layers) so that the same order is returned from run to run.

We have also provided a mechanism to disable this if you believe this is causing any issues. Simply set the environment variable "VK_LOADER_DISABLE_SELECT=1" before using the loader. This will cause the ordering algorithm to be skipped.

Defining Policies

We have also defined conformance policy statements and requirements for layers, drivers, and the loader. The intent is to clearly state requirements for proper execution of all components of Vulkan implementation on a user's system. The desired outcome is to improve the quality of Vulkan components and produce a better end-user experience. The loader has also added multiple checks to validate if drivers and layers are properly following the policy statements defined in the loader documentation. These messages should be mostly warnings now, but some could be raised in severity if it's determined to be a critical issue in the future.

Single vkEnumeratePhysicalDevice call

An irritating behavior of the Vulkan loader has been its behavior around *vkEnumeratePhysicalDevices*. The loader would always query all available devices from the top trampoline function for every *vkEnumeratePhysicalDevices* call by the application, no matter what the application wanted. The original intent of this behavior was to allow the loader trampoline to know about every available device -- always. However, this resulted in two *vkEnumeratePhysicalDevices* calls through every layer and driver for every call to *vkEnumeratePhysicalDevices* in the application. This doubling has caused confusion and produced a lot of redundant traffic through all enabled layers that complicated debugging. Now the loader will simply generate a single vkEnumeratePhysicalDevices call through each layer per application call.

Additive Environment Variables

In an attempt to help improve user configurability, we are going to add new environment variables that can be used to expand paths instead of override them. RIght now, the loader supports VK_ICD_FILENAMES and VK_LAYER_PATH. Both of these variables override the default driver and layer search behavior. In addition, we will be adding new environment variables that will add custom driver and layer locations without overriding the default search behaviors. Of course, these will be ignored in the elevated privileges scenario to preserve system security.

More Layer Debugging

We intend to continue to improve loader logging of layers. Some areas of improvement include not throwing warnings for layers that are of the wrong bit-type unless they are the only version found. While small, our intent is to continue to help end-users debug issues quickly. In addition, we should start throwing an error if a particular explicit layer was not found. Currently, that may occur silently under certain circumstances.

Handle Vulkan “Variant” Properly

Vulkan 1.2.175 introduced the concept of a Vulkan Variant, for the upcoming release of Vulkan Safety Critical (Vulkan SC). Since the Vulkan Loader is only designed for the main variant of Vulkan, any application, layer, or driver that indicates they support a different variant of Vulkan will be ignored with an info warning.