The Vulkan Profiles Tools

Better Vulkan Application Portability thanks to Vulkan Profiles

[Christophe Riccio](), [LunarG]()

January 2024

Why do *Vulkan Profiles* matter?

The discussion regarding shipping a cross-platform application is typically framed around a perfectly homogenous ecosystem against an overly heterogeneous ecosystem. The *Vulkan* ecosystem contains a huge diversity of hardware vendors, devices, and drivers that keep evolving, making it hard to be sure that any given Vulkan application will work on all targeted devices.

We believe at LunarG that expecting the ecosystem to be homogenous is unrealistic. This problem is not new and was also massive in the *OpenGL/OpenGL ES* ecosystem. *Vulkan* is a cross-platform and evolving industry standard enabling developers to target a wide range of devices with the same graphics API.

Instead, *Vulkan Profiles* look at the problem with a different paradigm: a Vulkan application is not designed to target the entire ecosystem, but has ***a domain of relevance***. The different actors in the ecosystem should instead better communicate (including programmatically) about Vulkan support and Vulkan requirements. This is formalized by Vulkan Profiles.

Explicit Vulkan capability requirements

The *Vulkan Profiles* make the Vulkan capability requirements explicit between the application and the domain of relevance of the Vulkan application. We should expect a co-evolution of the application and its domain, which can be formalized by profiles and revisions of the profiles.

Conceptually, *Vulkan Profiles* can be understood as the explicit expression and formalization of Vulkan capability requirements, which provide clear communication of these requirements within the Vulkan community and across Vulkan components in the ecosystem.

With this approach, we think *Vulkan Profiles* help create more portable Vulkan applications within a specific Vulkan domain; but we acknowledge Vulkan Profiles are as relevant as their adoption by the Vulkan ecosystem.

Projects using Vulkan Profiles

We are interested in knowing how Vulkan developers are using Vulkan Profiles. Contact us to let us know!

Unreal Engine: A real-time 3D creation tool

The Unreal Engine uses the Android and Desktop profiles to specify the feature requirements of some rendering code paths. Specifically, it's used to check the support of the required Ray Tracing capabilities in the user's system. The code base uses the Vulkan Profiles API library to

exit gracefully if the requirements are not satisfied. During the engine development process, the required Vulkan capabilities are gathered when the execution hits an assert. The validation layer checks for capability requirements scattered across the code, and these are then consolidated into profiles. When the application user starts an execution, the capability requirements are fully checked with the profile, ensuring they will not hit those asserts of missing capabilities.

DXVK: A Vulkan-based translation layer for Direct3D 9/10/11

This is a Vulkan-based translation layer for Direct3D 9/10/11 which allows running Direct3D applications on Linux using Wine.

The VP_DXVK_requirements profiles are used to list all the Vulkan capabilities required by the device's Vulkan driver to be able to run Direct3D 9, Direct3D 10.1 and Direct3D 11 for most relevant feature levels. For example, the `VP_DXVK_d3d11_level_11_1_baseline` profile lists the requirements to be able to run a Direct3D 11 Feature Level 11.1 application. The `VP_DXVK_d3d11_level_11_1_optimal` profile is similar to the `VP_DXVK_d3d11_level_11_1_baseline` but also includes Vulkan capabilities to get the best performance possible from the DXVK Direct3D 11 implementation.

vkd3d-proton: An implementation of the full Direct3D 12 API on top of Vulkan

vkd3d-proton is a fork of VKD3D, which aims to implement the full Direct3D 12 API on top of Vulkan. The project serves as the development effort for Direct3D 12 support in Proton.

The VKD3D profiles are used to list all the Vulkan capabilities required by the device's Vulkan driver to be able to run Direct3D 12 for most relevant feature levels. For example, the `VP_D3D12_FL_12_2_baseline` profile lists the requirements to be able to run a Direct3D 12 Feature Level 12.2 application. The `VP_D3D12_FL_12_2_optimal` profile is similar to the `VP_D3D12_FL_12_2_baseline` but also includes Vulkan capabilities to get the best performance possible from VKD3D-Proton Direct3D 12 implementation. VKD3D-Proton also has per-vendor dedicated profiles such ad `VP_D3D12_maximum_radv` and `VP_D3D12_maximum_nv` which represent the maximum feature set VKD3D-Proton can use on RADV and NVIDIA driver, respectively.

Zink: a Gallium driver that emits Vulkan API calls

The Zink driver is a Gallium driver that emits Vulkan API calls instead of targeting a specific GPU architecture. This can be used to get full desktop OpenGL support on devices that only support Vulkan.

The `VP_ZINK_requirements` profiles are used to list all the Vulkan capabilities required by the device Vulkan driver to be able to run each version of OpenGL. For example, the

`VP_ZINK_gl46_baseline` profile lists the requirements to be able to run an OpenGL 4.6 application. The `VP_ZINK_gl46_optimal` profile is similar to the `VP_ZINK_gl46_baseline` but also includes Vulkan capabilities to get the best performance possible from the Zink OpenGL implementation.

Android Baseline Profiles

The Android Baseline profiles addressed the challenge of determining what functionality Vulkan developers can rely on being present in Android devices. From sets of Vulkan extensions, features, formats, and limits that were found on the vast majority of active Android devices in 2021 and 2022, the Android Baseline 2021 and the Android Baseline 2022 profiles were created. These profiles can be used to filter the Android Vulkan applications on the Google Play store for devices not supporting the capabilities of these profiles.

Android 15 Minimums Profile

A collection of functionality that is mandated for chipsets that launch (or renew Google Requirements Freeze) on Android 15. This is effectively an Android Roadmap profile.

Khronos Roadmap Profiles

The Khronos Roadmap profiles are milestones referenced by all Vulkan Working Group hardware vendors actively developing mid-to-high-end devices for smartphone, tablet, laptop, console, and desktop platforms. These vendors have committed to supporting this milestone, starting with several shipping products in 2022.

Different types of Vulkan Profiles

Vulkan Profiles can be applied for a multitude of use cases, including:

- *Roadmap profiles*: to express guidance on the future direction of Vulkan devices.
- *Platform profiles*: to express the Vulkan support available on different platforms.
- *Device profiles*: to express the Vulkan support of a single Vulkan driver on a Vulkan device.
- *Architecture profiles*: to express the Vulkan support of a class of GPUs.
- *Engine profiles*: to express some rendering code paths requirements of an engine.
- Drivers bugs profiles: to express capabilities that can’t be used by an application.
- Etc.

The Vulkan Profiles Tools Components

LunarG provides the *Vulkan Profiles Tools* as part of the *Vulkan SDK* so that Vulkan application developers may leverage *Vulkan Profiles* during Vulkan application development and delivery. Developers can create portable Vulkan applications in terms of Vulkan capabilities, which include extensions, features, properties, formats, and queue properties.

Of course, the Vulkan Profiles only handle portability in terms of Vulkan capabilities, but Vulkan profiles may be used to document known bugs in Vulkan drivers.

The Vulkan Profiles Tools include the following components:

- The Vulkan Profiles schema
  - A JSON data format to exchange Vulkan capabilities: extensions, features, properties, formats, and queue properties.
  - Each revision of Vulkan API is represented by a schema that supersedes older versions of Vulkan API.
  - The schema covers Vulkan 1.3 and all extensions.
- The Vulkan Profiles merge script
  - A Python script to merge multiple Vulkan Profiles into a single Vulkan Profile using either an intersection or union operation on the Vulkan capabilities listed by the source Vulkan Profiles.
- The Vulkan Profiles layer
  - A layer used during application development to ensure adherence to the requirements of a chosen Vulkan Profile.
  - It simulates but doesn't emulate Vulkan capabilities. It works together with the Validation layer which reports errors when using capabilities not exposed by the Vulkan developer system.
  - The layer requires a Vulkan 1.1 driver.
- The Vulkan Profiles library
  - A header-only, C++ library to use Vulkan Profiles in Vulkan applications.
  - The library allows checking Profiles support on a device and creating a `VkDevice` and VkInstance instances with the profile features and extensions enabled.
  - The library requires a Vulkan 1.0 driver that supports the `VK_KHR_get_physical_device_properties2` extension.
  - A Vulkan sample is available for demonstrating Vulkan Profiles library usage.
- The Vulkan Profiles comparison table
  - Human-readable format of Vulkan Profiles in a table to simplify comparison.

Furthermore, the Vulkan SDK includes implementations of some Vulkan Profiles using the Vulkan Profiles JSON Schema:

- `VP_KHR_roadmap.json`

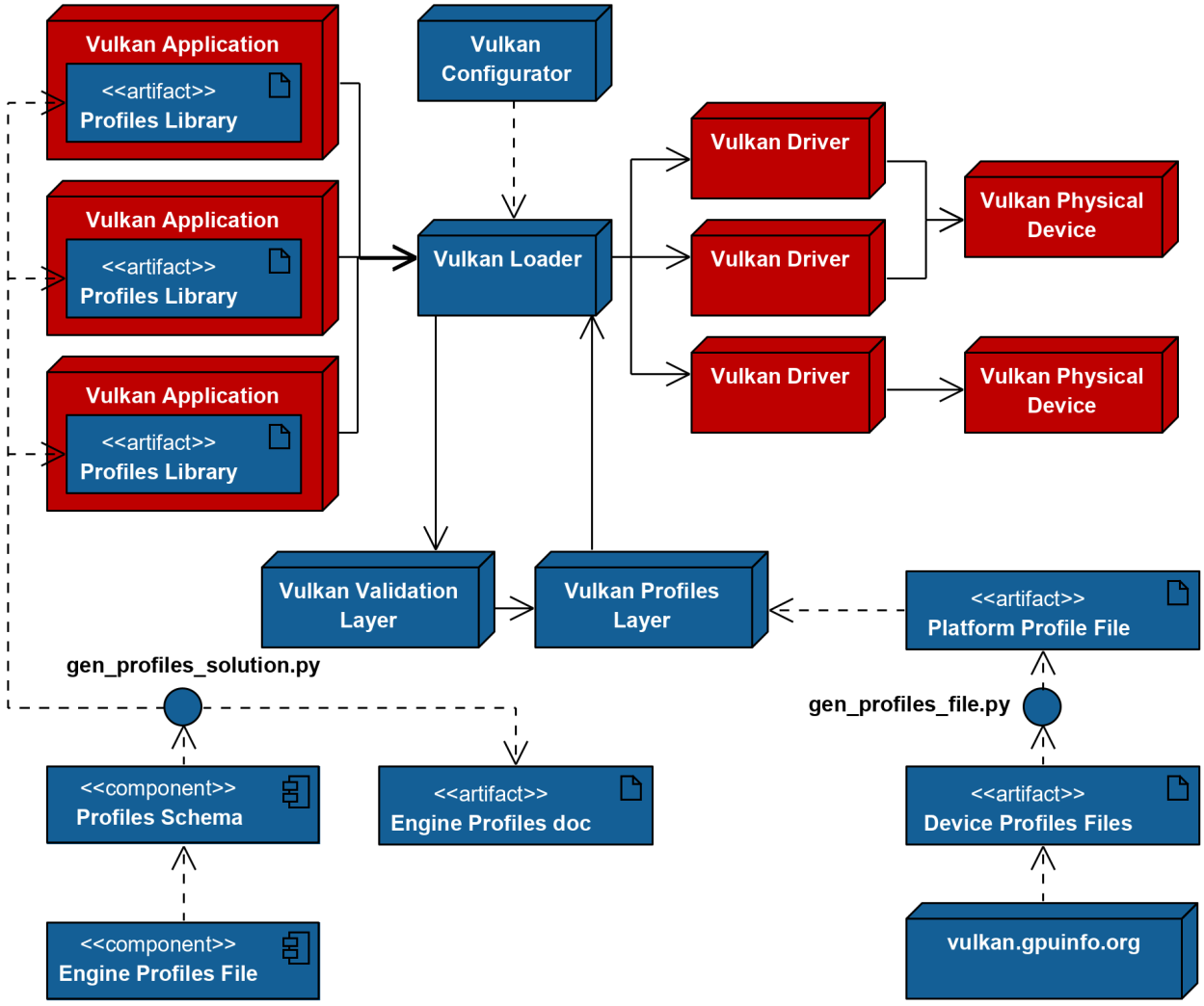
- VP_LUNARG_minimum_requirements.json
- VP_ANDROID_baseline_2021.json
- VP_ANDROID_baseline_2022.json
- VP_ANDROID_15_minimums.json
- VP_LUNARG_desktop_baseline_2022.json
- VP_LUNARG_desktop_baseline_2023.json
- VP_LUNARG_desktop_baseline_2024.json

These *Vulkan Profiles JSON files* are written against the *Vulkan Profiles JSON schema*. There is a *Vulkan Profiles JSON schema* file per *Vulkan Header* version. The Vulkan SDK contains the *Vulkan Profiles JSON schema* corresponding to the Vulkan Header version of that Vulkan SDK. The *Vulkan Profiles JSON schema* is located in the `share/vulkan/registry` directory of the Vulkan SDK. All the schema can easily be downloaded by cloning the Khronos-Schema repository.

The *Vulkan Profiles API* is not part of the Vulkan specification, but part of a library, for a very specific reason: we wanted the Vulkan Profiles solution to be effectively usable on day one. Hence, it should work with current Vulkan devices currently owned by users.

The solution to this problem is to not deliver the *Vulkan Profiles API* through the Vulkan drivers (that can't be updated easily on many platforms), but to deliver the Vulkan Profiles API as part of the Vulkan application codebase.

As a result, the Vulkan Profiles library is compatible with any Vulkan 1.0 drivers that support `VK_KHR_get_physical_device_properties2` and can be leveraged simply by including it in the Vulkan application codebase.



A typical setup of the Vulkan Profiles Tools components on the Vulkan developer system

- Vulkan application side:
  - The Vulkan applications are built with the *Vulkan Profiles API library*.
  - The *Vulkan Profiles API library* is generated using the `gen_profiles_solution.py` python script and the Engine Profiles file.
  - The *Engine Profiles JSON file* is written by the Vulkan application developers based on the actual rendering code paths Vulkan capabilities requirements.
  - The *Engine Profiles JSON file* is written and validated against a *Vulkan Profiles schema*.
  - The *Engine Profiles Definition doc* is a markdown file generated using the `gen_profiles_solution.py` python script.
- Vulkan development environment side:
  - The *Vulkan Validation layer* and the *Vulkan Profiles layer* are set up using the *Portability* built-in configuration in *Vulkan Configurator*.

- The *Vulkan Profiles layer* is configured so that the system simulates the Vulkan capabilities listed in the *Platform Profile file*.
- The *Platform Profile file* is generated using the `gen_profiles_files.py` python script and *Device Profiles files*.
- The *Device Profiles files* are selected and downloaded from GPUInfo.org. The criteria for the *Device Profiles files* depends on the developers of the Vulkan application and the device ecosystem they want to reach for typically technical and economical reasons.

Vulkan Capability minimum requirements

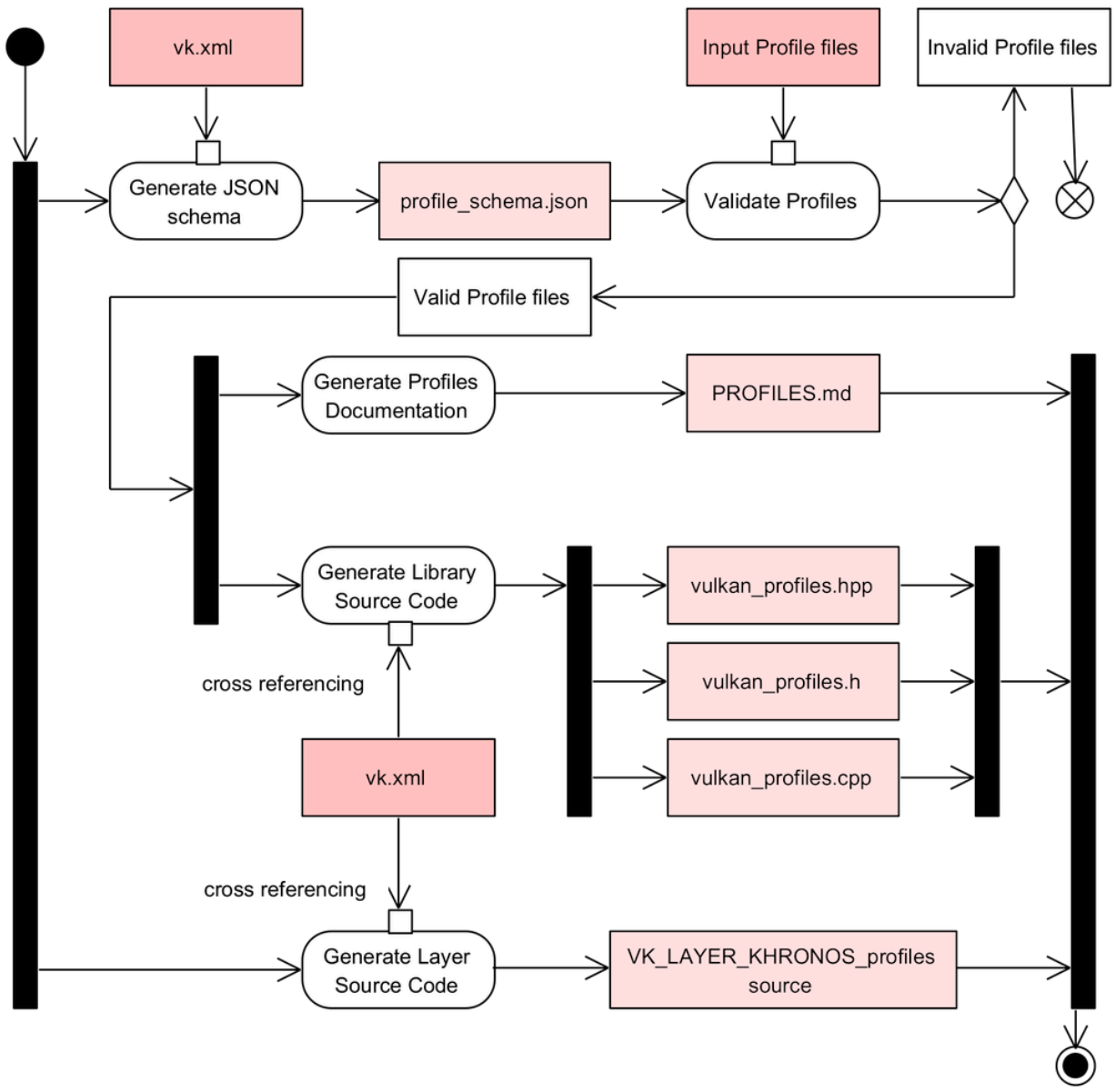
The Vulkan minimum requirements of the Vulkan specification are described in the features, limits and formats and additional capabilities sections of the specification. Unfortunately, there is no programmable way to access these minimum requirements. Ideally, this information would be included in the vk.xml. To workaround this issue we create the `VP_LUNARG_minimum_requirements` profiles that list the minimum requirements for each Vulkan version.

The Vulkan Profiles Tools Code Generation

Considering the complexity of the Vulkan ecosystem, there isn't a single Vulkan Profile that can fit all needs. As a result, on top of the predefined Vulkan Profiles, the Vulkan Profiles Tools solution is designed around the idea of code generation.

The Vulkan Profiles Tools are also generated against vk.xml (the canonical representation of the Vulkan specification) and Vulkan Profiles. This design guarantees that any Vulkan developer can regenerate the entire Vulkan Profiles Tools with any new Vulkan Header update or any set of Vulkan Profiles JSON files.

The diagram on the following page shows the Vulkan Profiles Tools generation pipeline with every produced component:



The *Vulkan Profiles* documentation

The *Vulkan Profiles* documentation can be used to easily read the requirements of a profile and compare multiple profiles side by side.

It can easily be regenerated and augmented with more profiles by simply copying the list of *Vulkan Profiles* files we want to document into the Profiles directory in the source.

Creating a Vulkan Profiles JSON files

Generating a Device Vulkan Profiles files

Vulkaninfo

Vulkaninfo provides a Vulkan Profiles file export allowing generating Device Vulkan Profiles directly from the console.

A use case from LunarG experience, we are using *Vulkaninfo* with the Vulkan Profiles JSON file output in our internal C.I. system so that we can get some Vulkan information on each C.I. run.

Another possible use case is to generate the JSON file, modify it to remove specific capabilities. Then with this modified Device Profiles JSON file and using the Vulkan Profiles layer, we can simulate the capabilities of this profile on our developer system to check that our Vulkan application falls back or fails correctly when these capabilities are not available on an end-user system.

Vulkaninfo is part of the Khronos Vulkan SDK.

To generate a device Vulkan Profiles JSON files from the system, we can use the command:

```
Unset
$ vulkaninfo --json -o my_device_profile.json
```

GPUInfo.org

The GPUInfo.org database is populated using the Vulkan Hardware Capability Viewer application, available for multiple platforms. It reads and displays Vulkan related information for the running implementation, and that data can then be uploaded to the database.

Each Vulkan driver entry in the GPUInfo.org can be exported in the form of a Vulkan Profiles JSON file.

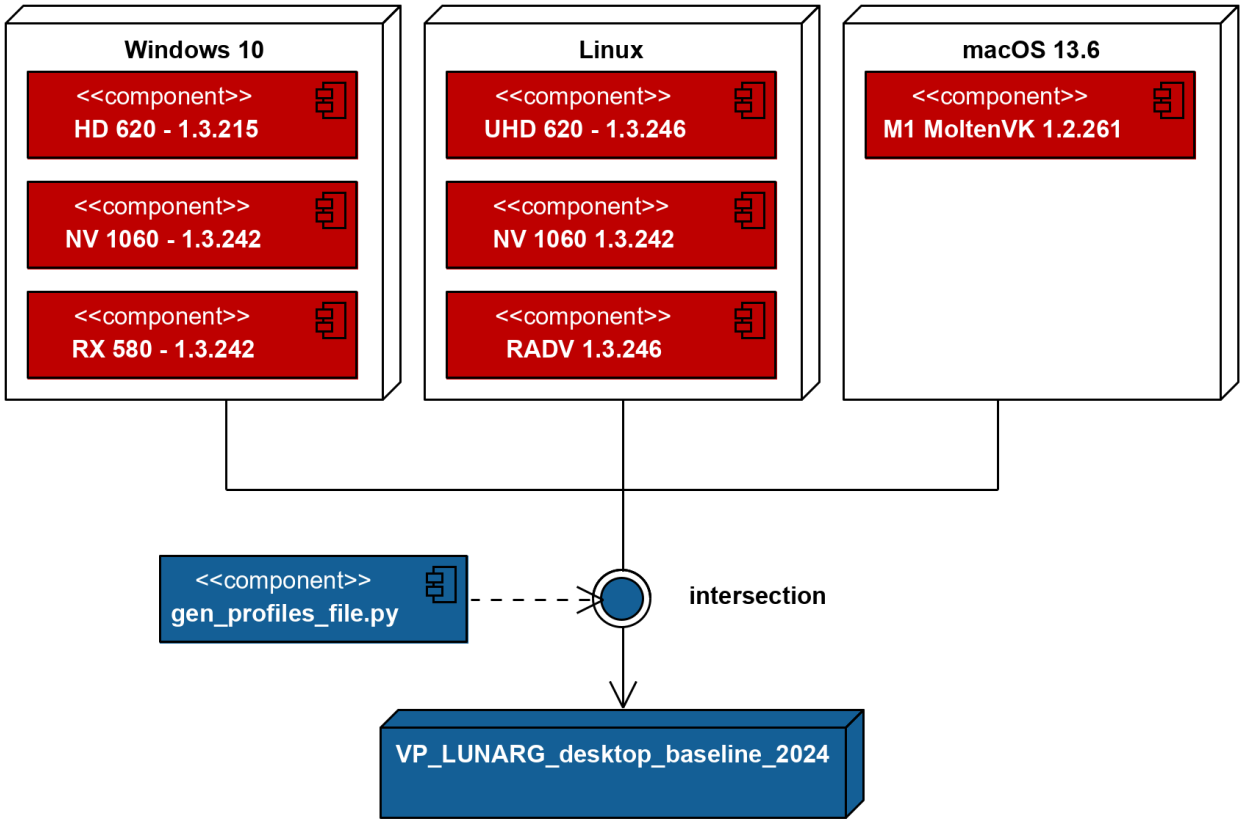
Vulkan Hardware Capability Viewer is part of the Khronos Vulkan SDK.

Generating a Platform Vulkan Profiles JSON files

The LunarG Desktop Baseline profiles were generated by downloading a set of Vulkan Device Profiles on GPUInfo.org and using a Python script to merge these Vulkan Device Profiles into Vulkan Platform Profiles by doing an intersection of all the capability of all Vulkan Device

Profiles : A platform profile reports the list of all the capabilities available in all Vulkan Device profiles.

This Python script is shipping as part of the Vulkan SDK in the `share/vulkan/registry` directory and is named `gen_profiles_file.py`.



As an example, the VP_LUNARG_desktop_baseline_2024 is a Vulkan Platform Profile generated from multiple Vulkan Device Profiles downloaded from GPUInfo.org.

The command to generate the LunarG Desktop Baseline profiles is :

```
Unset
python gen_profiles_file.py
  --registry vk.xml
  --input ./VP_LUNARG_desktop_baseline_2024
  --output-path ./VP_LUNARG_desktop_baseline_2024.json
  --output-profile VP_LUNARG_desktop_baseline_2024
  --profile-label "LunarG Vulkan Desktop Baseline 2024 profile"
  --profile-desc "A profile generated by the intersection of a collection of
GPUInfo.org device reports to support a large number of actual systems in the Vulkan
ecosystem. This profile is meant to be a usage example for Vulkan application
developer."
  --profile-date 2023-11-01
```

```
--profile-api-version "1.2.197"
--profile-required-profiles "VP_LUNARG_minimum_requirements_1_2"
--strip-duplicate-structs
```

We insist that the LunarG Desktop Baseline profiles are only usage examples of Vulkan Platform profiles. Nobody knows better what the set of platforms a Vulkan application should support than the Vulkan developer of that Vulkan application.

With this whitepaper, Vulkan developers should be able to produce their own platform profiles representing the ecosystem of Vulkan devices and drivers their Vulkan applications should support.

For more information on how to use `gen_profiles_file.py`, we can use the command :

```
Unset
python gen_profiles_file.py --help
```

Creating a Vulkan Profiles JSON files manually

The *Vulkan Profiles schema* is a JSON file generated using the vk.xml file. For each *Vulkan Header* revision, we can generate a *Vulkan Profiles schema* that is updated on the Khronos Vulkan Schema page and Khronos Schema git repository. Generating a new *Vulkan Profiles schema* may be necessary to leverage Vulkan capabilities introduced with a new *Vulkan Header* revision.

The role of the *Vulkan Profiles schema* is to ensure that profile files, created in the ecosystem, will be syntactically valid files that can be consumed reliably by Vulkan tools and applications. Specifically, the schema validated the name of the structures and structures members so there can't be any spelling errors. Similarly, the schema lists all valid enum values.

Validating *Vulkan Profiles* JSON files against the schema can be performed using any tools typically used for this purpose, including the web-based validators such as:

- http://www.jsonschemavalidator.net/
- https://json-schema-validator.herokuapp.com/
- https://jsonschemalint.com/#/version/draft-04/markup/json/

This can also be done with C++ libraries such as Valijson and most likely any JSON schema validation library already integrated into a Vulkan project codebase.

```
Unset
{
    "$schema": "https://schema.khronos.org/vulkan/profiles-0.8.2-276.json#",
    "capabilities": {
        "requirements_roadmap2022": {
            "features": {
                "VkPhysicalDeviceFeatures": {
                    "multiDrawIndirect": true,
                    "shaderInt16": true,
                    "shaderImageGatherExtended": true
                }
            },
            "properties": {
                "VkPhysicalDeviceProperties": {
                    "limits": {
                        "timestampComputeAndGraphics": true,
                        "maxColorAttachments": 8,
                        "maxBoundDescriptorSets": 7
                    }
                }
            }
        },
        "requirements_roadmap2024": {
            "features": {
                "VkPhysicalDeviceVulkan12Features": {
                    "shaderInt8": true,
                    "shaderFloat16": true,
                    "storageBuffer8BitAccess": true
                }
            },
            "properties": {
                "VkPhysicalDeviceVulkan12Properties": {
                    "shaderRoundingModeRTEFloat16": true,
                    "shaderRoundingModeRTEFloat32": true
                }
            }
        }
    },
    "profiles": {
        "VP_EXAMPLE_example_2024": {
            "version": 1,
            "api-version": "1.3.276",
            "label": "Vulkan Example 2024 profile",
            "description": "Description of Example 2024 profile",
            "profiles": [
                "VP_EXAMPLE_roadmap_2022"
            ],
            "capabilities": [
                "requirements_roadmap2024"
            ]
        },
        "VP_EXAMPLE_example_2022": {
            "version": 1,
            "api-version": "1.3.204",
```

```
            "label": "Vulkan Example 2022 profile",
            "description": "Description of Example 2022 profile",
            "contributors": {
                "Christophe Riccio": {
                    "company": "LunarG",
                    "email": "christophe@lunarg.com",
                    "contact": true
                }
            },
            "history": [
                {
                    "revision": 1,
                    "date": "2021-12-08",
                    "author": "Christophe Riccio",
                    "comment": "Initial revision"
                }
            ],
            "profiles": [
                "VP_LUNARG_minimum_requirements_1_3"
            ],
            "capabilities": [
                "requirements_roadmap2022"
            ]
        }
    }
}
```

Example of Vulkan Profiles file

A Vulkan Profiles JSON file has two main sections represented by respectively the `capabilities` and `profiles` elements.

The `capabilities` element just stores blocks of Vulkan capabilities. These blocks may be referenced or not by the profiles specified in the `profiles` elements.

The `profiles` elements contain elements that references required profiles and capabilities blocks. To interpret the profile, it's important to consider that if a required profile sets a capability and then a required capabilities block sets the same capability then the value from the required capability block overrides the value from the required profile. Similarly, the order in the array of the required profiles matters because the values of the following profile overrides the value of the previous profile. This applies also for the order in the array of required capabilities blocks.

Limitations of *Vulkan Profiles schema* validation

Unfortunately, a profile file would pass schema validation even if it requires a minimum Vulkan API version but uses a Vulkan structure that was introduced after that specified *Vulkan Header* revision. This makes the profile definition infringe upon its own requirements, which is indeed incorrect. Unfortunately, vk.xml is a snapshot of a *Vulkan Header* revision for a specific Vulkan

API version, so it doesn't store with which *Vulkan Header* revision a Vulkan capability was introduced.

For this reason, we recommend using the *Vulkan Profiles* schema revision that matches the *Vulkan Profile* API version minimum requirements. Many revisions of the profiles schema for *Vulkans Header* are available on Khronos.org.

Similarly, the Profiles JSON Schema can't validate a lot of the semantic aspects. For example, the `runtimeDescriptorArray` Vulkan feature can be enabled using multiple structures: `VkPhysicalDeviceDescriptorIndexingFeaturesEXT`, `VkPhysicalDeviceDescriptorIndexingFeatures`, and `VkPhysicalDeviceVulkan12Features`. Syntactically, all these structures can be used simultaneously for the definition of a profile; but what happens if they are specified with different values?

To address these cases, we can use the *Vulkan Profiles* layer which reports warning messages.

Human readable Vulkan Profiles documentation

The Vulkan SDK provides a human readable documentation for Vulkan Profiles.

This table can be generated for any set of profiles using the following command:

```
Unset
python gen_profiles_solution.py
  --registry vk.xml
  --input ./my_profiles/
  --output-doc ./PROFILES.md
```

For more information, use the following command:

```
Unset
python gen_profiles_solution.py --help
```

Using the *Vulkan Profiles layer*

Simulation vs. Emulation

The primary function of the *Vulkan Profiles Layer* is to simulate the capabilities of a device, modifying device responses to Vulkan query function calls by the application. Of course, the underlying device or driver functionality is never actually changed; they merely appear to have the capabilities of a different device or driver.

Effectively, the *Vulkan Profiles layer* is used for downgrading the Vulkan application developers' system capabilities.

This is different from emulation which would change the actual behavior of the underlying device or driver to match that of a different device or driver. In all but one case, the *Profiles layer* simulates changes and leaves it up to the *Validation Layer* to inform the developer about functions that do not adhere to the proper limits.

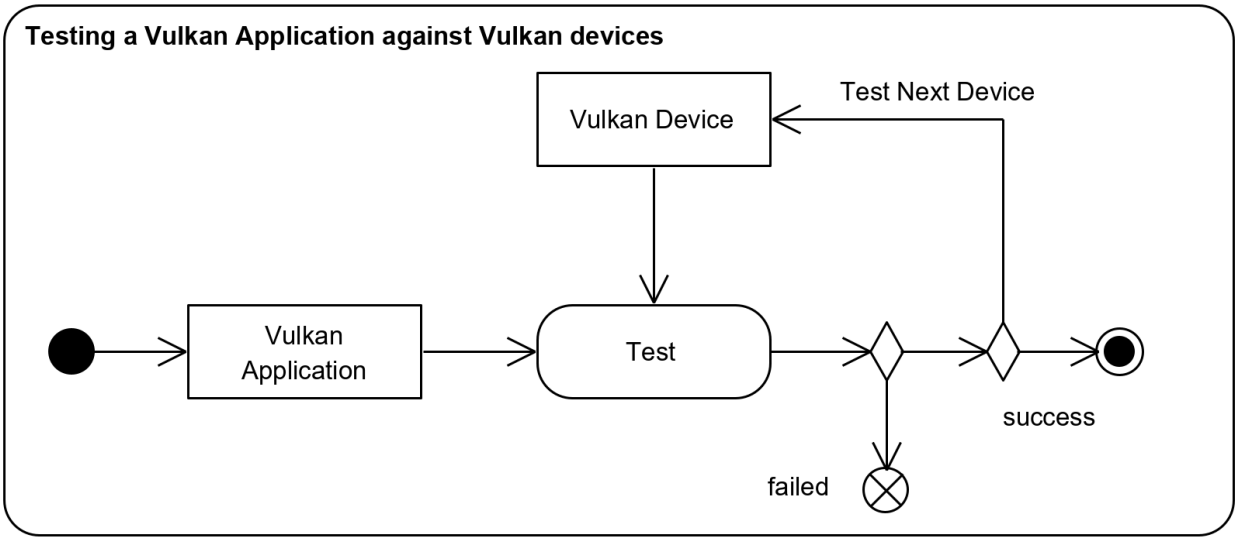
The one exception is the portability subset extension emulation, which causes the *Profiles Layer* to add the `VK_KHR_portability_subset` extension to the device extensions list, and pre-populate the `VkPhysicalDevicePortabilitySubsetPropertiesKHR` and `VkPhysicalDevicePortabilitySubsetFeaturesKHR` structures of this extension with default values.

The `VK_KHR_portability_subset` emulation can be controlled using the Profiles layer settings.

Reducing Vulkan application development time

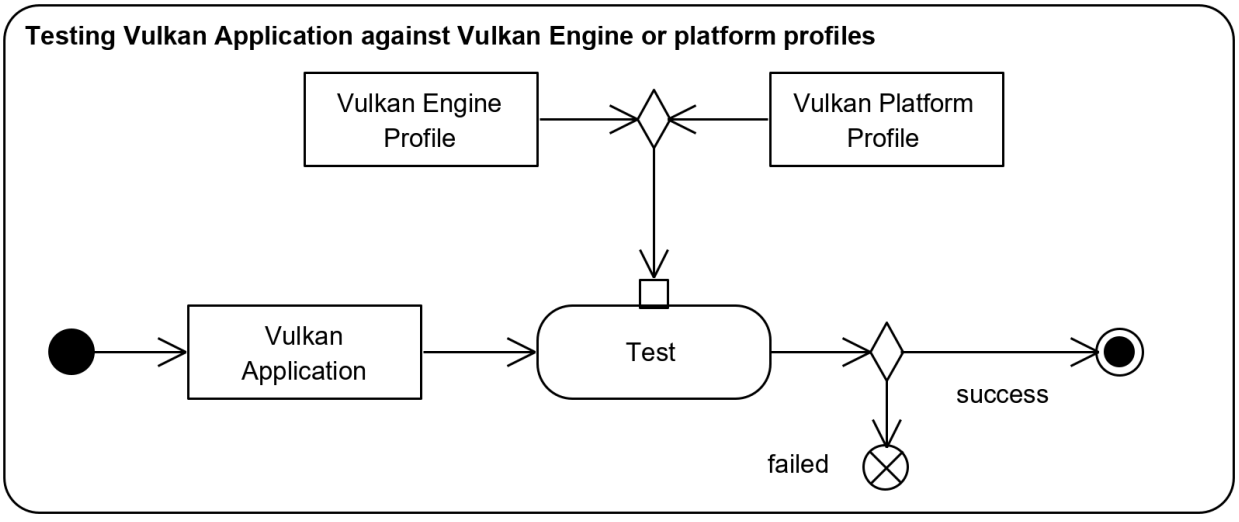
The Vulkan Profiles layer is expected to be used during Vulkan application development and testing. It aims at drastically improving the way we test our Vulkan applications across a wide range of hardware capabilities.

Typically when developing a Vulkan application, we need to check that our Vulkan application works on a set of platforms, devices, and even Vulkan driver versions. However, this process can be particularly tedious and time-consuming which translates into cutting some testing or reducing support of old Vulkan drivers.



Typical testing strategy, one device and driver at a time

The *Vulkan Profiles* and the *Vulkan Profiles* layer enable a new strategy: instead of checking capabilities of a device and driver set at a time, the solution allows checking an entire range of devices and drivers at a time by checking against a Vulkan profile that represents all these devices, or better, the Vulkan application requirements directly.



Testing Vulkan capabilities support against a Vulkan profile

The *Vulkan Profiles* layer can be used for many use cases:

- Using continuous integration to ensure that the Vulkan application never adds unintentional Vulkan capabilities requirements.
- Verifying that the Vulkan application falls back correctly when a driver doesn't support a capability, without updating the drivers or recompiling the Vulkan application.

- Verifying whether a Vulkan application behavior on a machine is due to the capabilities of that machine.
- Verifying the Vulkan application works on a less capable Vulkan device than the Vulkan developer device.
- Verifying the Vulkan Profile is well formed, with no unexpected duplicated references of Vulkan capabilities.
- Excluding device extensions and image formats to validate the robustness of the Vulkan application.
- Etc.

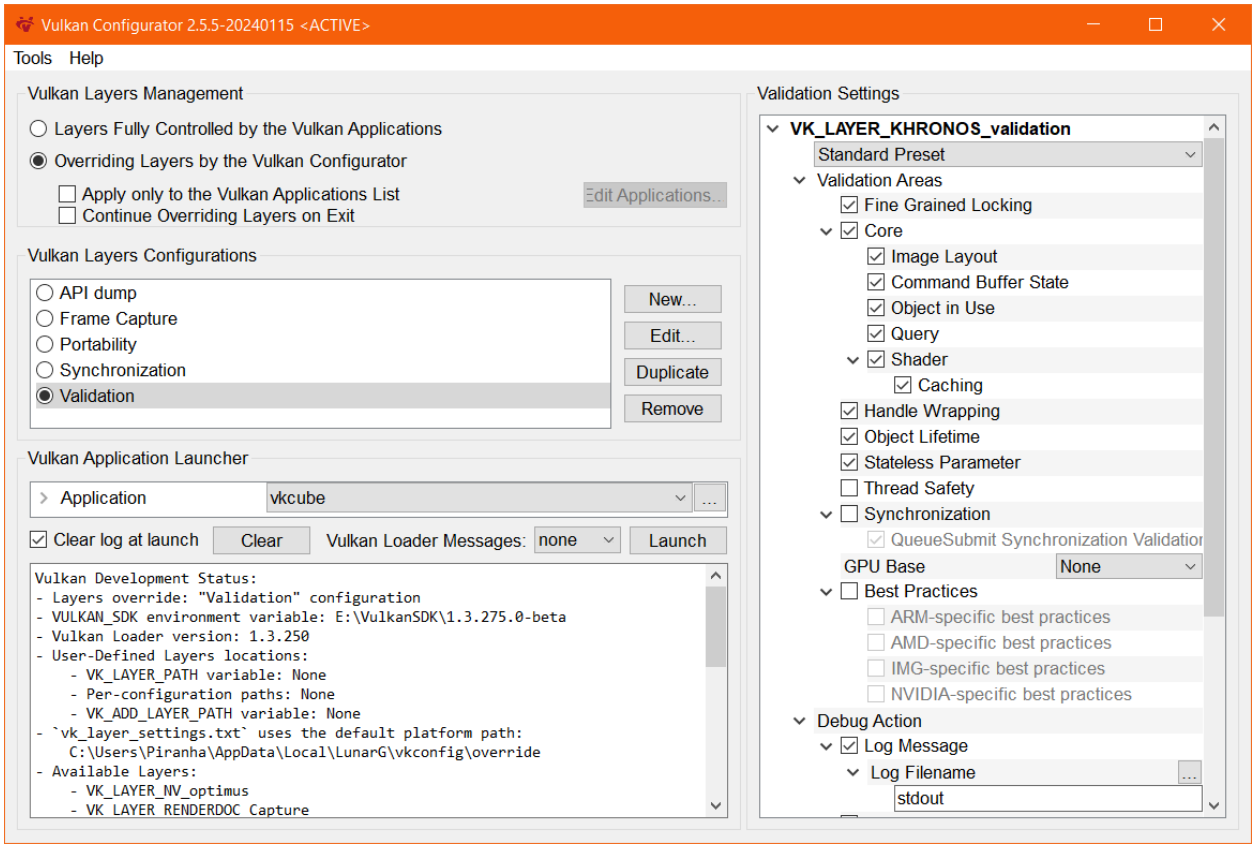
Enabling the *Profiles layer* using *Vulkan Configurator*

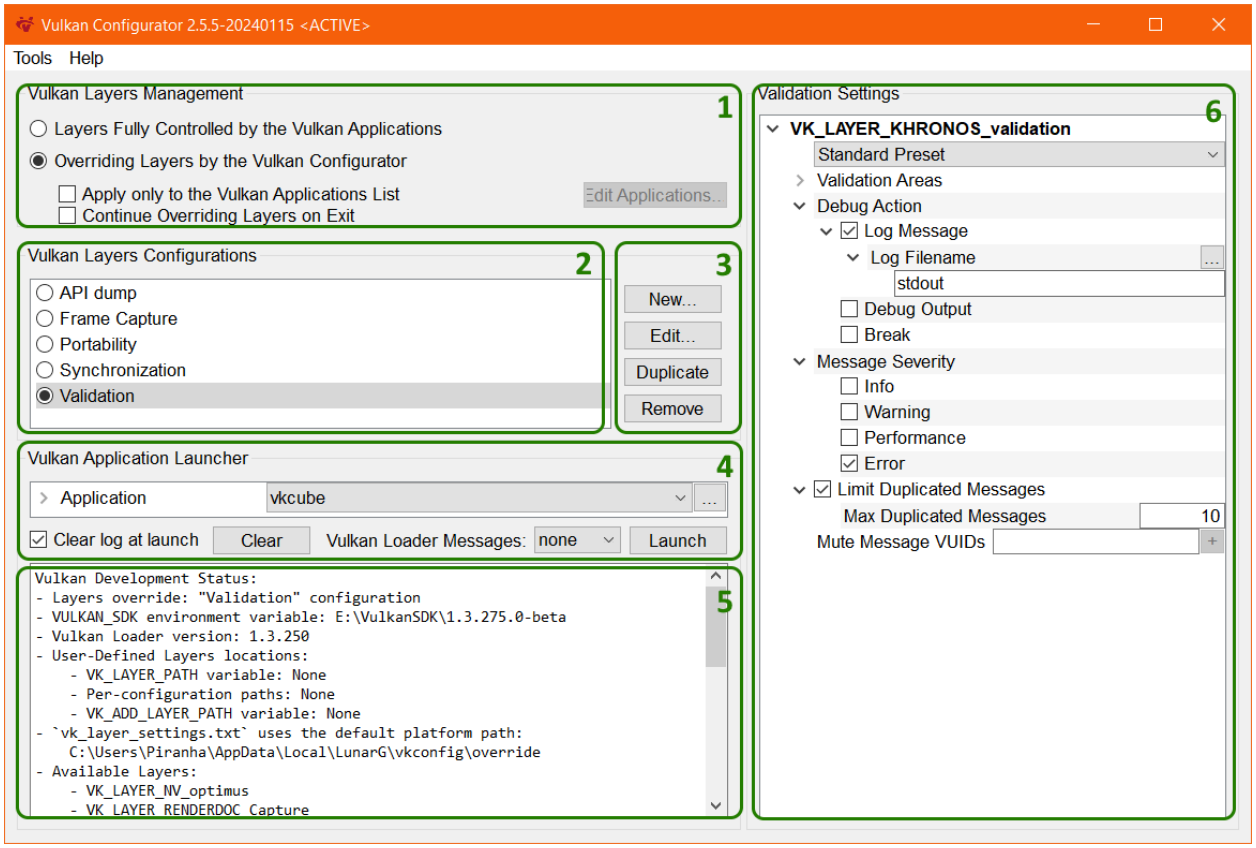
We highly recommend using *Vulkan Configurator* to use *Vulkan Layers* to improve Vulkan application development effectiveness.

Before *Vulkan Configurator*, a Vulkan developer would have to configure the layers either programmatically or by using environment variables specified by the layers' documentation, which required a significant and continuous learning curve as the Vulkan layers' capabilities evolved.

Vulkan Configurator was created to present the Vulkan layers with an intuitive interface enabling developers to use layer features with existing Vulkan applications, instantly and dramatically reducing development iteration time, as no compilation, no learning of the new settings, and no tracking of the new features are required. The features are directly available in the GUI.

When we open *Vulkan Configurator*, we are greeted with the following window.


Vulkan Configurator 2.5.5-20240115 <ACTIVE>
Tools Help
Vulkan Layers Management
Layers Fully Controlled by the Vulkan Applications
Overriding Layers by the Vulkan Configurator
Apply only to the Vulkan Applications List
Continue Overriding Layers on Exit
Edit Applications...
Vulkan Layers Configurations
API dump
Frame Capture
Portability
Synchronization
Validation
New...
Edit...
Duplicate
Remove
Vulkan Application Launcher
Application
vkcube
Clear log at launch
Clear
Vulkan Loader Messages:
none
Launch
Vulkan Development Status:
- Layers override: "Validation" configuration
- VULKAN_SDK environment variable: E:\VulkanSDK\1.3.275.0-beta
- Vulkan Loader version: 1.3.250
- User-Defined Layers locations:
- VK_LAYER_PATH variable: None
- Per-configuration paths: None
- VK_ADD_LAYER_PATH variable: None
- `vk_layer_settings.txt` uses the default platform path:
C:\Users\Piranha\AppData\Local\LunarG\vkconfig\override
- Available Layers:
- VK_LAYER_NV_optimus
- VK_LAYER_RENDERDOC_Capture
Validation Settings
VK_LAYER_KHRONOS_validation
Standard Preset
Validation Areas
Fine Grained Locking
Core
Image Layout
Command Buffer State
Object in Use
Query
Shader
Caching
Handle Wrapping
Object Lifetime
Stateless Parameter
Thread Safety
Synchronization
QueueSubmit Synchronization Validation
GPU Base
None
Best Practices
ARM-specific best practices
AMD-specific best practices
IMG-specific best practices
NVIDIA-specific best practices
Debug Action
Log Message
Log Filename
stdout



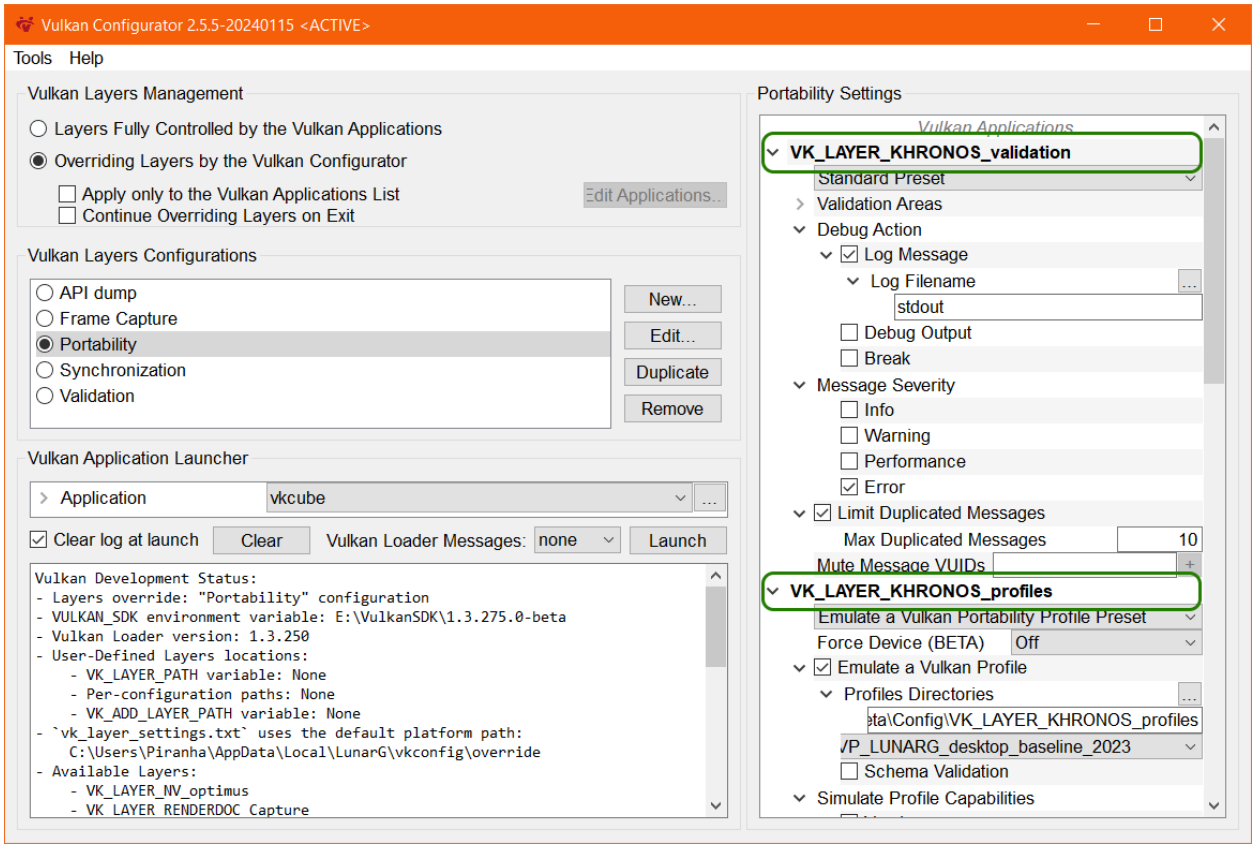
The Vulkan Configurator UI comprises six areas:

1) Vulkan Layers Management: this area controls whether the Vulkan Layers override is active or not. It also determines whether the override is applied only to a selection of Vulkan applications or to all Vulkan applications. Finally, this area specifies whether the override remains active or not when Vulkan Configurator is closed.
2) Vulkan Layers Configurations: the list of pre-configured layers configurations. Vulkan Configurator is installed with a selection of built-in configurations that are listed on the screenshot. Each built-in configuration is designed to handle a specific Vulkan application developer use case. Using the context menu, we can design user-defined layer configurations to create layers configurations for our specific use cases.
3) Create a new layers configuration: edit, duplicate or remove the selected layers configuration. The “Edit…” button allows opening the “*Edit Vulkan Layers*” window to select the layers behavior with the following actions:
    a) to override,
    b) to exclude,
    c) or to be handled by the Vulkan applications.

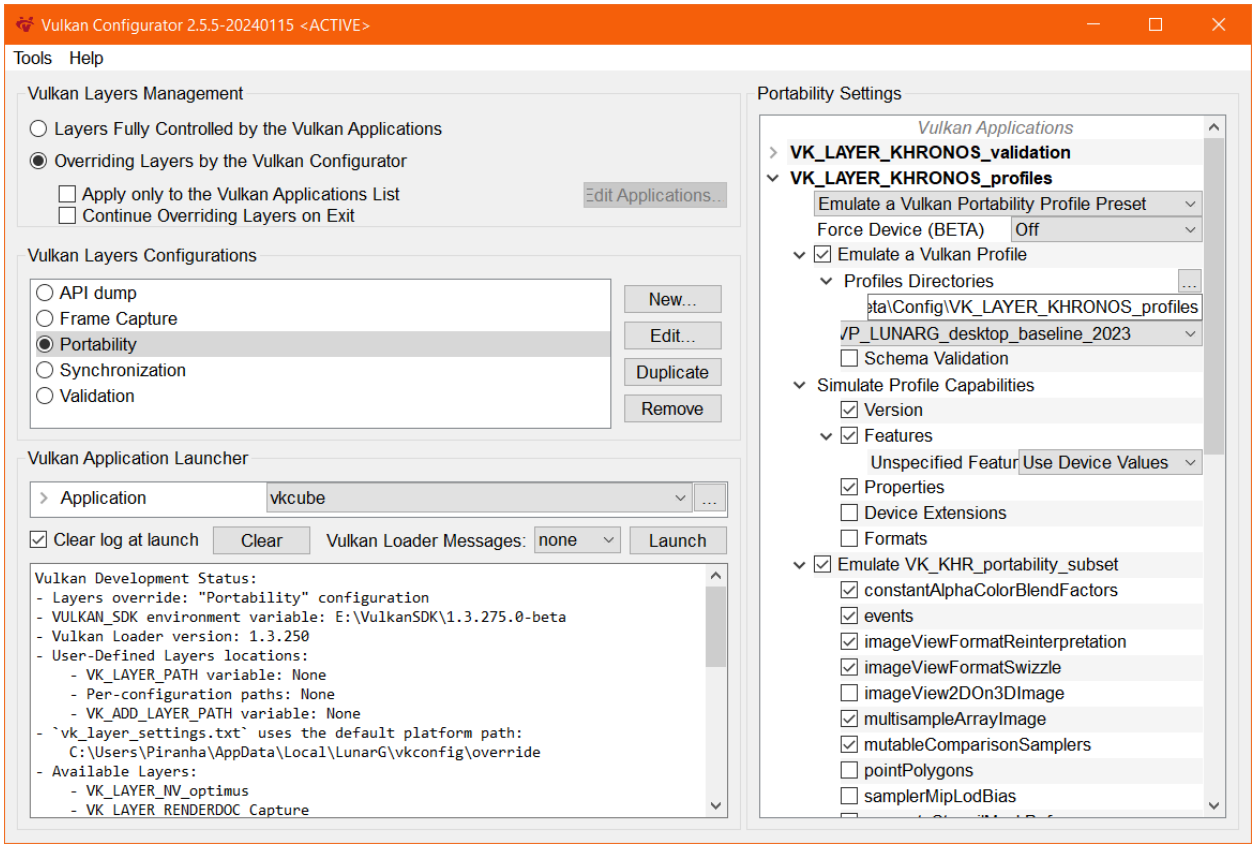
    The “*Edit Vulkan Layers...*” window allows adding paths to find additional layers on the system.

4) Vulkan Application Launcher: this area allows running any Vulkan application with the selected layers configuration.
5) Log window: on start-up, when selecting a layer configuration or updating the layers list of a layer configuration, the log window will display the “Vulkan Development Status” which reports the version of various components, relevant paths for Vulkan developers, and the list of available layers. When launching a Vulkan application from Vulkan Configurator, the log window will display anything sent to stdout or stderr from the Vulkan layers, Vulkan applications, and the Vulkan Loader.
6) Layers configuration settings: the tree of settings for each layer. If the layers have setting presets, they are displayed just below the layer name.

Select the “Portability” built-in configuration from the “Vulkan Layers Configurations” list:

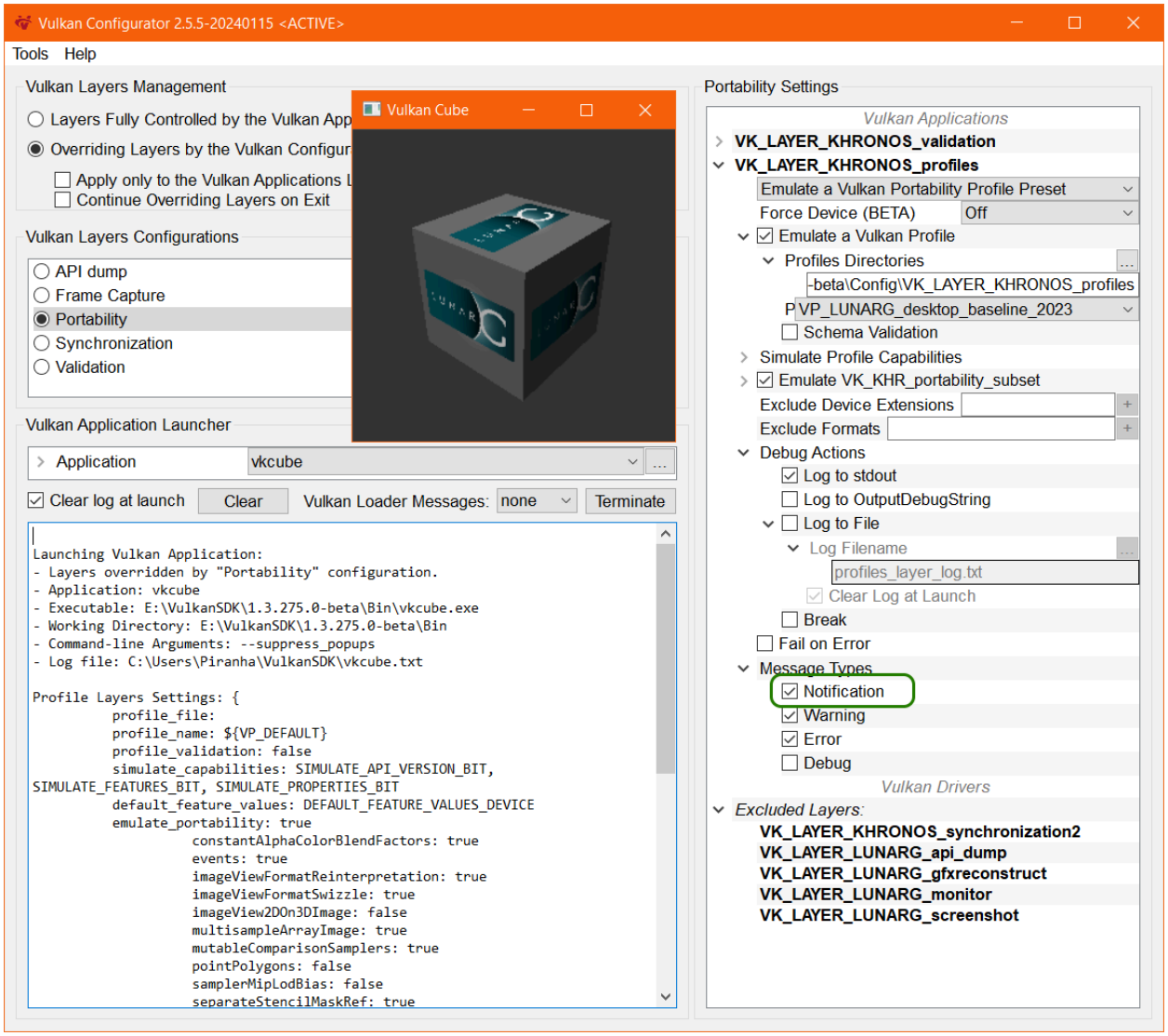


This configuration includes the *Vulkan Validation layer* and the *Vulkan Profiles layer*.
To the right, we can see the layers settings. Among this setting, there is the setting “*Emulate `VK_KHR_portability_subset`*” with each capability listed.



We can hide the *Vulkan Validation layer* settings for now by clicking the caret next to `VK_LAYER_KHRONOS_validation` and observing the *Vulkan Profiles layer* settings.

Selecting the *notification* message type, we get additional logging information to understand how the Vulkan Profiles layer behaves.



Vulkan layers can also be configured using environment variables, and programmatically using the `VK_EXT_layer_settings` extensions. For additional information on layer settings, have a look at the Configuring Vulkan Layers.

Vulkan Profiles layer limitations

The *Vulkan Profiles layer* only changes the reported capabilities of a Vulkan driver. Used together with the Vulkan Validation layer, the solution will report errors when running Vulkan applications that use capabilities that are not reported by the layer.

However, the *Vulkan Profiles layer* is limited in that it doesn't emulate Vulkan capabilities. Hence, we are expecting it to be used on Vulkan developers' machines which we think will

effectively support more capabilities than the majority of the targeted devices in the Vulkan ecosystem by the Vulkan applications.

Furthermore, each operating system implements platform specific extensions. Hence, testing on all required operating systems remains necessary for full Vulkan capabilities requirements verification.

Also, it's not impossible that a Vulkan driver reports a supported capability that is effectively not usable because of a Vulkan driver bug. The *Vulkan Profiles layer* will not help for such cases, but we are hoping for *Vulkan Profile* definitions to help document such scenarios.

Finally, and probably most importantly, the *Vulkan Profiles layer* doesn't help with device performance testing; but we are hoping using the Vulkan Profiles layer will free some development cycles for Vulkan developers to focus on this essential aspect of application development.

Using the *Vulkan Profiles API library*

The Vulkan Profiles API library is a generated helper C++ with a C interface library for Vulkan application developers that provides the following set of APIs:

- APIs to verify instance-level and device-level support for selected Vulkan profiles and/or Vulkan profile blocks.
- Instance and device creation APIs that automatically enable the extensions and features required by Vulkan profiles and/or Vulkan profile blocks.
- Capability introspection APIs to query the extensions, features, properties, formats, and queue families required by a particular Vulkan profile.

We typically expect that it would be trivial for a Vulkan application to integrate the Vulkan Profiles API library to check the support of Vulkan Profiles but harder to create `VkDevice` and `VkInstance` instances with the Vulkan Profiles API library. It is not necessary to use both capabilities.

Checking the support of Vulkan Profiles in the Vulkan application using a Vulkan Profiles API library generated from the Vulkan Profiles that represent the rendering code paths of this Vulkan application, ensure the Vulkan requirements of these rendering code paths remain correctly documented.

This Vulkan Profiles checking also gives a single point of failure in the Vulkan application code to exit cleanly, or choose other rendering code paths. We can expect that the code will be scattered with a lot of `asserts` to check capability requirements, or that the Vulkan application developer will rely on the Validation layer.

A Vulkan sample using the Vulkan Profiles library is available in the Khronos Vulkan Samples repository.

Integration of the *Vulkan Profiles API library* in an application

The Vulkan Profiles API library is provided as a header-only, C++ library (`vulkan/vulkan_profiles.hpp`) that is bundled with the Vulkan SDK. C++ applications thus can simply use the *Vulkan Profiles API library* by including this header-only, C++ library with no Vulkan application build system changes.

The library is primarily designed to be dynamically linked to the Vulkan implementation (loader or ICD). If applications want to dynamically load Vulkan then they have to make sure (one way or another) that the Vulkan API symbols seen by the Vulkan Profiles header-only library are valid and correspond to the dynamically loaded entry points.

In order to enable support for other language bindings, the library is also available in a header + source pair (`vulkan_profiles.h` and `vulkan_profiles.cpp`). In the Vulkan Profiles repository, there is no build configuration for this variant of the library, as it's not meant to be used as a standalone static or dynamic library. Instead, developers can incorporate the files into their own project to build the Vulkan profiles library into it. This may also be useful if the developer would like to optimize compilation times by not having to include the rather large header-only library in multiple source files.

The repository also contains a debug version of the Vulkan Profiles API library which allows logging unsupported capabilities of a Vulkan Profile when checking its support on a system.

The profile definitions are enabled depending on the preprocessor definitions coming from the Vulkan headers; thus the application has to make sure to configure the right set of preprocessor definitions. As an example, the `VP_ANDROID_baseline_2021` profile depends on the `VK_KHR_android_surface` instance extension; thus in order to use this profile, the application must define `VK_USE_PLATFORM_ANDROID_KHR`.

Generating Vulkan Profiles API library

The *Vulkan Profiles API library* is a generated header-only, C++ library. It doesn't support loading dynamically Vulkan Profiles because the solution provides the simplest integration to an engine codebase. In most cases a Vulkan application can't just load a profile, it implements that profile within the codebase. Hence, the *Vulkan Profiles API library* just simplifies the initialization code of that implementation.

The *Vulkan Profiles API library* shipping with the Vulkan SDK serves mostly as an example because it bakes many profiles, most of them being useful for only very restricted scenarios. We expect that most Vulkan developers may be interested in creating their own profiles and generate a copy of the *Vulkan Profiles API library* for their profiles.

The `gen_profiles_solution.py` python script is used to generate the Vulkan Profiles API library and is delivered as part of the Vulkan SDK in the `share/vulkan/registry` directory. Following, a command line example:

```
Unset
python gen_profiles_solution.py
  --registry vk.xml
  --input ./my_profiles/
  --output-library-inc ./my_library/
  --output-library-src ./my_library/
  --debug
```

The `--debug` argument is optional and useful during the development phase for the Vulkan Profiles API library to output useful debug information.

For more information on how to use `gen_profiles_solution.py`, we can use the command:

```
Unset
python gen_profiles_solution.py --help
```

Alternatively, the *Vulkan Profiles API library* can be trivially generated from the Vulkan Profiles repository to include support for any desired Vulkan Profiles. The Vulkan developer just needs to clone the Vulkan Profiles repository and replace the list of profiles files and edit the `${PROFILES_FILES_FOR_API_LIBRARY}` in the CMake file variable before rebuilding the project and grabbing the regenerated `vulkan_profiles.hpp` to check it in the project repository.

Basic usage of the *Vulkan Profiles library*

The typically expected usage of the Vulkan Profiles library is for applications to target a specific profile with their application, and leave it to the Vulkan Profiles library to enable any necessary extensions and features required by that profile.

In order to do so, the application first has to make sure that the Vulkan implementation supports the selected profile as follows:

```
VkBool32 supported = VK_FALSE;
VpProfileProperties profile{
    VP_LUNARG_DESKTOP_BASELINE_2024_NAME, VP_LUNARG_DESKTOP_BASELINE_2024_SPEC_VERSION
};

VkResult result = vpGetInstanceProfileSupport(nullptr, &profile, &supported);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
else if (supported != VK_TRUE) {
        // profile is not supported at the instance level
        ...
}
```

The above code example verifies the instance-level profile requirements of the `VP_LUNARG_desktop_baseline_2024` profile, including required API version and instance extensions.

If the profile is supported by the Vulkan implementation at the instance level, then a Vulkan instance can be created with the instance extensions required by the profile as follows:

```
VkApplicationInfo vkAppInfo{ VK_STRUCTURE_TYPE_APPLICATION_INFO };
```

```
// Set API version to the minimum API version required by the profile
vkAppInfo.apiVersion = VP_LUNARG_DESKTOP_BASELINE_2024_MIN_API_VERSION;
VkInstanceCreateInfo vkCreateInfo{ VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO };
vkCreateInfo.pApplicationInfo = &vkAppInfo;
// set up your own instance creation parameters, except instance extensions
// as those will come from the profile
...

VpInstanceCreateInfo vpCreateInfo{};
createInfo.pCreateInfo = &vkCreateInfo;
createInfo.enabledFullProfileCount = 1;
createInfo.pEnabledFullProfiles = &profile;

VkInstance instance = VK_NULL_HANDLE;
result = vpCreateInstance(&vpCreateInfo, nullptr, &instance);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
```

The above code example will create a Vulkan instance with the API version and instance extensions required by the profile (unless the application overrides any of them, as explained later).

Make sure to set the `apiVersion` in the `VkApplicationInfo` structure at least to the minimum API version required by the profile, as seen above, to ensure the correct Vulkan API version is used.

Once a Vulkan instance is created, the application can check whether individual physical devices support the selected profile as follows:

```
result = vpGetPhysicalDeviceProfileSupport(instance, physicalDevice, &profile, &supported);
if (result != VK_SUCCESS) {
    // something went wrong
    ...
}
else if (supported != VK_TRUE) {
    // profile is not supported at the device level
    ...
}
```

Finally, once a physical device supporting the profile is selected, a Vulkan device can be created with the device extensions and features required by the profile as follows:

```
VkDeviceCreateInfo vkCreateInfo{ VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO };
// set up your own device creation parameters, except device extensions
// and device features as those will come from the profile
...

VpDeviceCreateInfo vpCreateInfo{};
createInfo.pCreateInfo = &vkCreateInfo;
createInfo.pProfile = &profile;

VkDevice device = VK_NULL_HANDLE;
result = vpCreateDevice(physicalDevice, &vpCreateInfo, nullptr, &device);
if (result != VK_SUCCESS) {
```

```
    // something went wrong
    ...
}
```

Advanced usage of the *Vulkan Profiles library*

The Vulkan Profiles library provides many functionalities to extend Vulkan Profiles with additional capabilities or even alter Vulkan Profiles to drop some requirements according to Vulkan developers needs. All these functionalities are described in the Vulkan Profiles library documentation.

For more information on the Vulkan Profiles Library API, have a look at the API reference.

Vulkan Profiles Tools future improvements

- It would be convenient to be able to generate multiple profiles within a single Profile JSON file, for example, moving the content of `VP_LUNARG_desktop_baseline_2022.json`, `VP_LUNARG_desktop_baseline_2023.json` and `VP_LUNARG_desktop_baseline_2024.json` content into a `VP_LUNARG_desktop_baseline.json` file. A single file makes it easier for Vulkan applications and package managers to automatically pick up new profiles.
- It would be nice to be able to emulate with the *Vulkan Profiles Layer* more precisely the capabilities expressed in a Vulkan Profile by selecting capability blocks individually.
- It would be convenient to be able to run the Vulkan Profiles generator scripts from *Vulkan Configurator*, to quickly iterate between updating Vulkan Engine profiles and regenerating the *Vulkan Profiles API library* and Vulkan Profiles documentation.
- It would be interesting to add a new operator to the merge script to generate a profile that lists all the capabilities of a source (engine) profile that are not supported by a destination (platform) profile.
- Add dependencies between features and properties: some capabilities change values depending on whether some features or extensions are supported or not. For example, `pointSizeRange` depends on `largePoints`.
- It would be a good addition to provide a library form of `Vulkaninfo` so that any Vulkan tool could generate Vulkan Profiles files easily, for example, to integrate the Vulkan capabilities of the system into a crash log.

Revision History

Revision Date	SDK Release	Comments
January 2024	SDK 1.3.275.0	- Add a section for user examples of Vulkan Profiles. - Add information about the profiles merge script. - Update Vulkan Profiles API Library section, API changes. - Update Vulkan Layer: VK_KHR_portability_subset settings. - Update future improvements section.
January 2022	SDK 1.3.204.0	- Initial release