LUNARG

The Vulkan Loader and Vulkan Layers: Diagnosing Layer Issues

Mark Young, LunarG
Charles Giessen, LunarG
December 2022

Introduction

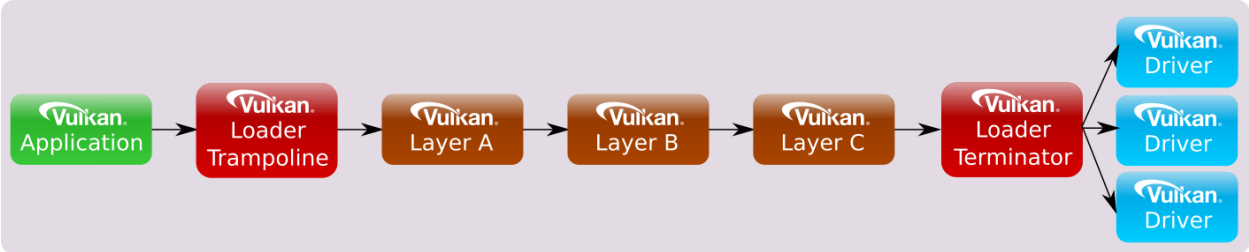
Vulkan layers are a unique mechanism that supports expanding the Vulkan API in ways other graphics APIs do not allow. The Vulkan loader provides the functionality required for Vulkan layers to be loaded dynamically, providing the ability to add and remove functionality at runtime. A primary example of this is only enabling validation when a developer is implementing and/or testing their Vulkan application. For production, the developer simply does not enable the Vulkan Validation layer so that the end-user is not hindered by the expense of validation during normal execution.

What's really unique about the Vulkan loader and layer design is that it is publicly available which allows any developer to create functionality that can be used by any Vulkan application on any user's system. But this is a two-edged sword because not every layer behaves correctly in all situations, causing issues that are hard to diagnose and debug. Enhancements to the Vulkan Loader enable easier debugging of layers that are causing issues.

Quick Overview of Layers

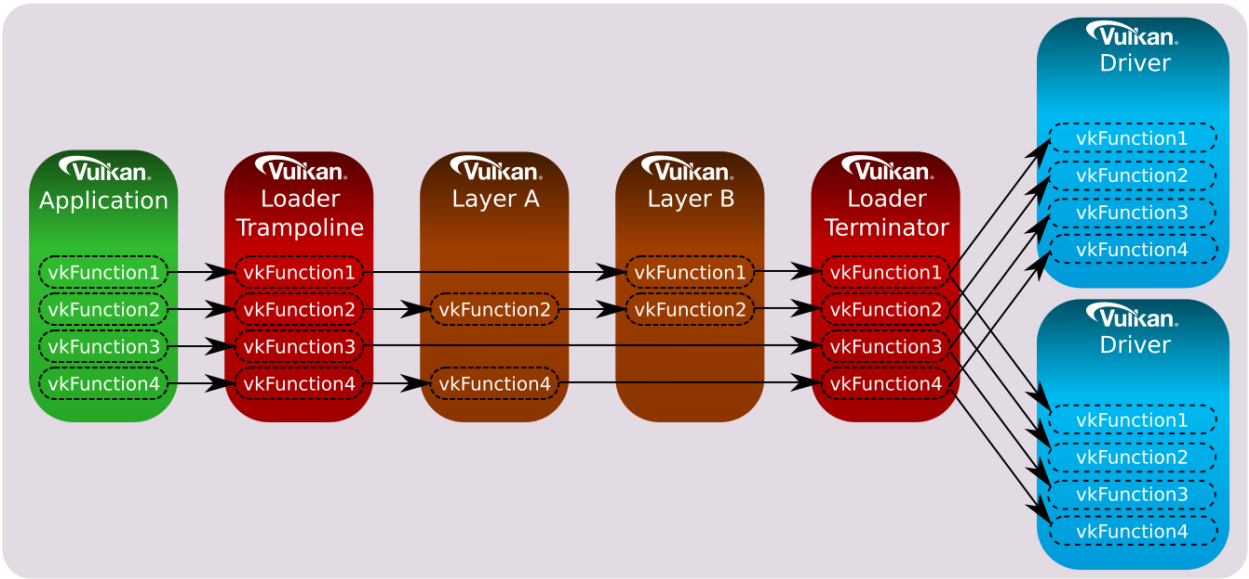
For the purposes of this paper, it is best to think of stepping through individual Vulkan commands as following a chain (referred to as a call chain). The start of the call chain is the Vulkan application with its call into a given Vulkan API command. This call proceeds into the Vulkan loader, through any enabled Vulkan layers, and finally ends up in one or more Vulkan drivers. It is a bit more complicated than that, but this simplified understanding is all you need for an overview of the Vulkan Layer mechanism.

Here's an example diagram of the Vulkan instance call chain flow from the application down to multiple drivers:



Layers are detected by the Vulkan loader and can either be automatically inserted by the loader or by request of the application or a user. Layers can augment the Vulkan API in many different ways by intercepting the Vulkan API commands. This could include intercepting every Vulkan command, or only a subset. Because of this, it can be difficult to determine which Vulkan command is being modified by any given layer short of walking through the call chain at runtime using a debugger.

The following image is intended to show how 4 theoretical Vulkan instance commands could be intercepted by two layers (or not at all) before finally proceeding down to the drivers:



Layers can adjust Vulkan behavior by adding new Vulkan functionality, removing existing Vulkan functionality, or by making additional API calls independently of the application. This is important to remember when working with layers as any layer could change the way an application behaves.

The Vulkan API defines two types of layers:

- Implicit Layers
- Explicit Layers

Implicit Layers

Implicit layers are intended to be enabled all the time, as long as the Vulkan loader finds them. Not every implicit layer wants to be always active, so is only enabled if a specific environment variable is defined. This is used for context-specific layers, like debugging. The “enable” environment variables are defined by the implicit layer itself. Many implicit layers do not define such “enable” variables and will therefore be automatically enabled.

However, all implicit layers **must** define a “disable” environment variable that can be used to disable the layer. This mechanism was designed to provide a fix in the case a layer was misbehaving. Unfortunately, the process of determining a given implicit layer’s “disable” environment variable is difficult because it is only defined in the layer’s manifest file, and is not available through the Vulkan API directly.

Common uses for implicit layers include:

- A store layer providing certain game utilities
- Chat discussion overlays
- Device prioritizing layers
- API capture layers
- Debugging and profiling

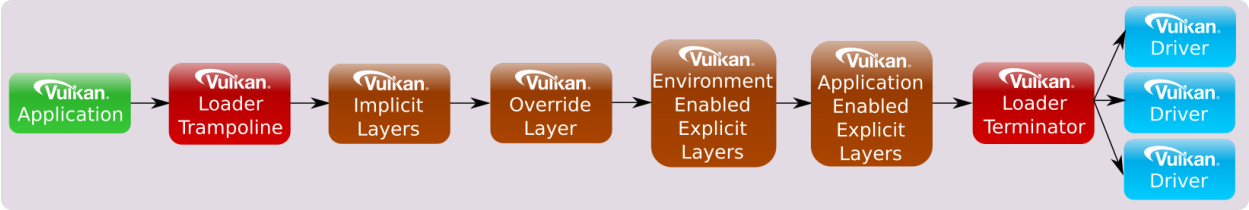
Explicit Layers

Explicit layers are layers that are enabled by the application with the call to vkCreateInstance. These will not be enabled by default and must be explicitly called out for the layer to be activated.

Explicit layers may also be enabled using environment variables (such as VK_INSTANCE_LAYERS). This method is most often used when attempting to enable a specific layer or diagnose issues without modifying the application.

Overall Layer Order

Implicit layers are added first in the call chain, and then explicit layers are added afterward. Additional special layers may be loaded in between or after, but typically the order of layers is as follows:



More details on how layers work, how the loader finds the various types of layers, and how the device call chain differs from the instance call chain can be found in the Vulkan Loader Github documentation. The most relevant document is the LoaderLayerInterface.md file in that directory.

The Benefits of Vulkan's Layered Architecture

While the layered architecture appears overly complicated, it does improve the API in several useful ways. A few examples of these improvements are listed below.

Validation Only During Development

A layered architecture allows for the vast majority of API validation code to be loaded only as needed. In most of OpenGL's lifetime, validation was always present and executed in every graphical application. A released application still ran with all the same validation checks executing as an application in development. Eventually, extensions were added to work around this in OpenGL. From Vulkan's beginning, validation is enabled only when the validation layers are added to the instance and device call chains by a developer. Thus, when an application is released to the public, it is without validation checks being executed resulting in a much more performant final product.

Simpler Debug Tool Support

Certain tools (such as RenderDoc and GFXReconstruct) require capturing graphical application data when they are used. In previous APIs like OpenGL or D3D, a developer needed to rename DLLs or SOs and adjust permissions to be able to insert the tool capture library in between the application and the system API library. With the layered architecture, the tool can simply enable a capture layer as an Implicit layer which will automatically capture the content required.

Runtime Optimization Support

Many times, applications are written to run on a variety of platforms and graphics cards. However, Vulkan provides multiple mechanisms to pre-cache content for an end user's system, which will speed up the load time on future runs of the same application. But not every application takes advantage of this behavior by default. Because of this, layers have been written to step in between the application and the API and automatically enable some of this pre-caching behavior to optimize application loading performance. One example of this is Steam's Fossilize layer which will pre-cache shaders for applications using the Steam engine.

Layer Problems That Can Occur

When a layer causes issues, it can be difficult to determine that the source of the issue is a layer and not the application or driver. Often a badly behaving layer can make it appear that the application is not behaving correctly -- wasting resources when the wrong team of engineers is attempting to investigate the issue. The following examples show how layers could affect the behavior of applications.

Changing available physical devices

Hardware vendors may ship their own implicit layers for special handling of their devices on a user's system. These layers may order devices based on some criteria such as placing discrete devices first in the list of available devices when a laptop is plugged in, but placing integrated devices first in the list when the laptop is on battery.

At one point, a problematic implicit layer from a hardware vendor caused all physical devices from other hardware vendors to disappear from the list of available devices on people's systems. Even though the other devices were all properly installed, had updated drivers, and functioned normally, the layer removed them from the list of available Vulkan devices returned to applications.

While some applications were not affected, applications using features specific to one of the disabled vendors suddenly no longer functioned and failed to run. It likely even caused performance losses as users' higher-performance cards were suddenly removed from the list of usable devices. This affected multiple game vendors as end-users reported the issues as problems with their games and were unaware of the source of the problem. Then, to further compound the problem, the issues were then reported to vendors whose devices had been filtered out instead of the vendor who created the layer.

This bad layer caused multiple companies to spend a lot of time on the wrong problem and caused many days of consternation for end-users.

Incomplete Handle Wrapping

Since Vulkan layers can take advantage of the Vulkan API by modifying parameters, they can define their own Vulkan handles, which are passed up or down the Vulkan call chain. In fact, they may pass their version of one or more Vulkan handles up the call chain to an application instead of the Vulkan handles returned by the driver. This is known as “handle wrapping”.

If a layer implements “handle wrapping” but fails to intercept all Vulkan commands that take a wrapped handle, errors and crashes occur. This is because the driver or application receives handles that do not match up with any known handles. Since Vulkan assumes that applications use the API correctly, applications, the Vulkan loader, other layers, and Vulkan drivers are likely to crash when they receive bad handles. This is compounded by the extensible nature of Vulkan, since when a layer is first written it may correctly wrap handles in all known places, but as new extensions and functions are added over time holes may start appearing in the layer’s wrapping coverage.

For example, if the layer-wrapped VkPhysicalDevice handles returned from the driver, but failed to intercept ***vkGetPhysicalDeviceFormatProperties***, then whenever an application called that command, it would enter the next lower layer or the loader with a handle created by our layer and not one created by a driver. The best thing that could happen is that the following layer would report a bad handle and return. But it’s entirely possible that the next layer or driver would attempt to dereference the invalid handle to get information about that VkPhysicalDevice, such as the dispatch table, and crash.

Using Additional Resources

Layers also can consume additional resources beyond those requested by an application. For example, if a layer implements some kind of console overlay, it is likely allocating textures and vertex buffers, descriptor sets, and command buffer resources on the GPU. This could lead to earlier exhaustion of GPU resources depending upon how complicated the layer gets with its rendering and how much the application requires.

Adding Extensions That Can Cause New Behavior

Layers may also cause problems depending on if they secretly enable certain Vulkan extensions. This is true because some extensions provided by the Vulkan API could introduce side effects in core Vulkan.

For example, if a layer enabled the **VK_KHR_portability_enumeration** extension without the application knowing, then the Vulkan loader would add devices to the list of available devices that are not fully Vulkan conformant if found on the system. This might cause issues if an

application accidentally selected one of those devices believing it would be fully Vulkan conformant.

Enabling API Features That Can Degrade Performance

Layers may enable features that it needs to operate. Certain features are known to have non-trivial performance or behavioral changes. For example, if a layer were to enable robustBufferAccess, the application may experience serious performance degradation with no observable cause.

Mitigations

Since there are complicated issues that could be caused by layers, what is the Vulkan ecosystem doing to avoid or diagnose these issues? Several efforts have been made to improve layer quality and debugging capabilities on end-user systems:

Loader Policies

The Vulkan Working Group works to define the clear and proper behavior of every Vulkan component. To this end, LunarG has created policies for clarifying the proper behavior of the Vulkan loader, Vulkan layers, and Vulkan drivers. The loader policies are documented in the Vulkan-Loader GitHub repository documentation in the LoaderLayerInterface.md file under the "Loader And Layer Interface Policy" section. Starting with the Vulkan loader built using the Vulkan 1.3.206 headers, the loader has been modified to start reporting various layer policy violations where possible. Some of the policies are currently untestable, but LunarG will continue to expand policy verification wherever possible.

Improved Loader Logging

In order to improve the debugging process for layer issues, the Vulkan loader has been modified to output layer information in more useful ways.

To enable this, simply set:

```
[Windows Command Prompt] set VK_LOADER_DEBUG=layer
[Bash or similar shells] export VK_LOADER_DEBUG=layer
```

This can be combined with other message types so that errors, warnings, and layer messages can be produced if the setting was changed to:

```
set VK_LOADER_DEBUG=error,warn,layer
```

NOTE: `'all'` will also reveal this information since it includes all sub-categories in debugging. Either way, layer-specific messages will be prefaced with a "LAYER" tag you will see below in the example output.

This will now output information on:

- What locations are searched for layer Manifest files
- What layers are enabled
- The type of each layer (explicit versus implicit)
- Where the corresponding libraries are located
- The environment variables that can be used to enable/disable each implicit layer

Here is some example output that shows what is generated when the loader is in the process of searching for implicit layer manifest files on a Linux system:

```
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, the Vulkan loader is listing the searched folders, and that seven implicit layers were discovered in two different folders (`/home/${USER}/.local/share/vulkan/implicit_layer.d` and `/usr/share/vulkan/implicit_layer.d`). Remember, just because these implicit layers 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.

Later on, when layers are actually loaded, output similar to the following would be shown:

```
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)
```

Notice that there can be multiple tags before a message but all layer messages will always have the LAYER tag. The above output example indicates that the loader has loaded the library files for 2 layers and inserted them into the call chain. These layers are: VK_LAYER_KHRONOS_validation and VK_LAYER_MESA_device_select.

Then, when ***vkCreateInstance*** is called and the instance call chain is developed. The loader will produce output similar to the following that shows the established 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, the two previously loaded layers are used in the instance call chain. 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 implies 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.

Since VK_LAYER_MESA_device_select is an implicit layer, notice that the logging now lists the environment variable that can be used to disable it. In this case, "NODEVICE_SELECT" is the name of the disable environment variable. Therefore, to disable the VK_LAYER_MESA_device_select layer from being used by the loader, define an environment variable named “NODEVICE_SELECT” and set it equal to a non-0 value (1). This is one way that can be used to disable a problematic implicit layer.

Additionally, when ***vkCreateDevice*** is called, a similar device call chain output is produced similar to the above ***vkCreateInstance*** call chain. This will show you similar information, but also directly identify which physical device is being used.

Layer Filtering

A Vulkan loader -- built using headers 1.3.234 and newer -- provides more robust functionality for enabling and disabling either drivers or layers. Previously, to force enable certain layers, the environment variable “VK_INSTANCE_LAYERS” was used with the full layer names that were to be forced enabled. This environment variable only allowed the enabling of specific layers and provided no mechanism to disable problematic layers.

To resolve this problem, the loader 1.3.224 and above supports several new environment variables that make it easier to control the Vulkan environment on a user’s system. The new environment variables should help close some of the missing holes and provide developers with the tools they need to more quickly diagnose layer or driver issues.

New Environment Variables

The loader will look for two new environment variables that can be used to filter access to underlying Vulkan layers. These variables are:

- VK_LOADER_LAYERS_ENABLE
- VK_LOADER_LAYERS_DISABLE

There are some important new characteristics of these new environment variables: comma-delimited lists, globs, case-insensitive, and environment variable priority.

Comma-delimited lists

The new environment variables accept comma-delimited input allowing multiple strings to be chained together to enable a broader selection of layers to enable or disable.

Globs

Instead of requiring full layer names as the old “VK_INSTANCE_LAYERS” environment variable required, the new variables work with a limited glob format. Because of this, setting ‘VK_LOADER_LAYERS_ENABLE’ to “*validation*” will enable any layer with “validation” in its name including the “VK_LAYER_KHRONOS_validation” layer.

Glob support is limited to only the star (‘*’) character plus a substring. Wherever the star is, it will match any set of other characters. This provides the ability to choose one of 4 mechanisms for matching layer names using the filter string:

1. Prefix ‘string*’

    - Will match “string” at the beginning of the layer name only
2. Suffix ‘*string’
    - Will match “string” at the end of the layer name only
3. Substring “*string*”
    - Will match any occurrence of “string” in the layer name.
4. Whole string “string”
    - Layer name must match “string” exactly

Why globs?
Globs provide finer grain control over what layers are enabled/disabled. For example, if there were two layers: “VK_LAYER_FRANK_awesomeness” and “VK_LAYER_FRANK_awesomeness_2”, globs allow selecting one layer over the other, or even both simultaneously.

Case-insensitive

All these new variables assume the substrings included are not case-sensitive. So “Bob”, “bob”, BOB” all match the same substring.

Environment Variable Priority

When evaluating available layers using the new environment variables, the loader will evaluate the contents from “VK_LOADER_LAYERS_DISABLE” first, and then consider the contents of “VK_LOADER_LAYERS_ENABLE”. This allows disabling all layers and then selectively enabling only the specific layers of interest.

Using VK_LOADER_LAYERS_ENABLE

The ‘VK_LOADER_LAYERS_ENABLE’ environment variable is a comma-delimited list of substrings to search for known layers. Known layers are those layers that are already found by the loader taking into account default search paths and other environment variables such as ‘VK_LAYER_PATH’. If no layers are found with a layer name that matches any of the provided substrings, then no additional layers will be loaded.

NOTE: This is intended to replace the existing ‘VK_INSTANCE_LAYERS’ environment variable. Newer loaders will continue to support both ‘VK_LOADER_LAYERS_ENABLE’ and ‘VK_INSTANCE_LAYERS’ since older loaders will only support the latter. However, it is recommended to use ‘VK_LOADER_LAYERS_ENABLE’ whenever the loader supports it since it provides additional functionality.

For example:

```
set VK_LOADER_LAYERS_ENABLE=*validation,*recon*
```

The above would result in both the “VK_LAYER_KHRONOS_validation” and “VK_LAYER_LUNARG_gfxreconstruct” layers being enabled in the loader. But only if they are found in the list of available layers that are already detected by the loader on the current system.

It is possible to force all layers created by a single vendor to be enabled by setting “set VK_LOADER_LAYERS_ENABLE=*<vendor>*”. This could work for vendor values like “*Khronos*” or “*LunarG*”.

Using VK_LOADER_LAYERS_DISABLE

The ‘VK_LOADER_LAYERS_DISABLE’ environment variable is a comma-delimited list of substrings to search for known layers. If no layers are found with a layer name that matches any of the provided substrings, then no layers will be disabled. If there's a substring match, even if the layer is an implicit layer, then the layer will be disabled in the loader.
For example:

```
set VK_LOADER_LAYERS_DISABLE=*MESA*,*validation
```

The above would result in both the “VkLayer_MESA_device_select” and “VK_LAYER_KHRONOS_validation” layers being disabled in the loader, even though one is an implicit layer, and the other may be attempted to be loaded by an application as an explicit layer.

Special Layer Disable Substrings

To make things even easier, additional special-case values have been added to the accepted values of the disable environment variable. These are:

- ~all~
  - All layers are disabled
  - NOTE: This can also be done with the ‘*’ glob alone
- ~implicit~
  - All implicit layers are disabled
- ~explicit~
  - All explicit layers are disabled

Just as they state, they will either disable all layers (yes ALL of them), all implicit layers, or all explicit layers. The intent of this is purely for debugging purposes and is not intended for use in situations beyond that. This gives developers and debuggers the “safe mode” capability that many have requested.

These can even be combined with other substrings where applicable. For example:

```
set VK_LOADER_LAYERS_DISABLE=~implicit~,*validation
```

would disable all implicit layers and validation. However, doing the following:

```
set VK_LOADER_LAYERS_DISABLE=~implicit~,*MESA*
```

would have no difference since the MESA device select layer would already be disabled by the “~implicit~” keyword.

When disabling any layers, the loader will report any disabled layers to the logging output when layer logging is enabled in the following way:

```
WARNING | LAYER:  Implicit layer "VK_LAYER_MESA_device_select" forced disabled because
name matches filter of env var 'VK_LOADER_LAYERS_DISABLE'.
WARNING | LAYER:  Implicit layer "VK_LAYER_AMD_switchable_graphics_64" forced disabled
because name matches filter of env var 'VK_LOADER_LAYERS_DISABLE'.
WARNING | LAYER:  Implicit layer "VK_LAYER_Twitch_Overlay" forced disabled because name
matches filter of env var 'VK_LOADER_LAYERS_DISABLE'.
```

This output can be used later when determining which layers to manually enable later if needed.

WARNING: Disabling either “~all~” (or ‘*’) or “~explicit~” layers will also disable layers that the application is intentionally trying to enable during **vkCreateInstance**. This could cause applications to crash or generally misbehave since the functionality they depend upon may be removed. Because of this, those settings should be used sparingly.

Combining the Layer Disable and Enable Filters

The new filter environment variables can be combined together. The disable environment variable is evaluated first, and then the enable environment variable is evaluated. The benefit of this ordering is that all layers can be disabled with one setting and then only the specific layers can be re-enabled afterward.

For example, to disable all implicit layers and then selectively re-enable all other layers by a hardware company:

```
set VK_LOADER_LAYERS_DISABLE=~implicit~
set VK_LOADER_LAYERS_ENABLE=*AMD*
```

Driver Filtering

Also starting with a Vulkan loader built using headers 1.3.234, it now provides similar filtering capabilities on drivers. This can be useful if all previous investigations still show no issues. If the system has more than one Vulkan driver available, the issue could be caused by driver interactions as well. Using the driver versions of the filter environment variables, a developer can control which drivers are used and which are ignored. Because Vulkan drivers don’t have a

name that is easy for the Vulkan loader to find, the Vulkan loader filters drivers based on their filename. Again, this is information that can be found by looking at the Vulkan loader logging.

The driver environment variables are:

- VK_LOADER_DRIVERS_SELECT: Used to select a given subset of valid drivers found by the loader
- VK_LOADER_DRIVERS_DISABLE: Used to disable drivers found by the loader

Driver Logging

Updated logging has been added to identify drivers being used when “driver” is added to the list of items provided to VK_LOADER_DEBUG.

Example driver search:

```
DRIVER:           Found ICD manifest file
/usr/share/vulkan/icd.d/intel_icd.x86_64.json, version "1.0.0"
DRIVER:           Found ICD manifest file /usr/share/vulkan/icd.d/lvp_icd.x86_64.json,
version "1.0.0"
DRIVER:           Found ICD manifest file /usr/share/vulkan/icd.d/nvidia_icd.json,
version "1.0.0"
```

This information can be used to filter out specific drivers when investigating possible driver issues.

Using VK_LOADER_DRIVERS_SELECT

This is similar to ‘VK_LOADER_LAYERS_ENABLE’ except that this list is exclusive. If this variable is defined, then only the driver names matching the provided globs will be added to the list of available drivers.

For example, to enable only Intel drivers on Linux:

```
set VK_LOADER_DRIVERS_SELECT=*intel*
```

In the above case, all other drivers would be ignored.

Using VK_LOADER_DRIVERS_DISABLE

This is similar to the ‘VK_LOADER_LAYERS_DISABLE’ environment variable except that there are no special-case keywords to disable drivers. All drivers can be disabled by default using the “*****” value.

For example:

```
set VK_LOADER_DRIVERS_DISABLE=*lvp*
```

The above would disable the LLVMPipe software implementation.

Combining Both the Driver Disable and Select Filters

The new filter environment variables can be combined together. The disable environment variable is evaluated first, and then the select environment variable is evaluated. Just as with layers, the benefit of this ordering is that all drivers can be disabled with one setting and then only the specific drivers can be selected afterward.

For example, to disable all drivers except for Nvidia:

```
set VK_LOADER_DRIVERS_DISABLE=*
set VK_LOADER_DRIVERS_SELECT=*nvidia*
```

More information on how to filter drivers can be found in the Vulkan desktop loader Github repository under the docs folder. Specifically in the LoaderDriverInterface.md markdown file.

Debugging A Possible Layer Issue Checklist

Here's a quick checklist to follow next time you have a possible layer issue:

1. Enable loader logging with
   `set VK_LOADER_DEBUG=error,warn,layer,driver`
2. Disable all implicit layers:
   `set VK_LOADER_LAYERS_DISABLE=~implicit~`
3. Run again
4. If the issue goes away, then it is likely a layer issue
   a. Repeat by enabling one layer at a time with VK_LOADER_LAYERS_ENABLE
   b. If the problematic layer is an implicit layer:
      i. Globally define the disable environment variable listed by the loader for that layer
      ii. Report the bug to the layer developer.
5. If the issue persists, try disabling one or more drivers using VK_LOADER_DRIVERS_DISABLE

Conclusions

Layers are a very powerful, and useful part of the Vulkan ecosystem. To aid in diagnosing failures due to misbehaving layers, the Vulkan Loader has been enhanced with logging improvements as well as layer and driver filters that gives application developers the tools they need to identify and resolve their issue.

Additionally, if there are any suggestions on ways the Vulkan loader or ecosystem could improve layer handling and debugging, please bring any them to the attention of the Vulkan loader by filing a feature request or issue on Github.

Happy Layering!