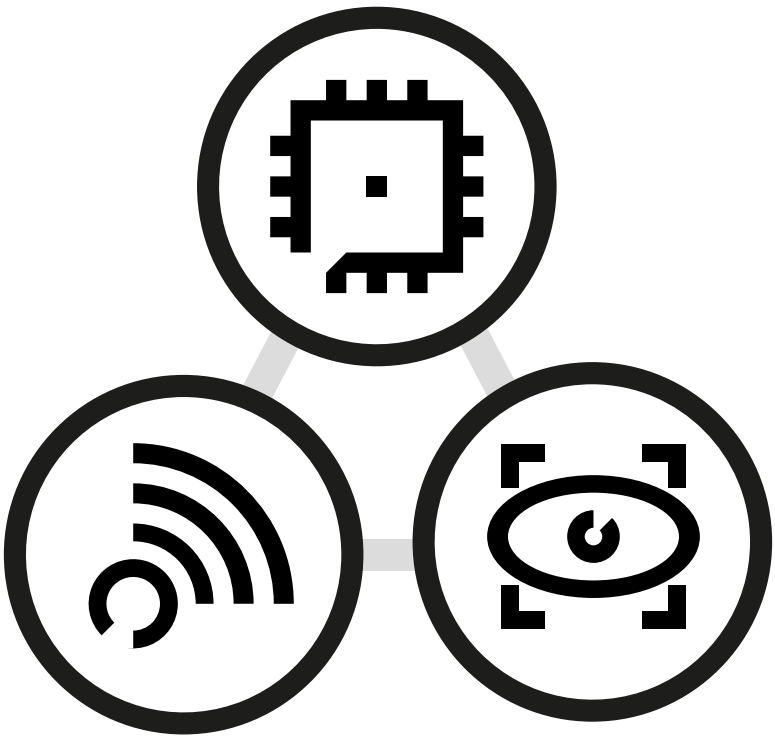
Imagination

Vulkan: Migrating from OpenGL ES

Revision: 1.0
13/03/2020
Public



Published: 13/03/2020-14:35

Contents

1. Introduction to Vulkan: Migrating from OpenGL ES

This document provides a quick introduction to the new graphical rendering API, Vulkan®. It focusses primarily on how Vulkan compares to another open standard API, OpenGL® ES, and the benefits and pitfalls for developers of migrating to Vulkan. In addition, this document will briefly discuss the advantages of Vulkan when compared to other new generation APIs, such as DirectX® 12 or Metal, as well as some of the technical aspects of Vulkan which may be unfamiliar to an OpenGL ES developer.

But first...

What is Vulkan?

Vulkan is a new low-level graphics and compute API that allows developers much greater control over the hardware. It has been designed to take advantage of many of the features of modern devices.

The most important features of Vulkan for developers are:

- better use of multi-core CPUs
- reduced driver overhead
- cross-platform support

Each of these will be discussed in several places throughout this document.

When used correctly, Vulkan should lead to better and more consistent application performance when compared to other high-level APIs such as OpenGL ES. The trade-off is that a significantly greater amount of work is required to initially set up an application in Vulkan. The benefits of this extra work may not be apparent in simple applications, but in more complex ones Vulkan can have a noticeable performance edge over OpenGL ES.

The current API market

Moving to Vulkan is very comparable to transitioning from fixed-function to shader-based pipelines. This was a huge change for developers and there was a lot to learn, but in the end it gave a greater amount of flexibility, control and freedom. Ultimately shader-based pipelines truly revolutionised the industry and it is hard to imagine going back to fixed-function.

The releases of new rendering APIs such as DirectX 12, Metal, and Vulkan have made it clear that all modern APIs are heading in a similar direction, with a focus on low-level control and low overhead, as well as providing both graphics and compute capabilities. Moving to any one of these next generation APIs will genuinely be a move into the future, so the obvious question is:

Why choose Vulkan?

One of the biggest advantages of using Vulkan is that it supports a myriad of platforms. This puts it ahead of its competitors. There is only one platform that Vulkan does not

support (at least not directly) which is macOS. However, there are wrappers available, such as MetalVK, that work on top of the supported low-level API. This multi-platform aspect of Vulkan enables developers to save a lot of time and money on learning and supporting all the new APIs.

Another advantage of Vulkan over the others is that it enjoys quite wide early adoption by game engine developers. The biggest game engines used by most developers today already have full Vulkan support. These include Unity, Unreal® Engine 4, CryEngine, and more. This means a developer can take full advantage of Vulkan without the hassle having to create an engine from scratch.

Despite its apparent complexity Vulkan can actually be quite a bit easier to learn than some of its competitors. This is particularly true for developers who have experience with older APIs such as OpenGL ES. Vulkan has the same graphics pipeline, so it is only a matter of learning the new interface (API) to control it. This also means that porting techniques from older APIs can be easier as well.

2. Comparing OpenGL ES and Vulkan

When considering a big transition such as changing rendering API, it is important to understand how this change will affect the development process.

This section will focus on how developing with Vulkan compares to developing with OpenGL ES. The core takeaway of this comparison is how much more control a developer has over the GPU when developing with Vulkan. This has both advantages and disadvantages, for instance, an application can be optimised much more effectively because a developer can specify more precisely what they want the GPU to do, however the Vulkan driver does a lot less hand holding. This means Vulkan applications generally take longer to initially write and it is easier for things to go wrong.

One of the main points of difference between OpenGL ES and Vulkan is:

Vulkan Gives Explicit Control Over the Application

When using OpenGL ES, the graphics driver can often seem like a black box to developers - everything is hidden and abstracted away. This can be useful for new graphics developers who do not want to overwhelmed straight away, as they can simply instruct the API to do a specific task and the graphics driver will decide how to do it. However, for more experienced developers who are looking to get the best performance out of their application this disconnect from the hardware and lack of control limits how far optimisation can ultimately go.

Vulkan was designed from the beginning to approach this very differently, giving as much control over the hardware to the developer as possible. The driver layer is significantly slimmer and everything is exposed. The developer tells the driver exactly what they want to happen and the driver has to do minimal guesswork.

Here are three key examples of how this design philosophy is implemented in Vulkan and how this differs in OpenGL ES.

API Objects/States

In OpenGL ES, there is a single global render state that can be modified by the developer. Based on this state the driver has to determine on-the-fly what commands need to be sent to the GPU. This approach is very error prone, as the developer has to ensure, when issuing draw calls, that the render state is correct and exactly how they want it to be.

In Vulkan, the new way of doing this is to prepare beforehand all the API objects which describe what actions to take. This way the graphic driver does not have to decide anything and can simply translate the actions into machine code and execute it on the GPU. This approach is much less error prone as the API objects usually do not change during runtime. The only thing a developer has to make sure that the right API object is bound and that this object has the correct content.

Synchronisation

Synchronisation in software engineering can be a challenging problem. That is why in OpenGL ES it is mostly hidden from the developer and only coarse-grained synchronisation is possible in some places.

In some cases the driver creates synchronisation point between the CPU and the GPU, where one would wait for the other to finish a particular operation. Unfortunately, the application developer has no idea these points exist, meaning performance could be poor for no obvious reason.

In Vulkan, synchronisation is explicitly specified in a fine-grained manner.

This is supported by fences, semaphores, events and barriers:

- Fences are used to signal to the CPU when a particular GPU operation has been completed. A very common example would be when an application needs to wait for a specific set of commands to finish before it can continue. A function called `vkWaitForFences()` is used to wait for any number of fences to signal.
- Semaphores can be used to marshal ownership of shared data. They are used to signal between GPU operations, as Vulkan offers very few guarantees about the execution order of GPU commands.
- Events provide fine-grained synchronisation primitives that can be signalled and waited upon by specific operations within a set of commands during execution.
- Barriers provide execution and memory synchronisation between sets of commands.

This does make development more difficult, but it also enables the developer to do things that were previously impossible due to the black box driver model. It is now possible to do precise and efficient synchronisation, which again could help application performance.

Image Layout Transitions

Image layouts specify how the driver should organise pixels in memory. Due to the way GPUs work, storing pixels in a linear fashion does not give an optimal performance, and instead pixels are stored in a zig-zag pattern.

In OpenGL ES, the driver determines if an image layout transition is needed and always tried to keep images (textures) in the best possible layout for operations like sampling. This helps to improve performance but unfortunately, sometimes this resulted in unnecessary transitions.

In Vulkan, the developer can explicitly specify when and what kind of image layout transitions will occur. It is still the driver's task to perform the transitions though. This means in a well-designed Vulkan application images will always be the most performant layout but unnecessary transitions are avoided.

The rest of this section will go over a few more of the important differences between the design of OpenGL ES and Vulkan.

Key Differences Between OpenGL ES and Vulkan

User-managed memory allocation

In OpenGL ES, a developer can only specify the amount of memory required and certain hints on what it would be used for.

In Vulkan, the developer has the ability to request large memory blocks from the driver and then sub-allocate it as required. This allows for finer-grained control over memory allocation and better resource lifetime tracking, but means the developer must take responsibility for freeing any allocated memory.

Vulkan memory is split up into two categories: host memory and device memory.

Host memory is the memory needed by the application for non-device-visible storage. Vulkan gives the opportunity of using a custom host allocator when allocating in this memory. The allocator is optional and if it is not used Vulkan will take care of allocations.

Device memory is the memory visible to the device. This is where some opaque images and buffers reside. In Vulkan, the number of device allocations is driver-limited. The correct way of allocating buffers and images is to use a single allocation for several images or buffers. Vulkan also provides resource pools to allocate resources such as command buffers and descriptor sets. The actual content is indirectly written by the driver.

Explicit data transfers

The performance of applications which are content heavy sometimes suffers from a phenomenon known as ghosting. Ghosting happens when a resource is used in multiple frames by the driver and the hardware, but it is being continuously updated by the application. As the resource is still in use by the hardware, the driver has to create 'ghost' copies of the resources which are invisible to the developer, so it can be updated.

In Vulkan, developers have to explicitly manage allocated memory and synchronise memory accesses. This approach completely eliminates ghosting, but developers need to implement a method, such as a ring buffer with sufficient synchronisation, to allow for modifying objects that are being used by the hardware.

Full control over resource lifespan

In OpenGL ES, a developer is only able to mark resources to be deleted, but the driver decides when to actually free the memory as the resource might still be in use by the GPU. This can result in spikes in frame times and unpredictable performance.

In Vulkan, the opposite is true the developer has full control over when resources are freed. This results in predictable performance and fewer frame time spikes but can lead to errors if the application tries to release resources that are still in use.

Scaling to multiple cores

The single threaded nature of older APIs like OpenGL ES has become more and more of a bottleneck over the years. There were many attempts to utilise multiple cores, such as driver-side multi-core work consumption, or deferred contexts in D3D11. However, these could not bring enough of a performance benefit for the amount of effort required to implement them.

Vulkan is designed to take advantage of multi-core architectures, providing many features to aid distributing work over multiple cores. The API splits the draw call submission into two parts: command buffer generation and command buffer submission. This allows the generation of individual command buffers to be distributed across multiple cores. After this is done a single core can submit all the command buffers generated. Command buffer submission is relatively cheap, so the serial part of this workload is minimised.

For further information see this blog post on the topic: *Vulkan: Scaling to multiple threads*

Precompiling shaders

In OpenGL ES, shader compilation is an expensive operation that has to be done on the target device, as each vendor uses their own binary format. Although shader caching can be utilised, this has helped less and less, as applications the numbers of shader variations that need to be compiled have increased. The driver usually compiles shaders just before they are needed, leading to spikes in frame time.

During the development of Vulkan a new vendor independent intermediate format called SPIR-V was proposed. This format allows developers to precompile the shaders into an optimised driver friendly format. Only register allocation, validation, shader patching and binary translation needs to be done on the target device.

Shader caching has also changed, as now entire pipeline states can be cached. This somewhat increases the shader variations needing to be cached, but it helps the graphic driver a lot to know the entire state. The developer has full control over when a shader is compiled and potentially cached.

Error handling

One of the many things that increases driver overhead with OpenGL ES is that it needs to validate the global state for errors all the time. This is because an application can query the error state after each and every API call, resulting in a lot of unnecessary overhead in release environments.

To alleviate the driver overhead, Vulkan removed the runtime error-checking mechanism, and introduced a validation layer-based system. These layers are not part of the core API, so can be switched on and off between debug and release builds. This way developers can still validate errors effectively, but release builds no longer have this overhead.

Pixel local storage support

Tile-based architectures have the ability to accelerate effects with fast Pixel Local Storage (PLS) memory. Utilising this memory has the benefit of significantly reduced

bandwidth usage and battery life savings. OpenGL ES did have support for PLS, but it was through an extension, `shader_pixel_local_storage(2)`.

In Vulkan, PLS memory (and therefore tile-based architecture) support is built-in through use of render passes and subpasses. These two features allow the developer to specify dependencies between effects so that intermediate data can be kept in PLS memory, rather than being written out to system memory.

3. Should You Switch to Vulkan?

In many cases, moving to Vulkan is very beneficial for a developer, however it is important to evaluate whether there are the resources available to afford the move.

Here are a few examples of different situations where moving to Vulkan will have a significant impact on performance and some where it might not have much effect.

Situations where Vulkan can help

A CPU-bound application with parallelisable graphics work submission

Vulkan particularly excels at making heavy API usage, CPU-bound applications faster, through its low driver overhead. It can also help if the developer needs to parallelise the graphics work for submission, as Vulkan has many features to aid in distributing work across multiple cores.

Few platforms targeted and max performance is required

It is quite difficult to create an optimised build for each and every platform, meaning it would take quite a lot of effort to fully utilise Vulkan's capabilities. However, when only a handful of platforms are being targeted, a lot more effort can go into optimisation. In these situations, it can be very worthwhile to switch to Vulkan in order to get the most out of the hardware. It is possible to reduce frame rate spikes, hitches, and jitters by using Vulkan to take full control over synchronisation in the application. It is still up to the developer to take the opportunity, but it is certainly there.

Situations where Vulkan can help, but requires more effort

An application needs to support pre-Vulkan platforms

When a developer needs to support pre-Vulkan platforms, moving to Vulkan will increase the amount of code that needs to be supported. This should be considered when thinking about the resources required to switch to Vulkan.

A heavily GPU-bound application

When an application is heavily GPU-bound, removing driver overhead might not help that much. However, it will certainly cut down on CPU usage, reducing power consumption and heat output. This may allow the GPU to run on higher frequencies.

A heavily CPU-bound application as a result of non-API work

In this case, Vulkan might not provide many benefits. However, it may help a small amount, as there would be more time for the CPU to do everything else. It may be a good idea to optimise the non-API workload first though.

A single-threaded application that is unlikely to be changed to use multiple threads

For an application like this switching will not help that much.

Situations where Vulkan cannot help

An application performance is already fine

It is a common situation nowadays on mobile that the application simply does not saturate resources at all – for instance, simple 2D games. In these cases it will not matter that much if the developer ports their application to Vulkan. Obviously it can help eliminate more of the CPU load, but in the case of an already well-running application there is little benefit.

An application which is already written, and is GPU-bound and heavily optimised

In the case of applications that are already heavily optimised even with last-generation APIs (for example if they follow the Approaching-Zero-Driver-Overhead (AZDO) principles) Vulkan probably will not help that much at all.

4. Porting Applications from OpenGL ES to Vulkan

When moving from OpenGL ES to Vulkan, it is important to consider the potential issues that can arise with the adoption of a new, more complex API.

Vulkan is much more verbose than OpenGL ES that makes it quite complex and harder to maintain. This verbosity ultimately improves performance but comes at a price, as Vulkan requires much more maintenance and a higher level of responsibility to make sure that everything is correct.

OpenGL ES is built in such a way that hides a lot of the operations required to run the application correctly. Vulkan is more hands-off and requires code to explicitly define and execute these operations. As mentioned in *Comparing OpenGL ES and Vulkan*, operations such as resource management, synchronisation, error checking and validation, and shader compilation are taken care of automatically by OpenGL ES, while Vulkan requires this all to be handled manually.

As general advice, where possible, do **not** port directly from OpenGL ES to Vulkan. Instead, completely rewrite the application in Vulkan and port back to OpenGL ES. This ensures the best possible performance gain from using Vulkan.

Wrapping Vulkan and PowerVR Framework

Since Vulkan is such a verbose API, there can be quite a steep learning curve for new developers. It can take some time to get used to it which can initially results in errors.

The best way to avoid these inconvenient issues is by wrapping the Vulkan part of the application in a custom framework. This cuts down on repetitive code and helps with avoiding unnecessary bugs and errors. Performance may take a hit on wrapping Vulkan, but that is an implementation-specific price that may be worth paying to speed up development.

The PowerVR Framework provides this functionality with minimal overhead. It makes sure that all the verbose and error-prone parts of Vulkan are taken care of automatically. The developer is then free to focus on the application-side while still maintaining all of the benefits of using the Vulkan API.

For more information on the PowerVR Framework, feel free to read the *PowerVR Framework Development Guide*.

5. Brief Overview of the Vulkan API

In *Comparing OpenGL ES and Vulkan*, the general differences in design philosophy between OpenGL ES and Vulkan were compared. This section will go into a bit more detail about particular aspects of the Vulkan API which may be unfamiliar to an OpenGL ES developer. This includes discussion of object initialisation, managing framebuffers, the presentation engine, and the pipeline.

An important concept to understand before moving on to anything else is the Vulkan loader.

What is the Loader?

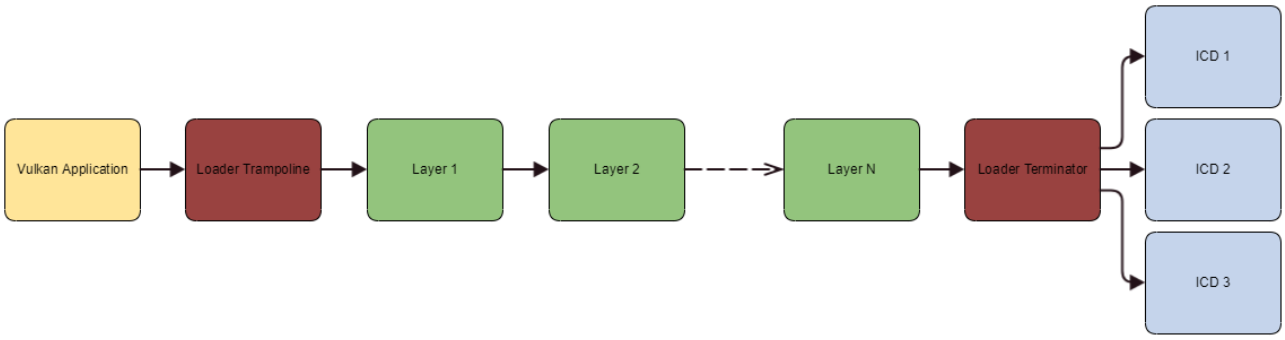
The Vulkan loader is used to connect an application with Vulkan-supporting hardware. This is usually one or more GPUs in the user's system. The loader sits between the application and one or more Installable Client Drivers (ICDs). An ICD is the software which interfaces with the hardware itself. The loader can also introduce special, optional functionality by using any number of Vulkan layers. Layers will be discussed in more detail in *Layers and Extensions*. When a Vulkan function is called in an application the loader determines how to dispatch it to the appropriate set of layers and ICDs.

Acquiring the Loader

There are two types of loader available: a desktop loader and a mobile loader. The desktop loader is currently targeted at Windows and Linux while the mobile loader is only available for Android and is also closed source. Either of these loaders can be acquired by installing a Vulkan application. This is can be through the OS, in the case of Android, but it can also be in a driver package or included in an SDK, such as *LunarG Vulkan SDK*.

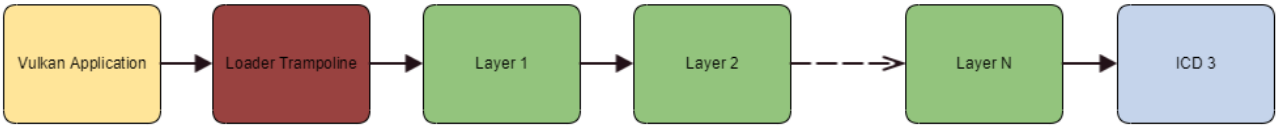
Call Chains and Dispatch Tables

The Vulkan loader uses the concepts of call chains and dispatch tables to control the dispatching of function calls. The call chain is the sequence of function calls from the application to its final destination at the IDCs. Call-chains, in most cases, are started by a trampoline. A trampoline is an entry point for a command that triggers the proper call-chain start. As mentioned above, the loader also introduces a series of layers which can add functionality beyond the core Vulkan specification.



The application initially calls a Vulkan function in the trampoline. The trampoline sets up the chain and then calls the function in the first layer. This continues along the chain with each layer calling the function to the next one until it reaches the loader terminator. This code then distributes calls to multiple ICDs..

There are two types of call chains, an instance call chain and a device call chain. An instance call chain (illustrated above) has a loader terminator because it dispatches calls to multiple ICDs, whereas a device call chain (below) is for function which affect a specific device so there is no need for a terminator to distribute the calls.



A dispatch table is used at each point in the chain to track which function needs to be called in the next layer. Each layer has an individual dispatch table which is essentially just a collection of function pointers. Each layer creates a dispatch table by calling `vkGetInstanceProcAddr` or `vkGetDeviceProcAddr` in the next layer to retrieve a list of function pointers. If a layer does not implement any functionality for a specific function then it can simply call `vkGetInstanceProcAddr`/`vkGetDeviceProcAddr` in the next layer and then pass that back. This means certain layers can be avoided in the chain.

To get the best performance, the trampoline can be scrapped altogether by using `vkGetDeviceProcAddr` in the application. This means the developer must create their own dispatch table, making the application run faster.

Layers and Extensions

Layers

Vulkan requires a developer to be very explicit in everything they do. This is an intentional design of the API and reduces the driver overhead but is more prone to validity errors as Vulkan does very little error checking of its own.

There are the two main types of errors in Vulkan:

- Validity errors - these stem from the incorrect usage of the API. This is generally due to the application not following the API rules described in the Vulkan specification document. Normally, the incorrect usage of the API will result in an undefined behaviour which can make it difficult to determine the initial error.
- Runtime errors - these occur whether the API is used correctly or not. They are errors that take the form of return codes returned from function calls. These return codes point toward a specific issue, for instance running out of memory on allocation of an object.

Despite not catching these errors itself, Vulkan does provide help with application debugging through use of its layer framework. Enabled layers are able to intercept API entry points, evaluating or modifying the called functions before passing them on to the ICDs. This is mentioned in *Brief Overview of the Vulkan API*. In addition, multiple layers can be chained sequentially to enhance their functionality. These layers are optional components and therefore can be disabled in the final version of the application, reducing the driver overhead.

A particular type of layer, called a validation layers, is able to detect and log validity errors. This is really useful when trying to catch incorrect usage of the API.

An example of these validity errors can be seen when creating and destroying an object in Vulkan. The validation layer will track the object and monitor its lifecycle. If at any time the object is destroyed incorrectly, the validation layer will log the error in the standard output.

Setting up the validation layers is one of the most important first steps when setting a Vulkan application. The next step is defining any required extensions to run the application.

Extensions

Extensions in Vulkan work similarly to the ones in OpenGL ES. They extend the API's functionality and may add additional features or commands. They can be used for a variety of purposes, such as providing compatibility for specific hardware.

Extensions can be divided into two types:

- **Instance-level extensions** are extensions with global functionality. They affect both the instance-level and device-level commands.
- **Device-level extensions** specifically affect the device they are bound to.

Vulkan does not make assumptions about the type of application being developed, as it could just as easily be a compute application as a graphics one. For this reason, some fairly key functionality for graphics applications such as surfaces and swapchains are both considered extensions to the core API. The surface extension (`VK_KHR_SURFACE_EXTENSION_NAME`) is an instance-level extension, while the swapchain extension (`VK_KHR_SWAPCHAIN_EXTENSION_NAME`) is a device-level one.

Initialising Objects

The next step after initialising the layers and extensions is to initialise the application. Vulkan provides a simple, albeit slightly verbose way of initialising objects.

The initialisation of objects is divided into two parts:

- defining an info struct
- using the info struct to create an object of the desired type

An info struct is essentially a collection of information and parameters that will determine how the object is created. Vulkan requires that all of the information in the structs be defined in advance.

Creating an instance

A quick example of creating a Vulkan instance is shown below.

```
VkInstanceCreateInfo instanceInfo = {};
instanceInfo.pNext = nullptr;
instanceInfo.flags = 0;
instanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instanceInfo.pApplicationInfo = &applicationInfo;

instanceInfo.enabledLayerCount = static_cast<uint32_t>(appManager.instanceLayerNames.size());
instanceInfo.ppEnabledLayerNames = appManager.instanceLayerNames.data();
instanceInfo.enabledExtensionCount
 = static_cast<uint32_t>(appManager.instanceExtensionNames.size());
instanceInfo.ppEnabledExtensionNames = appManager.instanceExtensionNames.data();
```

The specific variables of different info structs vary depending on the object being initialised, as the information required is different for each object. The variables, `sType`, `flags`, and `pNext` are common to all structs.

- `sType` is the type of info struct
- `flags` are used to pass additional information. This is usually set to zero.
- `pNext` can be used to point to another struct containing additional information. This is often used in cases where an extension requires more information when setting up the object. This means the original info struct does not have to be modified by the extension.

This info struct is then used to create the object by calling the function:

```
vkCreateInstance(&instanceInfo, nullptr, &appManager.instance)
```

When this function is called, a pointer to the info struct is passed along with a pointer to variable where the newly-created instance will be returned (`appManager.instance`).

After creating an instance, the next steps in setting up a Vulkan application are:

- querying the available physical devices (the GPU) to find the one required for the application
- defining a surface to present the result of the rendering operation

- creating a swapchain to manage how rendered images are presented to the surface

Defining a surface is platform/OS dependent and therefore requires the correct extension to be loaded.

Devices and Queues

Physical devices

After creating an instance, the application needs to select a physical device. A physical device represents a piece of hardware installed in the system, most commonly a single GPU. The application can retrieve a list of the available physical devices and select the one which has the properties and features which most closely matches the application's requirements. These properties and features can be simply things like whether the device is integrated or discrete, or something more complex like the maximum size of the pool of push constant memory.

Queues and queue families

Once a physical device has been selected, the device needs to be queried for available queues. In Vulkan, all commands need to be executed on queues. Each device makes a set of queues available which can execute certain operations such as compute, rendering, presenting, and so on.

Queues that share certain properties, for instance those used to execute the same type of operation, are grouped together into queue families. In order to use queues, a queue family which supports the desired operations of the application needs to be selected. For example, if the application needs to render anything, a queue family which supports rendering operations should be selected.

Logical devices

After the physical device and queue family have been selected, they can be used to generate a logical device handle. A logical device is the main interface between the GPU (physical device) and the application and is required for the creation of most objects. It can be used to create the queues that will be used to render and present images. A logical device is often referenced when calling the creation function of many objects.

Swapchains

In OpenGL ES, the process of rendering onscreen starts with acquiring the device context associated with the window. This is then followed by defining how to present the image on the screen, including finding out the format of the window and what capabilities it supports. Finally, a rendering context is created and activated.

In Vulkan, the process is slightly different. After creating an instance and selecting a physical and logical device, a swapchain must be created. A swapchain contains a collection of framebuffers, called images, that can be thought of as the individual

frames of the application. The application retrieves these images, renders to them, and then returns them to their place in the swapchain.

Creating a swapchain

Like other Vulkan objects, a swapchain need to be setup in advance, with its features specified in an info struct. These include properties like the size of its images, the format of its images, the presentation mode, and so on. The presentation mode of the swapchain is very important as it determines the order in which the images will be presented to the surface.

The swapchain cannot be modified on-the-fly, so if, for example, the surface size is changed during runtime, the swapchain needs to be destroyed and recreated.

It is important to note again that a Vulkan application may not even need to render anything at all, so would not require a surface or any images. It is for this reason a swapchain functionality is found in an extension (`VK_KHR_swapchain`).

Render passes and Framebuffers

Render passes

A render pass describes the set of data necessary to accomplish a rendering operation. In Vulkan, this is a set of framebuffer attachments that will be used during rendering. These attachments include any buffers that will read from or written into during rendering, such as colour, depth, and stencil buffers. This can also include input attachments which are intermediate buffers which are written into in one sub-pass and then read out of by another one.

Following the standard Vulkan paradigm, these attachments have to be explicitly defined when creating a render pass, with information like image format, number of samples, and load and store behaviour specified. This reduces the driver workload during runtime, as it does not have to deduce this information itself.

In addition to attachments, render passes also contain one or more sub-passes that order the rendering operations. Sub-passes essentially represent a phase of rendering in which rendering work is done with a sub-set of the attachments in the render pass. A set of commands are recorded into each sub-pass to describe what work needs to be done in that sub-pass.

The render pass also defines a set of sub-pass dependencies which determine the order of execution for pairs of sub-passes. They act as execution and memory dependencies. Dependencies are vital when two or more sub-passes access the same attachment, as Vulkan does not guarantee the order in which the sub-passes will be executed by the GPU.

It is important to note that while a render pass describes the characteristics of all of the attachments used and what to do with them, it does not point to any actual objects. This is handled by framebuffer objects.

Framebuffers

In OpenGL ES, framebuffers are strictly defined by their use. They are a set of textures or attachments that are rendered on, and later presented on screen.

In Vulkan, all the attachments used by a render pass are defined in framebuffers. Each framebuffer defines the attachments related to it. This can encompass the textures which includes the colour and depth/stencil attachments, and input attachments.

Framebuffers are fairly simple objects. Each new framebuffer contains a reference to a specific render pass, a set of pointers to the attachments (images), and some information on the dimensions of these attachments.

Splitting the description and definition of attachments between the render pass and the framebuffer respectively helps the overall optimisation of operations on the GPU and potentially allows the framebuffer object to swapped without changing the render pass, as long as the framebuffer is compatible with the render pass.

Shaders

In Vulkan, shaders use a new intermediate, byte-code format called SPIR-V. SPIR-V can be used for both graphical and compute operations, so is also compatible with OpenCL.

Shaders can still be written in GLSL, but then must be converted to SPIR-V using an offline compiler. To help with compile code from GLSL to SPIR-V, Khronos released a vendor agnostic compiler called *glslang*. *glslang* also automatically error checks and verifies that the shader is standard-compliant.

In Vulkan applications, the shader code is loaded into a thin wrapper called a shader module. Shader modules hold a pointer to the source code in memory and also reference the stage of the pipeline at which the shader will be used, for example the vertex shader or fragment shader stage. Shader modules are referenced when a pipeline is created.

The main advantages of using the SPIR-V format are:

- Less flexibility to vendor compilers on how to interpret the code
- Streamlined portability of shaders across different vendor compilers that have to turn the shader into native code

Descriptors and Descriptor Sets in Vulkan

Descriptors, as the name implies, are used to describe resources, such as buffers and images, which are going to be passed to shaders. They hold information that helps with binding data to shaders, and additionally describe any information Vulkan needs to know before executing the shader. Descriptors are not passed individually and are actually opaque to the application, but instead are bundled together in objects, known as descriptor sets.

The process of creating a descriptor set is a three-step process:

1. Create the descriptor pool that the descriptor sets are allocated out of. This is similar to command buffers being allocated out of command pools and helps spread the cost of resource creation.
2. Create a descriptor layout that includes information on the binding points (where in the pipeline the resource is needed) and the type of data to be passed to the shader.
3. The descriptor sets themselves are actually initialised in two steps. First, the descriptor set is allocated out of the previously created descriptor pool and then these descriptor sets are updated to hold a pointer to the data that is going to be passed to the shader, for instance textures, uniform buffers and so on.

When creating a pipeline, a pipeline layout is created which describes all of the resources required by the pipeline. These resources include descriptor sets (via descriptor set layouts) and push constants.

The Vulkan Pipeline

A pipeline is best described as a collection of stages in a rendering or compute process. Each stage processes the data it receives from the previous stage and then passes on its output to the next stage.

In OpenGL ES, some pipeline stages such as vertex and fragment shaders are programmable, so their functionality is entirely under the control of the application, while other stages such as the rasterizer and blending are fixed function so are immutable.

The states and parameters for each stage can be changed during runtime. This means that the pipeline can be set up to render an object, and then the programmable stages can be updated to render another object.

This is different in Vulkan as the pipeline is stored in one large, immutable object that needs to be prepared during initialisation and cannot be changed during runtime. If the pipeline does need to be altered, the object has to be destroyed and then re-created. Each object that needs to be rendered will therefore potentially use a different pipeline. The advantage of this approach is that the driver no longer has to check for the validity of the pipeline object during runtime, drastically reducing overhead.

Note: In addition to a graphics pipeline which performs rendering operations, Vulkan also offers a separate compute pipeline. The compute pipeline allows the application to perform more general computational work such as physics calculations.

Otherwise, the pipeline structure in Vulkan is virtually identical to the one in OpenGL ES, with a set of fixed function and programmable stages. The fixed function stages are somewhat configurable in Vulkan, but their general functionality is still immutable.

Creating a pipeline

When creating a pipeline object, each of the individual need to setup and configured individually before the entire pipeline is created with a single command. For the fixed function stages this includes configuring the vertex input, rasterization,

colour blending, and multisampling stages. For the programmable stages this includes specifying which shader modules are going to be used and selecting a set of descriptor sets which link the resources that are going to be used by those shaders. Creating these shader modules was covered in *Shaders* and initialising the descriptor sets was in *Descriptors and Descriptor Sets*.

A note on the viewport

While most of pipeline cannot be modified after its creation, certain elements can be set as dynamic state, meaning they can be updated at a later point.

This is the case for the viewport which can be set as dynamic state during pipeline creation and then updated with the command buffer command `vkCmdSetViewport`.

An important point to remember is that the convention for specifying the viewport has been changed from OpenGL ES.

In OpenGL ES, the viewport has a left hand Normalised Device Coordinates (NDC) space, while in Vulkan it was switched to a right hand NDC space. This change in convention essentially means the coordinate (-1, -1) maps to the bottom left corner in OpenGL ES, but the top left corner in Vulkan.

Keeping this in mind is important.

When porting an application from OpenGL ES to Vulkan, there are two options to dealing with this switch:

1. Convert the coordinates during runtime to ensure that it follows the correct coordinate space
2. Convert all of the assets used in the application to follow the new convention

The choice depends on the specifics on the application.

Command Buffers in Vulkan

Command buffers in Vulkan are similar to listing drawing commands in OpenGL ES. The major difference is that command buffers are much more flexible.

In simple terms, command buffers are sections of memory which store GPU commands, such as draw calls and memory transfer operations. Commands are recorded into command buffers in advance. When the application is ready for these commands to be executed, it can submit the command buffer to a queue. The GPU actually executes these commands asynchronously at some point after submission. Vulkan makes very few guarantees about execution order to give the GPU the freedom to decide the optimal way to order its processing. This means explicit *synchronisation*, through use of semaphores, fences, and events, is required.

Splitting recording and submission of commands is intended to improve performance, as the driver can determine, in advance, the most efficient way to divide the work between multiple threads.

Synchronisation in Vulkan

Synchronisation objects in Vulkan were discussed briefly in *Comparing OpenGL ES and Vulkan*, but it is useful to go slightly deeper now that some of the basic concepts of Vulkan have been covered.

In Vulkan, concurrency takes three main forms:

- Between host and device
- Between commands in a queue
- Between queues

Types of synchronisation object

Vulkan provides four different object types to help with the synchronisation of the situations mentioned above.

These are:

- **Fences** - GPU to CPU syncs. They are signalled by the GPU and can only be waited on by the CPU. They need to be reset manually and are often waited on by functions such as `vkWaitForFences()`.
- **Semaphores** - GPU to GPU syncs. They are specifically used to sync queue submissions on the same or different queues. They get signalled by the GPU and can only be waited on by the GPU. As long as they have been waited on they are reset automatically.
- **Events** - These can be set, reset and checked on both CPU and GPU. However, they can only be waited on by the GPU. They are limited within a single queue and can also be used to synchronise work within a command buffer.
- **Barriers** - These are used to synchronise operations in the command buffer between different stages of the pipeline. There are five different types of barriers:
  - **Execution Barriers** ensure that any stage specified as a source stage is executed and completed by the command, before the stage specified as destination is started.
  - **Memory Barriers** ensure the caches are flushed and invalidated between the source stage and destination stage of the execution barriers. They make all resources needed by the stage available at the time it needs for execution.
  - **Global Memory Barriers** apply to all the memory objects that exit at the time of its execution and not only the ones that are needed by the stage.
  - **Buffer Memory** apply only to the buffer that is being working on.
  - **Image Barriers** perform image layout transitions and operations on sub-regions of the image.

6. Summary of Vulkan: Migrating from OpenGL ES

Vulkan provides developers with many exciting new features, such as explicit control over all aspects of rendering, precise synchronisation and the possibility of improved performance and smoother frame rates. In addition, explicit memory allocation provides the power to better utilise GPU memory, and fully eliminate ghosting.

More broadly, Vulkan can make better use of many modern platforms with its multi-core support. It also has broad cross-platform support and is an open standard, meaning knowledge of the API will be useful for years to come.

For developers it is simple to get started with Vulkan, as major game engines such as Unity and Unreal already support it.

Ultimately, whilst moving to Vulkan will require more effort than sticking with OpenGL ES, it will most likely be worthwhile in the long term.

Where to start?

This document has only really touched very briefly on the features and design of Vulkan. For a complete description of all that Vulkan has to offer take a look at the *Vulkan specification*.

Finally, while the specification is thorough and detailed, it definitely helps to see Vulkan in action. *Getting Started with Vulkan* is a step-by-step guide demonstrating how to render a simple textured triangle on screen. It shows how many of the features mentioned in this document are used in a simple application. This guide is based on a demo included with the PowerVR SDK called *HelloAPI*