Vulkan

Migrating from OpenGL ES

Filename	:	Vulkan.Migrating from OpenGL ES
Version	:	PowerVR SDK REL_18.2@5224491a External Issue
Issue Date	:	23 Nov 2018
Author	:	Imagination Technologies Limited

Contents

1. Introduction

This document outlines the key differences between OpenGL ES and the new Vulkan, and why a developer would want to migrate to Vulkan.

Vulkan is a new low level graphics API that allows the developer to get very low level with an almost console-like API. This allows for greater control, performance and transparency. This is offset by an increase in implementation complexity.

2. Paradigm shift

2.1. Explicit control

2.1.1. Transparent driver model

When using previous generation graphics APIs, the graphics driver might often seem like a black box to the developer. Everything is hidden and abstracted away and a developer may feel they do not have any control over what happens behind the scenes. The developer only instructs the API to do a specific task, and it is up to the graphics driver to decide how to do it.

With the new Vulkan API, the developer has explicit control over what the hardware does and does not do. The driver layer is significantly slimmer, everything is exposed and the developer has full control. The developer tells the driver exactly what they want to happen, and the graphics driver has to do minimal guesswork.

2.1.2. API objects/states

The old way that graphics APIs followed was that there was a single global render state that the developer could modify. Based on this state the driver had to figure out what commands to create on the fly that the GPU can execute. To add to these issues, this approach was very error prone. This is because the developer had to make sure that at the point of issuing a draw command the render state was correct and what they want it to be.

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

2.1.3. Synchronisation

Synchronisation in software engineering can be a challenging problem. That is why in the previous generation of APIs it was hidden from the developer. Only coarse-grained synchronisation was possible in some places. In some cases it would be expected that the driver would synchronise the CPU and the GPU, in other words one of them would have to wait for the other. This also meant that sync points could exist without the developer knowing about them, and performance could be poor for no apparent reason.

In the new Vulkan API, synchronisation is explicitly specified in a fine-grained manner. This is supported by fences, semaphores, events and barriers:

- **Fences** are used to communicate the completion of execution of command buffer submissions to queues
- **Semaphores** can be used to marshal ownership of shared data
- **Events** provide fine-grained synchronisation primitives that can be signalled and waited upon at command level granularity
- **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 could make migrating to Vulkan worthwhile.

2.1.4. Explicit image layout transitions

One of the things that used to happen outside the developer's control with the old APIs was 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. Drivers always tried to keep images (textures) in the best possible layout so that sampling or using them as render targets always had the best performance. Unfortunately sometimes this resulted in unnecessary transitions.

In the new Vulkan API, the developer can explicitly specify when and what kind of image layout transitions to do. It is still the driver's task to perform the transitions though.

2.2. User-managed memory allocation

Another task specific to older APIs is memory allocation. Originally, developers could only specify the amount of memory required, and specify certain hints on what it would be used for.

In the new Vulkan API, the developer has the ability to request large memory blocks from the driver and sub-allocate it however it is needed. This allows for finer-grained control over memory allocation, and better resource lifetime tracking.

2.3. 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 copies of the resources (ghost copies – ghosting) invisible to the developer so that the resource can be updated.

In the new Vulkan API, developers have to explicitly manage the memory they allocate, and they have to explicitly synchronise memory accesses. Therefore Vulkan completely eliminates ghosting, but developers would need to implement a method to allow for modifying objects that are being used by the hardware.

One way for developers to still be able to modify buffers that are in use is to implement a ring buffer with sufficient synchronisation. This way, one can update a copy while the other buffer is in use by the hardware.

2.4. Full control over resource lifespan

With the older APIs a developer was only able to mark resources to be deleted, but it was up to the driver if/when to actually free the resource as the resource might still be in use by the GPU. This could result in spikes in frame times and unpredictable performance.

In the new Vulkan API, the developer has full control over when resources are freed, which results in predictable performance and fewer frame time spikes.

2.5. Scaling to multiple cores

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

The new Vulkan API considers multi-core architectures as first-class citizens, providing many features to aid distributing work over multiple cores.

Vulkan splits the draw call submission into two parts: command buffer generation and command buffer submission. This allows for distributing the generation of individual command buffers to multiple cores and after it is done, a core can submit all the buffers generated. As the command buffer submission is relatively cheap, the serial part of the workload is kept to the minimum.

For further information see this blog post on the topic: https://www.imgtec.com/blog/vulkan-scaling-to-multiple-threads/

2.6. Shader binaries

Shader compiling always used to be an expensive complicated operation. With older APIs precompiling shaders was impossible. Shader compilation, including optimisation, had to be done on the target device as each vendor had their own binary format. Although shader caching could be utilised, this helped less and less, as applications had more and more shader variations to compile. The timing of shader compilation was also hidden from the developer. The driver usually compiled shaders just before they were needed, leading to spikes in frame time.

The new Vulkan API proposed a new vendor independent intermediate format called SPIR-V. 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 also changed, as now entire pipeline states can be cached. This somewhat increases the shader variations needed to be cached, but it helps the graphic driver a lot that it knows the entire state. The developer has full control over when is a shader compiled and if/when a shader is cached.

2.7. Error handling

One of the many things that increased driver overhead with the older APIs is that they needed to validate the global state for errors all the time. This is because the application could query the error state after each and every API call. This resulted in a lot of unnecessary overhead in release environments.

To alleviate the driver overhead, the new Vulkan API removed the runtime error-checking mechanism, and introduced debug layers and tools that are meant to be used only at development time. This way developers could still validate errors effectively, but release builds no longer have this overhead.

2.8. Pixel Local Storage support

Tile based architectures have the ability to accelerate effects with their fast Pixel Local Storage (PLS) memory. Utilising this memory has the benefit of significantly reduced bandwidth usage and battery life savings. The old APIs used to support this feature via extensions.

In the new Vulkan API, PLS memory - and therefore tile based architectures - is a first-class citizen and the API has direct support for it through renderpasses and subpasses. These two features allow the developer to specify dependencies between effects in detail so that data can be kept in PLS.

3. Pros/cons of Vulkan

Moving to Vulkan is only comparable to moving from fixed function pipelines to shader-based pipelines. The change was huge, there was a lot to learn, but in the end the amount of flexibility, control and freedom it gave developers, truly revolutionised the industry.

The recent release of many new rendering APIs such as DirectX 12, Mantle, Metal and Vulkan shows that while it is not entirely clear what the best way is to tackle driver overhead and full control, it is clear that all these APIs head in a similar direction. So moving to one of the next generation APIs is genuinely a move into the future.

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 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 low-level API that the platform supports. 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 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 engines include Unity, Unreal Engine 4, Cryengine, and more. This means that to take advantage of the new Vulkan API without creating an engine from scratch, one of the major game engines can be used to get the benefits of Vulkan without the hassle.

One of the things that make it easier to learn Vulkan is if a developer already knows older graphics APIs. It has the same graphics pipeline, but a different interface (API) to control it. This makes learning Vulkan quite a bit easier and porting techniques from older APIs is easy as well.

4. When is it beneficial to switch to Vulkan?

Moving to Vulkan is always a forward-looking step and it is always beneficial, but some developers might not have the extra resources to afford the move. Therefore, here are a few situations where it is more rewarding to switch to Vulkan, and might still be worth doing even if a developer has scarce resources.

4.1. Situations where Vulkan can help

CPU-bound application and parallelisable graphics work submission

Vulkan particularly excels at making heavy API usage CPU-bound applications faster. The low driver overhead it provides really helps in these situations. It can also help if the developer needs to parallelise the graphics work for submission.

Few platforms targeted and max performance is required

As it is quite difficult to create an optimised build for each and every platform, it might be difficult to fully utilise Vulkan's capabilities. However, when only a handful of platforms need to be supported, a lot more effort can go into optimisation. In these situations it might be very worthwhile to switch to Vulkan and get the most out of the hardware.

As the developer has full control over synchronisation with Vulkan it is possible to reduce frame rate spikes, hitches and jitters. It is still up to the developer to take the opportunity, but it is certainly there.

4.2. Situations where Vulkan can help, but requires effort

Need support for pre-Vulkan platforms

When a developer needs to support pre-Vulkan platforms, moving to Vulkan will certainly add more code to support.

Heavily GPU-bound application

When an application is heavily GPU-bound, removing driver overhead might not help that much. However, it will certainly reduce CPU usage, and therefore power consumption and heat, so it is possible that the GPU may be able to run on higher frequencies.

Heavily CPU-bound application due to non-API work

In the case of a heavily CPU-bound application that is mainly bottlenecked by non-API work, Vulkan might not provide that many benefits. It may help a little as there would be more time for the CPU to do everything else, but certainly it may be a good idea to optimise the non-API workload first.

Single threaded application, unlikely to change

In the case where an application is unlikely to be changed to use multiple threads, there's fewer benefits Vulkan can provide.

4.3. Situations where Vulkan cannot help

Application performance is fine

It is a common situation nowadays on mobile that the application simply does not saturate resources at all – for instance simple 2D games - and 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. That said, in the case of creating a new application it might be worthwhile to move to Vulkan.

Already written applications that are GPU-bound and heavily optimised

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

5. Key API differences

5.1. Object binding

One key difference between older APIs and the new Vulkan API, is that API objects no longer need to be bound to a global state to be able to use them. Graphics pipelines, vertex/index buffers and descriptor sets are now bound to thread-local command buffer objects.

Graphics pipelines describe all shader related states. These include which render pass it will use, rasterization/blending/depth/stencil state, vertex attribute layout, and shaders. They are compiled before actually being used, so the driver does not need to do any guesswork about what the shader will do.

Descriptor sets describe all shader-bound resources such as what buffers, textures, and samplers are bound. This way the available binding slots are completely configurable.

5.2. Uniforms

In the new Vulkan API, the fast on-chip constant memory is now exposed in the form of push constants. While this memory is limited, it does allow for very frequent (per draw call) updates of shader constants. This is because unlike buffer objects, they do not need to be mapped/unmapped every time the developer wishes to update them.

There are multiple areas to consider when using push constants. One of these is command buffer recording. Using push constants means that command buffers need to be re-recorded each frame, and cannot be cached. This wastes a lot of CPU time as the same operations need to be done each frame. Therefore push constants should only be used when command buffers cannot be re-recorded anyway.

However, uniform buffers should be used when command buffers can be re-recorded. It is worth considering that push constant space is limited and it is shared with other resources as well. It is therefore a good idea to minimise the amount of data put into push constants.

5.3. Clear, discard, invalidate

Clear, discard and invalidate calls are now replaced with load/store operations specifiable in the subpasses.

- **Load operations** specify what to do with subpass attachment when rendering begins. These options are to clear it, load it from memory, or leave it uninitialised.
- **Store operations** specify what to do with subpass attachments when rendering ends, whether to store results or throw them away.

6. Overview of porting applications to Vulkan

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

The Vulkan API has a level of verbosity that makes it quite complex and hard to maintain. Vulkan improved performance comes at a price - Vulkan requires more maintenance and a higher level of responsibility to make sure that everything is correct.

OpenGL ES in built in such a way that hides a lot of the operations required to run the application correctly. Vulkan in this regard is more hands-off and requires code to explicitly define and execute these operations. Operations such as resource management, synchronisation, error checking and validation, and compiling shaders would be taken care of by OpenGL ES automatically, but Vulkan will not.

As general advice, when possible, do not port to Vulkan but rather rewrite in Vulkan and port to OpenGL ES. This ensures the best possible performance gain from using Vulkan.

6.1. Wrapping

As stated above, Vulkan is a verbose API, so it is sometimes not easy to use. It can take some time to get used to, and there tends to be learning curve that initially results in errors.

The best way to avoid these inconvenient issues is by wrapping the Vulkan part of the application into 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 using Vulkan are taken care of. This way, the developer is free to focus on the development of the application side while still maintaining all the benefits of using the Vulkan API.

For more information on the PowerVR Framework, please read our PowerVR Framework Development Guide available through our SDK, or alongside this document on our website.

7. Technical overview of Vulkan

7.1. Loader

The Vulkan Loader sits between the application, and the Layers and Installable Client Driver (ICD). It takes care of connecting the application with the devices and setting up the layers.

There are two types of loaders: a desktop loader and a mobile loader, although the latter is only for Android and it is closed source. The Vulkan loader can be acquired by installing a Vulkan application. This is through the OS in the case of Android, but it can also be in a driver package or included in an SDK.

The Vulkan Loader runs on a concept called a call chain. The call chain is the sequence of function calls from the application to its final destination. There are two types of call chains, an instance call chain and a device call chain.

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

At the end of the call-chain there is a terminator. A terminator is loader code that distributes calls to multiple ICDs. In case of device call-chain with only one ICD, a terminator is not needed.

7.2. Layers

The Vulkan API requires the developer to be very explicit about everything they do. This is an intentional design of the API. It achieves minimal driver overhead, but it is more prone to validity errors.

Vulkan provides help with debugging applications in the form of a layer framework. API entry points, or specific subsets of entry points, are intercepted by the layers in the framework. Validation layers can then be used to help debug and validate the code. Multiple layers can be chained sequentially to enhance their functionality.

Validation layers are optional components and they can be disabled in the final version of the application. Validation layers only help with validity errors.

An example of these errors is the creation and destruction of 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 to the standard output.

In order to recognise the type of error the validation layers will catch, here are the two main types of Vulkan errors:

- **Validity errors** are errors that stem from the incorrect usage of the API. This is 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. The Validation layers help with catching these errors.
- **Runtime errors** may 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.

Setting up the validation layers is one of the first steps to start development in Vulkan. The next step is defining the needed extensions to run the application.

7.3. Extensions

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

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

Vulkan does not make assumptions on the type of application being developed, as it could be a compute or graphics application. For this reason, surfaces and the swapchain are both considered extensions that add functionality to the core API. The surface extension is an instance extension, while the swapchain is a device one.

7.4. Initialisation

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

The initialisation of objects is divided into two parts.

- the *info struct* is defined
- the info structs are then used to create an object of the desired type

An info struct is essentially a collection of information and parameters that will be added to the object when it is created. Vulkan requires that everything is defined in advance.

The next steps are:

- the Vulkan application handle is instantiated
- available physical devices (the GPU) are queried to find the one required for the application
- a surface is defined to present the result of the rendering operation

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

7.5. Devices and queues

After selecting a physical device, the device needs to be queried for available queues. In Vulkan all commands need to be executed on queues. Each device makes queues available for certain operations such as compute, rendering and so on.

Queues that share certain properties, for instance those used to execute the same type of operation, are grouped into queue families. A queue family needs to be selected that fits the desired operation.

After the physical device and queue family has been selected, their handles can be used to generate a logical device handle. A logical device handle is required for the creation of most other handles. It can be used to create the queues used to render and present the application.

7.6. Swapchain

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 begins by creating an instance, a device and finally creating a swapchain. A Vulkan application may not even render anything at all and therefore does not require a surface or framebuffers. For this reason a swapchain is an extension in Vulkan.

If an application is being developed that displays something, a swapchain needs to be created and its properties defined. A swapchain is a series of images that are used to render and then present to the surface. On changing the screen size or other changes, the swapchain needs to be destroyed and recreated at runtime.

7.7. Renderpass

A renderpass is a set of data necessary to accomplish a rendering operation. In Vulkan, a renderpass is a collection of data that describes a set of framebuffer attachments that are needed for rendering. It is composed of subpasses that order this data.

A renderpass collects all the colour, depth and stencil attachments and makes sure to explicitly define them so that the driver does not have to work them out itself. Renderpasses, and their subpasses, define all aspects of the attachment involved in the rendering operation, including the dependencies between them. They also define the input attachments that provide the initial data. This helps the overall optimisation of the operations on the GPU.

Where the renderpass describes the attachments, the framebuffers define the attachments themselves.

7.8. Viewport

In OpenGL ES, the viewport has a left hand Normalised Device Co-ordinates (NDC) space, while in Vulkan it was switched to a right hand NDC space. Keeping this in mind is important. The options when porting the application to Vulkan are either:

- convert the coordinates during runtime to ensure that it follows the correct coordinate space
- convert all of the assets used in the application to follow the new convention

7.9. Framebuffers

In OpenGL ES, framebuffers are strictly defined in 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 the render pass are defined in framebuffers. Each frame in a framebuffer defines the attachments related to it. This can encompass the textures which includes the colour and depth/stencil attachments, and the input attachment.

This method of separating descriptions in renderpasses and definitions in framebuffers gives the option to use different renderpasses with different framebuffers. The degree of flexibility with which this can be done is based on the compatibility of the two.

7.10. Pipeline

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

In Vulkan, there are two types of pipelines: a graphics and a compute pipeline. The graphics is used for rendering operations, while the compute allows the application to perform computational work such as physics calculations.

In OpenGL ES, the pipeline is composed of programmable stages and fixed function stages. Programmable stages such as vertex and fragment shaders are mutable stages, while the fixed function stages such as the rasterizer and blending are immutable stages.

The states and parameters for each stage can be changed to perform operations. The pipeline can be set up render an object, and then change the programmable stages to render another object.

In Vulkan, the pipeline’s structure is virtually identical to the one in OpenGL ES. The pipeline is stored in one object that is immutable. This means that each object that needs to be rendered will potentially use a different pipeline. The pipeline in Vulkan needs to be prepared before it is used, which helps with increasing the performance of the application.

Creating and defining the shader programs associated with the pipeline is another important part of creating the pipeline.

7.11. Command buffers

Command buffers in Vulkan are similar to listing drawing commands in OpenGL ES. The major difference is in the flexibility that Vulkan provides when using command buffers.

Command buffers are containers that contain GPU commands. They are passed to the queues to be executed on the device. Each command when executed performs a different task. The command buffer required to render an object is recorded before the rendering itself happens. Once recorded, the command buffer is then used to render its content by playing it back. When the rendering stage of the application is reached, the command buffer is submitted to execute its command.

7.12. Descriptors

Descriptor sets in Vulkan are required in order to pass data to shaders. Descriptors, as the name implies, are used to describe the data to be passed. They hold information that helps with binding data to shaders, and additionally describe any information Vulkan requires to know before executing

the shader. Descriptors are not passed individually and are opaque to the application, but are instead bundled in sets, known as descriptor sets.

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

- Create the descriptor pool that is used to allocate descriptor sets
- Create a descriptor layout that defines how the descriptor is laid out, with information on the binding points and the type of data to be passed to the shader
- The descriptor sets themselves hold in the form of a pointer the data that is needed to be passed to the shader, for instance textures, uniform buffers and so on

7.13. Shaders

In Vulkan, shaders are in SPIR-V format which is a byte-code format rather than a human readable one. SPIR-V can be used for both graphical and compute operations, so as such it is also compatible with OpenCL. The advantage to using this format is that it gives less flexibility to vendor compilers on how to interpret the code. It streamlines the portability of the shader across the different vendor compilers that have to turn the shader into native code.

To help with compile code from GLSL to SPIR-V, Khronos released a vendor agnostic offline compiler called glslang. glslang also automatically error checks and verifies that the shader is standard compliant.

7.14. Synchronisation

In Vulkan, concurrency takes three main forms:

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

The Vulkan API provides different object types to help with the synchronisation of the situations mentioned above. These are described below:

- **Fences** are GPU to CPU syncs signalled by the GPU. They can only be waited on by the CPU. They need to be reset manually.
- **Semaphores** are GPU to GPU syncs. They are specifically used to sync queue submissions on the same or different queues. They get signalled by the GPU but can only be waited on by the GPU. They are reset after being waited on.
- **Events** 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. They can also be used to synchronise work within the command buffer.
- **Barriers** are used to synchronise operations in the command buffer between different stages of the pipeline. The following are the types of barriers:
    - **Execution Barriers** make sure 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** make sure 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 Barriers** apply only to the buffer that is being working on.
    - **Image Barriers** perform image layout transitions and operations on sub regions of the image.

7.15. Memory allocation

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 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 CommandBuffers and DescriptorSets. The actual content is indirectly written by the driver.

8. Summary

In summary, moving to Vulkan is a forward-thinking move. It provides the developer with many exciting new features such as explicit control over all aspects of rendering, enabling precise synchronisation and the possibility of smoother frame-rates. Explicit memory allocation provides the power to better utilise GPU memory, and fully eliminate ghosting.

Multi-core platform support makes Vulkan optimal for modern many CPU core platforms. It also has support for most platforms on the market.

For developers it is out already, and ready to be used with one of the engines that support it.

Whilst moving to Vulkan requires more effort than before, it is usually more than worthwhile.

Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp