Frequently Asked Questions

- Why did Apple deprecate OpenGL® in 2018?

In 2018 Apple decided to deprecate OpenGL, explaining that OpenGL was designed 25 years ago when 3D graphics were in their earliest days, and has kept going thanks to extensions for modern GPU features. However, the fundamentals of 3D rendering have changed, including the overall GPU pipeline, multithreaded operations, and asynchronous processing. Apple released Metal in 2014, seeking GPU/CPU efficiency, predictability, and resource control, particularly for its modern A-series processors. PC developers are moving to Vulkan for similar reasons.

- My system should support Vulkan, but it's not working. What can I do?

The Vulkan Installation Analyzer (VIA) tool is designed to diagnose issues with Vulkan components on a user's system. Download the LunarG SDK to get access to the tool. Running VIA should produce a command-line result hinting at any possible issues. Additionally, VIA generates a dynamic HTML file containing useful information about your system including installed Vulkan drivers, and other information that can be useful to diagnosing Vulkan issues. If after running VIA you still don't understand your problem, submit an issue on the LunarXchange web site and include your VIA HTML output.

- I received a virus alert from Vulkan Loader. Is Vulkan Loader going to give me a virus?

No. Sometimes virus scanning software will flag a false positive. We have seen false positives with obscure virus scanning software and also from others like Microsoft and Norton. Eventually, false positives disappear once those virus scanning programs are updated with latest false positive signatures. See this recent Khronos Blog for more about Vulkan Loader security.

- ## In addition to cross-platform support and being open source, what else sets Vulkan apart from Microsoft DirectX 12?

From an external point of view, the biggest difference has to do with the platforms and the hardware that they support. DirectX 12 was designed for just a few target hardware devices. It runs on only one operating system, whereas Vulkan supports hardware from 10s of companies and supports a much broader range of platforms. There is a bit higher level of abstraction in Vulkan that accommodates that increased affordability. The abstraction is a good thing because it enables affordability, while providing the same performance and usability you get from DirectX 12.

- ## How do I choose between using Vulkan and OpenCL when doing general computing?

If your application is interacting with graphics and displaying your results, then doing it all in Vulkan is probably easier and more suitable. But if it's a more scientific application, then OpenCL is more appropriate. OpenCL has more focus on things like type precision requirements than Vulkan does with its foundation in graphics.

- ## Can OpenGL be developed as a layer on top of Vulkan?

It is certainly feasible. There are rumors of projects for running OpenGL on top of Vulkan. However, issues with accessing all the hardware features exist when writing an OpenGL driver straight to the metal. You can obtain comparable performance because Vulkan is at that level, but you would duplicate all the functionality that has evolved inside the major IHVs over the years.

- ## As a beginner, is it better to learn OpenGL before Vulkan?

If you're really new to 3D graphics, OpenGL or one of the other traditional APIs is probably the best place to start learning. In the future, Vulkan's middleware layer could evolve to potentially provide some interesting learning platforms.

- ## What performance improvements might a well-tuned OpenGL application for games/simulation expect with a switch to Vulkan?

If your application is well-tuned and GPU-bound, you may notice some reduction in CPU power. The CPU may be idle more often, but there won't be a graphics difference if the application is already GPU-bound. If you get many CPU cycles back, you can explore enhancing rendering using some offline CPU techniques outside the rendering pipeline using a combination of CPU and GPU work to get better results.

- ## Are Vulkan shaders written in the same way as in OpenGL or GLSL?

They can be. For Vulkan, you can write in GLSL (with Vulkan extensions) and use an offline toolchain headed by glslangValidator to translate that to SPIR-V (standard Vulkan only accepts SPIR-V). This offline toolchain is included in the LunarG Vulkan SDK. See the SPIR-V toolchain document in the SDK for more information.

There are other ways to write shaders for Vulkan-based applications using different high-level languages and supporting toolchains. Some high-level shaders can be linked directly into an application, so you don't have to rely solely on offline support. We call this off-driver support, and you can do it online by using a side tool in real-time. Note that it isn't part of the core Vulkan drivers stack. SPIR-V can also be written directly in SPIR-V assembly language and assembled using the spirv-as tool included in the LunarG SDK.

Many applications porting to Vulkan also need a way to port their HLSL shaders to SPIR-V. glslangValidator also provides a method to translate HLSL shaders to SPIR-V. Currently, glslangValidator supports HLSL through SM 5 complete enough to run complex, real-world workloads such as Dota 2 and Ashes of the Singularity. It accepts shaders for any shader stage, and handles common language constructs for functions, control flow, variable and type declarations, registers and pack offsets, most DX10 and later texture methods, most intrinsic functions, most preprocessor functionality, most built-in semantics, and attributes that affect stage functionality. To learn more about HLSL support in glslangValidator, visit the FAQ in the Wiki at the glslang GitHub.

HLSL can also be translated to SPIR-V using DXC. DXC supports HLSL starting with SM5. DXC is currently not included in the LunarG SDK. See this link for more information.

- ## Can I mix OpenGL rendering with Vulkan rendering?

Many extensions have been added to Vulkan to allow coordination between OpenGL and Vulkan interaction, including Khronos-wide supported extensions. In addition, vendor specific extensions continue to improve the amount of coordination that can be done. The best place to start investigating the amount of interaction possible is by reviewing the VK_KHR_external_memory*, VK_KHR_external_semaphore*, and VK_KHR_external_fence* groups of extensions."

- ## Does Vulkan provide text support (high-quality LCD) that is lacking in OpenGL?

The initial goal of Vulkan 1.0 is to address the functionality of current GPUs and current drivers. So hardware capabilities are stressed as opposed to middleware. High quality text rendering (anything involving fonts) is out of

scope for this level of activity. Vulkan is ideal for layered implementations and layered APIs that would run on top of Vulkan.

- Is GLSL the standard shading language for Vulkan?

Vulkan does not contain a normative reference to any high-level language specification. The only requirement is the ability to accept shaders in the SPIR-V format. For the convenience of the developers, an extension to GLSL was defined to allow it to generate Vulkan-compatible SPIR-V.

There is no barrier for other methods of generating valid SPIR-V. The SDK includes tools that you can use to validate whether a SPIR-V program is legal for Vulkan and should be accepted by a Vulkan implementation.

- Should we port existing applications from OpenGL to Vulkan, or should we add an OpenGL abstraction layer on top of Vulkan?

There is currently no existing OpenGL abstraction layer that sits on top of Vulkan. For now, the question should simply be “Should I port my existing application to Vulkan?”

It depends if you want to or if you feel you need to. OpenGL will still continue to be supported on most of the platforms Vulkan is available on. However, if your OpenGL application is CPU limited or bandwidth limited, it might benefit from being ported to Vulkan.

- Why use Vulkan? What makes it better then OpenGL?

Vulkan was created as the result for a need for a new generation GPU API. GPUs are increasingly programmable and compute capable, plus platforms are becoming mobile, memory-unified, and multi-core. GPUs will accelerate graphics and compute across diverse platforms. Hence there was a need for flexibility and portability.

As a result, the Vulkan API is Explicit (opens up the high-level driver abstraction to give direct, low-level GPU control), Streamlined (faster performance, lower overhead, less latency), and Portable (cloud, desktop, console, mobile, and embedded).

To gain the benefits that the Vulkan API provides, more work and responsibility is put into the application. Not every developer or application will need to make that investment. For many situations, OpenGL will remain the most effective API. If your application isn’t CPU bound, you don’t put a premium on avoiding hitches, and you won’t do whatever it takes to squeeze out maximum performance, then OpenGL is still a good choice.

- Will there be a SPIR-V layer to allow running DirectX over Vulkan?

That is certainly a technical possibility. The issue is translating DirectX shader code that is post-compiled code into SPIR-V for consumption. The other option is to compile directly from the HLSL if that's how you're generating your binary blobs, which may be typical in the more modern DirectX APIs. Either path will work.

Many applications porting to Vulkan need a way to port their HLSL shaders to SPIR-V. Glslang provides a method to translate HLSL shaders to SPIR-V, which is now available and ready to use. Currently, the HLSL mode of the glslang frontend is complete enough to run complex, real-world workloads such as Dota 2 and Ashes of the Singularity. It accepts shaders for any shader stage, and handles common language constructs for functions, control flow, variable and type declarations, registers and pack offsets, most DX10 and later texture methods, most intrinsic functions, most preprocessor functionality, most built-in semantics, and attributes that affect stage functionality. To learn more about the HLSL to SPIR-V translator, visit this FAQ at the glslang GitHub.

- Does the SPIR-V specification guarantee that precompiled SPIR-V objects will run on all Vulkan targets regardless of the hardware vendor?

That is the goal of the SPIR-V and Vulkan relationship. Khronos has written the most rigorous specification possible for SPIR-V, and any issues should be reported to the SPIR-V project.

The one caveat is that Vulkan does not specify error behavior. If you write a SPIR-V shader that does not follow the SPIR-V specification, different implementations may have different behaviors.

If your SPIR-V passes the SPIR-V validator but behaves differently on different implementations, let Khronos know.

There are feature capability support bits in SPIR-V, so if you have a shader that works on one implementation that supports some capabilities and you run it on an implementation that does not support those capabilities, that's not going to work. But otherwise the intent is that things are portable and that there are less implementation variations in SPIR-V consumers than there has been in GLSL compilers.

- Is Vulkan intended to be used with specific SDKs only?

An SDK is not required. You can find the different elements, which are open source and free from multiple repositories, then build them and compile them together.

Bringing them together in one place, binaries already built, and providing a forum where questions can be asked is the benefit of getting the SDK from vulkan.lunarg.com.

- Do any of the current SDKs include a Vulkan emulator?

Not that we are aware of.

- Does the LunarG SDK have a non-standard way of finding the driver?

The SDK includes the official Khronos loader (libvulkan.so), which was developed by LunarG.

For more information on how drivers are discovered, refer to the Architecture of the Vulkan Loader Interfaces document.

- Does the LunarG SDK support building software without access to the driver or hardware?

Yes, the SDK supports building Vulkan applications without a driver or hardware. The SDK installs the Vulkan header files and the Vulkan loader library in standard system locations.

A Vulkan application compiles using the Vulkan header files. It links using the Vulkan loader library. No Vulkan-capable hardware or drivers are needed for compilation and linking.

- How do the debug and validation tools in the SDK work if you mix Vulkan and OpenGL in an application?

If you mix the use of Vulkan and OpenGL within an application, the debug and validation tools in the LunarG SDK give you information for the Vulkan API only.

- Which Linux Distributions are supported by the LunarG SDK?

The LunarG SDK is tested against the two latest Ubuntu LTS releases and supported on 64-bit Linux systems. LunarG also creates native Ubuntu packages for the two latest Ubuntu LTS releases.

- What is the relationship of LunarG to Khronos?

LunarG is an independent software company sponsored by Valve to help build out the ecosystem for the Vulkan API. LunarG is an associate member of Khronos and participates in multiple Khronos open standard working groups.

- Does the LunarG SDK have installable packages for Ubuntu LTS releases?

Yes, LunarG provides installable packages for Ubuntu LTS releases. LunarG provides both a “Linux Tarball” SDK and native Linux packages for the Ubuntu Linux distribution. The SDK installer for the Linux “tarball” is self-extracting and does not install files to system locations that require administrator privileges. The installer will create a local SDK directory of the form VulkanSDK/<version> in the current working directory. See the Getting Started guide for Linux found on LunarXchange (https://vulkan.lunarg.com/) for more details. The native Linux packages for Ubuntu install the binaries in the system locations. For more information on the two types of Linux SDKs, visit the SDK page on LunarXchange.

- With OpenGL, we had helper APIs like GLUT. Is the LunarG SDK intended to serve as that common front end for OS dependencies?

The LunarG SDK provides the critical and canonical Vulkan API Loader and Validation Layers. In addition, it provides sample code as examples, and other tools such as RenderDoc, vulkaninfo, vkVIA (Vulkan Installation Analyzer), the Layer Factory, Device Simulation layer, Assistant Layer, vkcube, vkconfig (allows a user to specify which layers will be loaded by Vulkan applications at runtime), vktrace/vkreplay, SPIR-V tools, VK_LAYER_LUNARG_monitor (FPS overlay), VK_LAYER_LUNARG_screenshot, and VK_LAYER_LUNARG_api_dump.

For more information about what is in the SDK, you can download it for free from vulkan.lunarg.com. You can also see an overview at here on LunarG.com.

- What are the benefits of using LunarXchange to submit SDK issues?

If you are developing a Vulkan application and are having issues and you are not sure if the issues are in the Vulkan loader, or Vulkan validation layers, or other components within the SDK, you can submit it to LunarXchange.

The engineers at LunarG will help triage the issue and determine if it is a loader or validation layer issue and work with the corresponding public GitHub project as needed to get it resolved.

LunarXchange also allows you to create a private organization. This is useful when a company needs help from LunarG on an SDK component but does not want the issue to be public. The private organization allows for confidential sharing of information between LunarG and the organization. If you are interested in having a private organization on LunarXchange, contact info@lunarg.com.

- What does the Organization function on LunarXchange provide?

If you don’t want your issues to be visible to the public, you submit them privately on LunarXchange within your organization.

If you are interested in creating an organization on LunarXchange, send email to info@lunarg.com.

- How is Vulkan similar to Mantle and what are the differences?

The original proposal for Vulkan was basically a cleaned up version of Mantle. AMD got the conversation going, but over an 18-month period, there were contributions from many other Khronos members. Some are small tweaks and others are major features. For example, render pass was never in Mantle, yet it's a core part of Vulkan now.

Some of the API names are the same, but it is a significant update. There are some very important differences and enhancements, and Vulkan is cross-vendor, cross-platform, so we've landed in a really good place. Overall Vulkan is truly a better API.

- Are CAD applications planning to use Vulkan?

The professional design community is interested in Vulkan. It is not designed just for games; any application that is CPU bound can benefit from Vulkan.

For example, Vulkan can help with a CPU-bound, high-end CAD application model with hundreds of millions of polygons. In the professional authoring industry, there's a lot more legacy code and OpenGL is not going away. But if CAD application developers want to migrate to Vulkan to take advantage of things like the multi-threading for large model interactivity, that is the kind of application that Vulkan is designed to enable.

- Does Microsoft support Vulkan?

Microsoft has not expressed intent to support Vulkan. However, there are well-defined mechanisms by which IHVs can ship a Vulkan driver on any version of Windows. When developing in Vulkan, the application negotiates with the implementation to install an appropriate Vulkan driver for the hardware that is on the machine — similarly to how it works when developing in OpenGL. Vendors can ship drivers for whichever API they wish, and Vulkan drivers are already shipping from multiple hardware vendors for multiple versions of Windows. Vulkan is available on all versions of Windows that are on DirectX 12, so there is potentially some value to the developer community to having a single API, like Vulkan, that spans multiple Windows versions.

- Is Scalable Link Interface (SLI) supported in Vulkan?

Two extensions were introduced after the release of Vulkan 1.0 to allow SLI-style rendering. The extensions are **VK_KHR_device_group** and **VK_KHR_device_group_creation**. At the time of the Vulkan 1.1 release, these extensions were pulled into core Vulkan. More info can be found in this Stack Overflow issue.

- ## Does Apple support Vulkan?

Apple is a member of Khronos and active in several of Khronos APIs, but the company has not publicly stated support for Vulkan. From Khronos' point of view, there is no barrier to any OS platform adopting Vulkan.

In February of 2018, **The Khronos™ Group,** an open consortium of leading hardware and software companies creating advanced acceleration standards, announced that the Vulkan® Working Group's Portability Initiative had collaborated with Khronos members Valve, LunarG, and The Brenwill Workshop to enable Vulkan applications to be ported to Apple platforms. The **Vulkan Portability resource page** links to a collection of the free and open source set of tools, SDKs, and runtime libraries that enable Vulkan development on macOS and deployment on macOS and iOS platforms.

Also available is the open source **LunarG Vulkan SDK for macOS on LunarXchange**, which enables developers to build, run, and debug their Vulkan applications on the Apple Mac platform. The LunarG SDK for macOS provides loader and validation layers, which allows programmers to check their code for correct API usage. LunarG will continue to evolve the macOS SDK to add additional tools and features.

- ## How does Vulkan conformance testing work?

You can find out all the official details about conformance testing on the Khronos website: https://www.khronos.org/conformance

---

- ## Is there any benefit to having multi-thread command submission for Vulkan?

If you have multiple queues, it might make sense to submit work to each queue from different threads. Serialize on a single queue, then have multiple threads submitting to a single queue if it's convenient in your app.

If you just have multiple threads to submit, synchronization is a problem. There's probably no real benefit to doing that.

It's a design intent of Vulkan that submit is supposed to be very cheap. The heavy lifting is done at command buffer construction time, so the expected model for a given render target is to multi-thread the command buffer construction like crazy, but then if you marshal all of those and have a single thread submit them, that's not going to hurt you. It's not going to be the top pull.

Also, If an application remains single threaded, you can still expect CPU performance improvements and the stuttering improvements and so on that Vulkan offers. Multi-thread it is not a requirement to reap the benefits of Vulkan.

[illegible]

- Is a shader debugging tool available for Vulkan available?

There are excellent debugging capabilities provided by tools included in the LunarG SDK, and we look forward to other products from the community. In particular, RenderDoc (included in the LunarG SDK) is the best tool currently available for debugging.

- What is the desktop GPU's hardware support for render passes, considering DirectX does not have this concept?

If you use them wisely, there are certainly things there that desktop GPUs can take advantage of. It's not a mobile-only feature.

- How does the Vulkan Loader detect vendor drivers in a Linux system?

On Linux, the loader detects the drivers via a JSON file that's installed in some system paths. The JSON file indicates where the actual driver library is. There are also some environment variable overrides that you can use to point to a different location rather than a system location.

For more detailed information review the SDK Linux documentation at LunarXchange (vulkan.lunarg.com)

- How much platform-dependent code is needed to write a Vulkan application?

The core Vulkan specification has no platform dependencies whatsoever. We have brought Windows system integration closer to the core in the sense that Windows system integration is now a core extension rather than a separate API. The WSI, for instance, on Windows vs. Linux is mostly the same extension. There would be some difference there, but a lot would be the same calls.

- What anti-aliasing modes are available in Vulkan?

The Vulkan header file defines multi-sampling modes. Multi-sampling is the basic mode. Super-sampling is not formally supported, but per-sample shading which is similar is supported. The Vulkan specification allows implementors to expose directly multi-sampling and super-sampling. We require every implementation to support 4x multi-sampling.

- Will frequently used Vulkan extensions be added into core Vulkan?

The Khronos working group is very interested in your input. If you see an extension that is of intense interest for addition to the core, notify them on the Khronos Github platform.

- ## Does Vulkan support multiple GPUs or multiple GPU acceleration?

The Vulkan 1.1 release made available in March 2018 introduced multi-GPU support allowing use of multiple GPUs in the same system. Check out this writeup on Stack Overflow for more details.

- ## Does Vulkan support 2D graphics?

Vulkan is a very low-level API. If your GPU is capable of it, you can draw with it. At the moment it exposes pretty much the set of primitives that core OpenGL does, so if you're talking about drawing lines and screen-line, co-ads, and things like that, sure. If IHVs have support for real, actual hardware that accelerates 2D better than you could do it with a 3D engine, then they will probably become available as extensions.

- ## How does Vulkan enable performance on mobile GPUs and the desktop?

Vulkan is unique among modern APIs in terms of treating mobile and desktop equally. It is possible to write paths in a Vulkan application which will suffer no penalty on a desktop architecture but will equally exploit all the capabilities which are offered by a mobile architecture. This is primarily through the render paths and the command buffer structure of the API.

- ## Why are there WSI KHR extensions?

Interface problems exist between the OpenGL family APIs and platform APIs due to external delegation of Windows System Integration (WSI). Vulkan brings these things to the surface so the Vulkan loader is visible, explicit, and endorsed by the entire working group.

Solving platform integration is part Vulkan, as opposed to OpenGL which says it's someone else's problem. The intent is to present the best solution based on variations of what's been done with OpenGL. If needed, there are projects on Khronos Github that you can post related issues to.

- ## Does Vulkan use hardware-specific extensions?

Yes, Vulkan supports hardware-specific extensions, which provide support for new hardware functionality specific to one vendor. This feature helps developers quickly and easily take advantage of new hardware functionality.

- ## What is the ideal forum for Vulkan discussion?

There is no official forum. Commonly used forums are the Vulkan subreddit, StackOverflow, and two Khronos initiated forums: 1) the Khronos Vulkan forum, and 2) the new KhronosDevs Slack forum.

- Where should we submit code samples for the LunarG SDK?

The LunarG Vulkan SDK includes samples from the lunarg-github/samples GitHub repository. You can submit samples there.

- How can Vulkan be used to create web applications? Will there be a WebVulkan?

A WebVulkan won't be needed. If you are creating a web application in 3D, use WebGL, especially since Apple will not be supporting Vulkan in the foreseeable future. It's native in the browsers and you don't need a plug in. As WebGL evolves, it's going to very carefully depict the functionality that could be supported on every platform.

- What is the LunarG Vulkan SDK?

The LunarG® Vulkan™ SDK provides you with a variety of resources to aid in creating Vulkan applications. The SDK includes the Vulkan loader and validation layers. Also included are the following utility layers: API Dump, Device Simulation, Assistant Layer, FPS overlay monitor, and Screenshot. Useful tools deliver critical functionality, such as vulkaninfo, trace and replay, SPIR-V™ tools, Vulkan Runtime Installer, Vulkan Installation Analyzer (vkVIA), vkcube, vkconfig, the Layer Factory, plus samples and demos. Refer to the documentation on the LunarXchange website for more information about the SDK and its' contents.

- Where can I find the LunarG Vulkan SDK?

The LunarG Vulkan SDK is available on the LunarXchange (vulkan.lunarg.com) portal.

- What are the benefits of the LunarG Vulkan SDK?

The SDK offers a number of advantages:

- Vulkan, by design, is a very low-level API that provides applications direct control over GPU acceleration with minimized CPU overhead and efficient multi-threaded performance. The SDK provides tools to help you develop to this lower level API.
- The SDK is available for Linux and Windows® operating systems. Because Vulkan is a direct competitor of Microsoft® DirectX12®, Microsoft will not offer Vulkan developer tools for Windows. The SDK

  provides comprehensive tools for both Linux and Windows operating systems.

- The SDK also offers support for macOS and enables developers to build, run, and debug their Vulkan applications on the Apple Mac platform. The LunarG SDK for macOS provides loader and validation layers, which allow programmers to check their code for correct API usage. LunarG will continue to evolve the macOS SDK to add additional tools and features.
- Vulkan is a cross-platform API that supports a range of devices including desktop, mobile, and console. The Vulkan loader and validation layers are canonical pieces that are critical in achieving this cross-platform compatibility. These components are open source, and you can choose to build them. To speed development efforts, the SDK includes the loader and layers already built for you.
- Because Vulkan is a very low-level API, a simple program is much longer to write compared to OpenGL. The included utilities and samples help you learn the process.

- What are Vulkan validation layers and how are they used?

Vulkan requires drivers to omit most traditional error checks; drivers assume that the application is using the driver correctly.

The validation layers are the key for identifying errors in using the API during development. The layers check for correct implementation of the Vulkan API and return any errors found.

You can enable validation layers during application development, and then disable them for production shipments.

- Does the LunarG SDK include code samples for correctly using the Vulkan API?

Yes, examples demonstrating the use of the Vulkan API, with utilities that you can leverage and use, are located in the SDK. In addition, a Vulkan tutorial is included in the LunarG Vulkan SDK.

- What trace tools are available with the LunarG Vulkan SDK?

You can capture a trace file of the Vulkan API calls and replay them using the trace tools. Commonly, application developers share application workloads with third parties, such as IHVs (Independent Hardware Vendors), who don’t have access to their application.

- The trace tool (**vktrace**) captures the Vulkan API activity in an application and stores it in a file.
- The replay tool (**vkreplay**) plays back the trace file, independent of the application.

As an example, this capability is useful for reporting a rendering problem to an IHV. The trace file reduces the need for the IHV to set up an environment to reproduce the problem.

- What are the loader and runtime installer in the LunarG Vulkan SDK?

Loader: Consistent loader behavior across platforms is key to Vulkan. The Khronos-branded loader binary and installation package is included in the LunarG Vulkan SDK. The loader discovers and manages the Vulkan devices/drivers and layers available to the application.

Runtime installer: The runtime installer has been greatly simplified and has been modified so that it can coexist with the INF installers that drivers use. This means that drivers can now install the Vulkan loader without using a runtime installer. In situations where the runtime installer is still useful (such as applications that need to ensure a loader is present), the runtime is now much simpler, which reduces the chances of bugs, and of it being flagged by antiviruses.

- What operating systems are supported by the LunarG Vulkan SDK?

Windows

The LunarG Vulkan SDK is supported on 64-bit Windows systems:

- 64-bit Microsoft Windows 7, 8, 8.1, and 10

Linux

The LunarG SDK is tested against the two latest Ubuntu LTS releases and various other Linux distributions. For the specific distributions refer to the LunarG SDK documentation found on **LunarXchange**.

macOS

The open source LunarG Vulkan SDK for macOS on **LunarXchange** enables developers to build, run, and debug their Vulkan applications on the Apple Mac platform. The LunarG SDK for macOS provides loader and validation layers, which allow programmers to check their code for correct API usage.

- What is SPIR-V™?

The Vulkan shader language is SPIR-V, which is a low-level binary intermediate representation (IR). The Vulkan API requires the SPIR-V format for all shaders.

SPIR-V splits the compiler chain, enabling high-level language front ends to emit programs in a standardized intermediate form to be ingested by Vulkan.

Eliminating the need for a built-in high-level language source compiler significantly reduces GPU driver complexity, enabling a diversity of language front ends.

- Do I need to write shaders directly in SPIR-V?

No. For example, you can use existing shaders with the LunarG SDK included tool glslangValidator, which creates SPIR-V shaders from equivalent GLSL or HLSL shaders. See the question "Are Vulkan shaders written in the same way as in OpenGL or GLSL?" elsewhere in this FAQ.

- What SPIR-V tools are included in the LunarG Vulkan SDK?

Shader Creation Tool

The **glslangValidator** creates SPIR-V shaders from equivalent GLSL and HLSL shaders.

SPIR-V Disassembler and Assembler Tools

Disassembler **spirv-dis** converts SPIR-V shader into human-readable and parsable form to help with debugging. Assembler **spirv-as** takes edited, disassembled output and assembles it.

SPIR-V Validation and Optimization Tools

Validator **spirv-val** reports if a file contains valid SPIR-V. Optimizer **spirv-opt** performs a number of ala-carte transformations on a SPIR-V module. Currently, its transformations fall into two general categories: specialization constants manipulation and code-reduction.

SPIR-V Remapper Tool

The **spirv-remap** tool enhances compression of SPIR-V binary files via entropy reduction, including optional stripping of debugging information and dead functions. As an example, the tool is useful when games have large numbers of shaders and need smaller file sizes.

- What is RenderDoc?

RenderDoc is a graphics debugging tool that's included in the LunarG Vulkan SDK. It allows you to capture a single frame of an application, then load that capture up in an analysis tool to inspect the API use and GPU work in detail. For more information, see the RenderDoc GitHub page.

- What documentation is available for the LunarG Vulkan SDK?

Read about the following documents for the LunarG SDK on the LunarXchange website:

Getting Started Guides
Release Notes
Vulkan Tools Framework
Vulkan Loader Specification and Architecture
Vulkan Validation Layers and Debugging Layers
Trace and Replay Tools
Device Simulation Layer
API Dump and Screenshot Layers
Device Simulation
Assistant Layer
vkconfig
Layer Factory
Vulkan Installation Analyzer (vkVIA)
SPIR-V Toolchain
Vulkan Tutorial
Vulkan Build and Run Samples
Vulkan Specification
Vulkan Man Pages
Various Vulkan Extensions
Various White Papers

- What type of ongoing support does LunarG provide for the SDK?

LunarG provides ongoing maintenance for the loader and validation layers and ongoing development to produce samples and other tools and layers.

For SDK-related support from LunarG, submit issues on the LunarXchange portal.

- What drivers are included in the LunarG Vulkan SDK?

Display drivers should be obtained by visiting the appropriate hardware vendor's website.

- How can I learn more about other Vulkan tools and resources?

Information on tools can be found on LunarXchange, the LunarG VulkanTools GitHub repository, or on the Khronos Vulkan resources page linked from the main Vulkan landing page on the Khronos site.
