PowerVR Performance Recommendations

The Golden Rules

Filename	:	PowerVR Performance Recommendations.The Golden Rules
Version	:	PowerVR SDK REL_18.2@5224491a External Issue
Issue Date	:	23 Nov 2018
Author	:	Imagination Technologies Limited

Contents

The Golden Rules

This document covers key principles developers should follow in order to avoid critical performance flaws in their graphics applications. These recommendations come from the combined experience of the PowerVR Developer Technology Support team and the developers they work with, profiling and optimising their applications and games during development.

1. **Do understand the target device**
2. Seek to learn as much information about the target platforms as possible in order to understand different graphics architectures, to use the device in the most efficient manner possible.
3. **Do profile the application**

   Identify the bottlenecks in the application and determine whether there are opportunities for improvement.
4. **Do not use Alpha Blend unnecessarily**

   Be sure Alpha Blending is used only when required to make the most of deferred architectures and to save bandwidth.
5. **Do perform Clear**

   Perform a clear on a framebuffer's contents to avoid fetching the previous frame's data on tile-based graphics architectures, which reduces memory bandwidth.
6. **Do not update data buffers mid-frame**

   Avoid touching any buffer when a frame is mid-flight to reduce stalls and temporary buffer stores.
7. **Do use texture compression**

   Reduce the memory footprint and bandwidth cost of the texture assets.
8. **Do use mipmapping**

   This increases texture cache efficiency, which reduces bandwidth and increases performance.
9. **Do not use Discard**

   Avoid forcing depth-test processing in the texture stage as this will decrease performance in the early depth rejection architectures.
10. **Do not force unnecessary synchronisation**

    Avoid API functionality that could stall the graphics pipeline and do not access any hardware buffer directly.
11. **Do move calculations 'Up the Chain'**

    Reduce the overall number of calculations by moving them earlier in the pipeline, where there are fewer instances to process.

1. Do understand the target device

Seek to learn as much information about the target platforms as possible in order to understand different graphics architectures, to use the device in the most efficient manner possible.

Manufacturers' websites for devices are a good place to look for specifications and they may also provide other helpful developer community resources. The PowerVR Graphics SDK provides public architecture and performance recommendation documents for reference:

- PowerVR Hardware Architecture Overview for Developers
- PowerVR Series5 Architecture Guide for Developers
- PowerVR Performance Recommendations
- PowerVR Low Level GLSL Optimisation
- PowerVR Instruction Set Reference

Further PowerVR architecture documentation is available from us under a non-disclosure agreement.

Even after the graphics architecture is thoroughly understood, it is important to remember that other factors such as variations in CPU processing power, memory bandwidth, and thermal load will also impact an application’s performance.

2. Do profile the application

Identify the bottlenecks in the application and determine whether there are opportunities for improvement.

It is important to understand where performance is bottlenecked before attempting to optimise an application. This ensures effort is not wasted, or visual quality is not sacrificed for minimal gains. If an optimisation is inappropriately applied to an area that is not bottlenecking performance, there may be no performance improvements. In some cases, an incorrectly applied optimisation may lead to worse performance.

From the PowerVR Developer Technology team's experience, we have derived the following list of common bottlenecks generally found in applications that have not been optimised, as ordered from most to least common:

- CPU usage
- Bandwidth usage
- CPU/graphics core synchronisation
- Fragment shader instructions
- Geometry upload
- Texture upload
- Vertex shader instructions
- Geometry complexity

Profiling tools are vital in this process for developers to understand what is happening in their application, the hardware it is running on, and how and where bottlenecks are occurring. The PowerVR SDK includes the profiling tools PVRTrace and PVRTune to aid development on platforms powered by PowerVR hardware.

3. Do not use Alpha Blend unnecessarily

Be sure Alpha Blending is used only when required to make the most of deferred architectures and to save bandwidth.

Disable alpha blending wherever possible. If transparent objects are required, keep the number of transparent objects to a minimum. The reasoning behind this is that deferred renderers, such as PowerVR graphics cores, calculate the visibility of fragments before the corresponding fragment shader is invoked to process it. This prevents invisible fragments in the output image being processed unnecessarily.

If alpha blending is enabled, then the hardware used to determine a fragment's visibility cannot be used. This is because the occluded (alpha-blended) fragment may impact the final rendered image. Due to this behaviour, enabling alpha blending eliminates the benefits of deferred rendering graphics architectures. This means the hardware is no longer able to make decisions about a fragment's visibility and drop it from the pipeline. This will likely result in overdraw which is where fragments are being processed that are not actually visible in the final image. Overdraw can negatively impact the application's performance, particularly if the application is already limited by rendering.

4. Do perform Clear

Perform a clear on a framebuffer's contents to avoid fetching the previous frame's data on tile-based graphics architectures, which reduces memory bandwidth.

System memory accesses use more bandwidth and power than any other graphics operation. Keeping memory accesses to a minimum will reduce the chances of an application being memory bandwidth bound, and will also reduce the power consumption of an application.

Most applications need to generate a colour image at the end of the render, but have no need to preserve depth and stencil data between frames. Therefore, if framebuffer attachments do not need to be preserved at the end of a render, the appropriate framebuffer attachments can be invalidated to prevent them being written out to system memory.

Even fewer applications have a genuine need to upload the contents of the colour buffer's previous contents at the start of a new frame. Therefore, if the contents previously written to a framebuffer are not required, the driver can be informed not to load them from system memory to on-chip tile memory through a *clear* operation at the start of the render.

The net result of performing a clear and invalidating framebuffers will be a massive reduction system memory bandwidth usage and reduced power consumption.

In OpenGL ES a clear can be performed by calling the `glClear` function at the beginning of a render. Additionally, the `glDiscardFramebufferEXT` or `glInvalidateFramebuffer` functions can be used to invalidate a framebuffer at the end of a render.

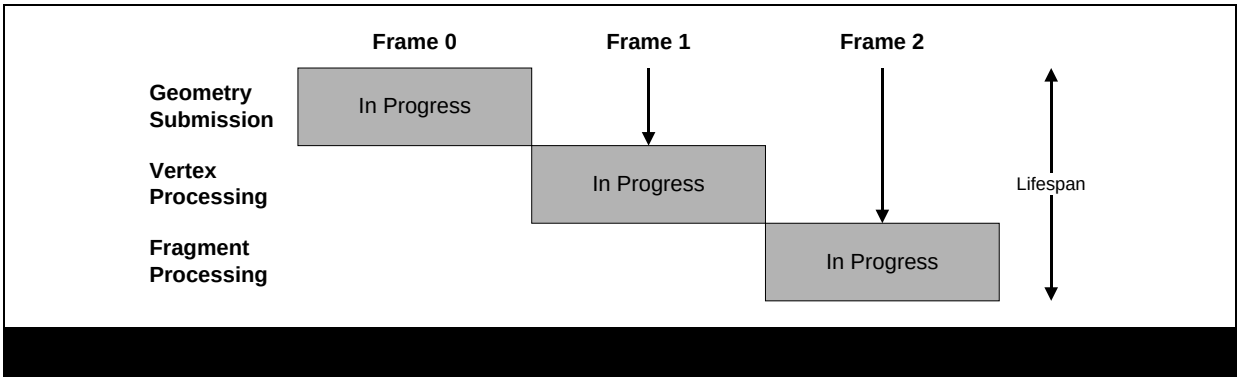
In Vulkan the API gives explicit control over load and store operations on framebuffer attachments. When creating a framebuffer, set the load operation to either `VK_ATTACHMENT_LOAD_OP_DONT_CARE` or `VK_ATTACHMENT_LOAD_OP_CLEAR`. The store operation should preferably be set to `VK_ATTACHMENT_STORE_OP_DONT_CARE` unless the data requires preserving.

5. Do not update data buffers mid-frame

Avoid touching any buffer when a frame is mid-flight to reduce stalls and temporary buffer stores.

Modifying in-flight resources currently in use by the GPU such as vertex buffers and textures has a significant cost. Graphics processors tend to have at least one frame of latency to ensure that the hardware is always well occupied with work. Therefore, altering a resource required by an outstanding render will usually result in one of the following actions being taken:

1. Stall in the buffer modifying API call until the outstanding render completes
2. A new temporary buffer allocated for the new data, so the buffer modifying API call can complete without stalling the CPU thread



As textures are generally accessed during fragment shading much later in the graphics pipeline than vertex attributes, the cost of a graphics driver stalling a texture modification is higher than modifying a vertex buffer. The driver may choose to avoid a stall entirely by creating temporary buffer stores (a.k.a. ghosting) which is good for performance, but it may not be desirable for applications that are already running out of buffer storage space.

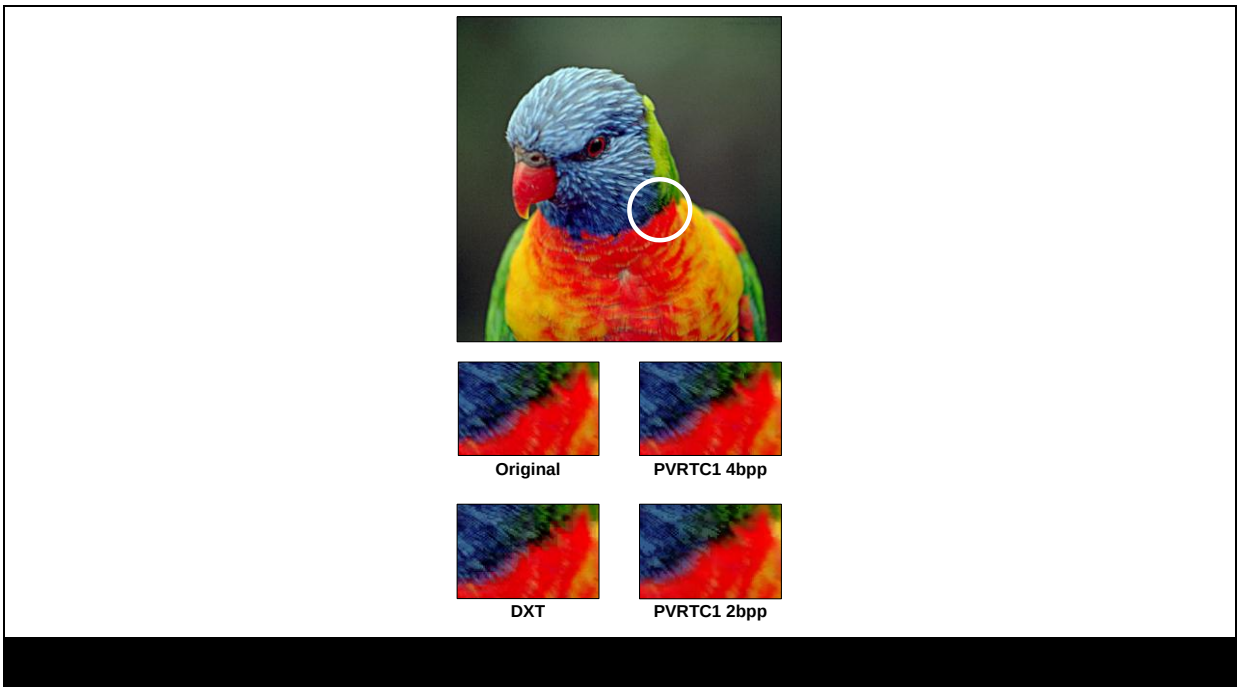
The stalling and ghosting behaviour of graphics processors varies between different GPUs and driver versions. For optimal performance, only modify vertex buffers and textures when absolutely necessary. If buffers must be modified, use application-side circular buffering so that the graphics processor can read from one buffer object while the application's CPU thread writes to another. This prevents the stalling and ghosting behaviours.

If the application is using the Vulkan graphics API, then it is the responsibility of the application developer to synchronise with the graphics processor. The appropriate mechanisms such as fences and semaphores must be put in place, to ensure that the application does not access a resource while the graphics processor is using it. This gives much more control over how and when resources are accessed, but comes at the cost of a more complex application as the driver will not safeguard against accessing data currently in use by the graphics processor.

6. Do use texture compression

Reduce the memory footprint and bandwidth cost of the texture assets.

In some instances, it is worth considering the balance between texture size and texture compression. It may be possible to use a larger texture and a low-bitrate compression scheme and achieve a better balance of bandwidth savings and acceptable image quality.



Texture compression, not to be confused with image file compression, minimises the runtime memory footprint of textures. This provides several performance benefits, but primarily reduces the amount of system memory bandwidth consumed sending data to the Graphics Core.

PVRTC and PVRTCII are PowerVR specific compression technologies and will achieve best performance on the hardware, consuming as little as 2 bits per pixel. These textures are also very texture cache efficient as the lower pixel size allows more pixels to fit in the limited amount of cache memory available to the texture units.

Depending on the PowerVR generation and graphics API targeted, additional compressed texture formats may be supported, such as ASTC.

7. Do use mipmapping

This increases texture cache efficiency, which reduces bandwidth and increases performance.

Mipmaps are smaller, pre-filtered variants of a texture image, representing different levels of detail of a texture. By using a minification filter mode that uses mipmaps, the graphics core can be set up to automatically calculate which level of detail comes closest to mapping the texels of a mipmap to pixels in the render target. This means it can then use the right mipmap for texturing.

Using mipmaps has two important advantages:

1. It increases graphics rendering performance by massively improving texture cache efficiency, especially in cases of strong minification – the texture data is more likely to fit inside tile memory
2. It also improves image quality by reducing aliasing that is caused by the under sampling of textures that do not use mipmapping

The single limitation of mipmapping is that it requires approximately a third more texture memory per image. Depending on the situation, this cost may be minor when compared to the benefits in terms of rendering speed and image quality.

There are some exceptions where mipmaps should not be avoided. For example:

- where filtering cannot be applied sensibly, such as for textures that contain non-image data such as indices or depth textures
- textures that are never minified, such as UI elements where texels are always mapped one to one to pixels

Ideally mipmaps should be created offline using a tool like PVRTexTool, which is available as part of the PowerVR Graphics SDK.

It is possible to generate mipmaps at runtime, which can be useful for updating the mipmaps for a render to texture target. In OpenGL ES this can be achieved using the function `glGenerateMipmap`. In Vulkan there is no such built in function, and developers must generate mipmaps manually.

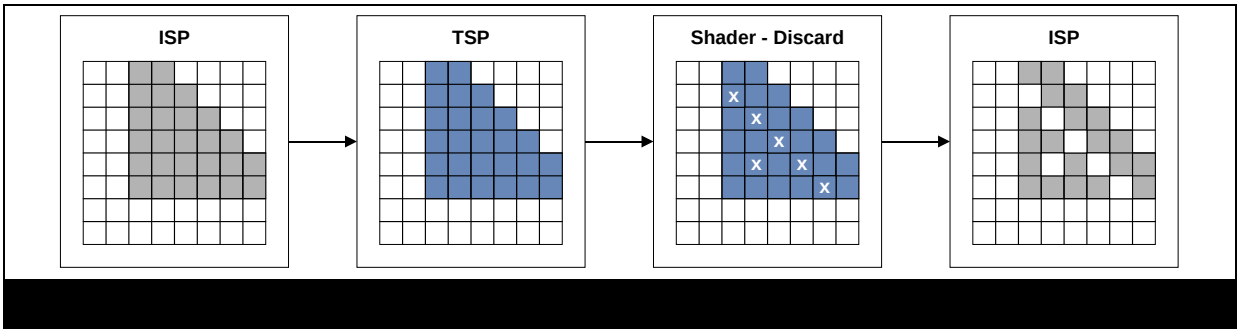
Generation of mipmaps online will not work with compressed textures such as PVRTC, which must have their mipmaps generated offline. A decision must be made as to which cost is the most appropriate: the storage cost of offline generation, or the runtime cost (and increased code complexity in the case of Vulkan) of generating mipmaps at runtime.

8. Do not use Discard

Avoid forcing depth-test processing in the texture stage as this will decrease performance in the early depth rejection architectures.

Applications should avoid the use of the discard operation in the fragment shader as using it will not improve performance. Most mobile graphics cores use a form of tile based deferred rending (TBDR) and using discard negates some of the benefits of this type of architecture. If possible an application should prefer alpha blending over discarding.

Applications should also avoid alpha testing. When an alpha tested primitive is submitted, early depth testing, such as PowerVR's Hidden Surface Removal (HSR), can discard fragments that are occluded by other fragments closer to the camera. Unlike opaque primitives that would also perform depth writes at this pipeline stage, alpha tested primitives cannot write data to the depth buffer until the fragment shader has executed and fragment visibility is known. These deferred depth writes can impact performance, as subsequent primitives cannot be processed until the depth buffers are updated with the alpha tested primitive's values.



For optimal performance, consider alpha blending instead of alpha test to avoid costly deferred depth write operations. To ensure HSR removes as much overdraw as possible, submit draws in the following order:

1. opaque
2. alpha-tested
3. blended

9. Do not force unnecessary synchronisation

Avoid API functionality that could stall the graphics pipeline, and do not access any hardware buffer directly.

Graphics applications achieve the best performance when the CPU and graphics core tasks run in parallel. Graphics cores also operate most efficiently when the vertex processing tasks of one frame are processed in parallel to the fragment colouring tasks of previous frames. When an application issues a command that causes the CPU to interrupt the graphics core, it can significantly reduce performance.

The most efficient way for the hardware to schedule tasks is vertex processing executing in parallel to fragment tasks. In order to achieve this, the application should aim to remove functions which cause synchronisation between the CPU and graphics core wherever possible.

- In OpenGL ES - synchronisation functions such as `glReadPixels`, `glFinish`, `eglClientWaitSync` and `glWaitSync`
- In Vulkan - developers have much finer control over synchronisation between resources, as any synchronisation between the graphics processor and CPU is defined by the developer

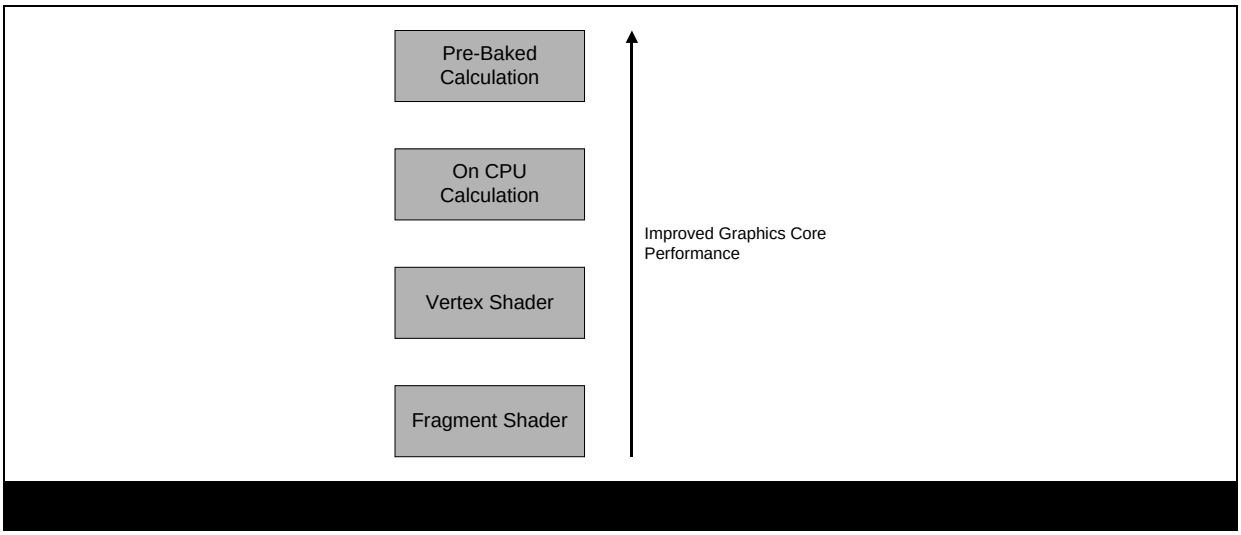
One of the most common causes of poor application performance is when the application accesses the contents of a framebuffer from the CPU. When such an operation is issued, the calling application's CPU thread must stall until the graphics core has finished rendering into the framebuffer attachment. Once the render is complete, the CPU can begin reading data from the attachment. During this time the graphics core will not have write access to that attachment, which can cause the graphics core to stall subsequent renders to that framebuffer.

Due to the severe cost, these operations should only be used when absolutely necessary - for example to capture a screenshot of a game when a player requests one.

10. Do move calculations ‘Up the Chain’

Reduce the overall number of calculations by moving them earlier in the pipeline where there are fewer instances to process.

By performing calculations earlier in the pipeline, the overall number of operations can be reduced, and therefore the workload can also be substantially reduced. Generally in a scene there are far fewer vertices than fragments that need to be processed. This means processing per vertex, instead of per fragment would greatly reduce the number of calculations. One use case, for example, could be to perform per vertex lighting instead of per pixel lighting.



It is also possible to consider moving calculations off the graphics core altogether. Although the graphics core may be able to perform operations far more rapidly than the CPU can, it would be even faster for the CPU to perform an operation just once instead of the allowing the operation to be performed for many vertices on the graphics core.

To take the concept even further, consider performing calculations offline by baking values into the scene, effectively replacing expensive run-time calculations with a simple lookup. For example, replacing real-time lighting with light maps for static objects in a scene, such as terrain, buildings and trees can be a particularly effective compromise. This substantially improves performance, and in many cases provides higher quality lighting than would be possible to calculate at run-time.

Other Considerations

Do group per material

Modifying the GL state machine incurs CPU overhead in the graphics driver, as changes need to be interpreted and converted into tasks that can be issued to the graphics core. To reduce this overhead, minimise the number of API calls and state changes made by the application.

For geometry data, combine as many meshes into a single draw call as possible. Here is an example use case:

Meshes for seats on a train have static position and orientation relative to one another, and use the same render state. The seats and the train could all be combined into a single mesh. To draw the train interior, several draw calls have merged into a single call.

With the Hidden Surface Removal (HSR) feature on PowerVR hardware it is not necessary to submit geometry in depth order to reduce overdraw. By freeing applications from this restriction they can focus on sorting draws by render state, ensuring state changes are minimised.

Similar to geometry data, it is possible to combine several textures into a single bindable object by using texture atlases or texture arrays where available. Textures can then be applied per object with the appropriate shader uniforms.

As discussed in Avoid Read/Write of In-Use Buffer Objects, modifying buffer data may stall the graphics pipeline or increase the amount of memory allocated by the graphics driver. When batching draws together, it is important to consider the update frequency of buffers. For example, batch spatially coherent objects with static vertex data into one vertex buffer, and objects with dynamic data, such as soft body objects like cloth, into another.

Do use indexed lists

Vertex buffers enable the graphics driver to cache vertex data attributes, such as texture coordinates for mapping 2D images to the mesh, and model/space position. For static objects which have vertex attributes that change infrequently if at all, vertex buffers improve performance as the cached data can be reused to render many frames.

In the example above, an index buffer is used in conjunction with a vertex buffer. Index buffers define the order in which elements of a vertex buffer should be accessed to represent the triangles in a mesh. Index buffers improve performance and reduce the storage space requirements of complex mesh data. This is because vertex attributes are written to the vertex buffer once, then referenced as many times as required to represent the triangles surrounding that vertex position.

PowerVR hardware is optimised for indexed triangle lists. For finely-tuned performance, vertex and index buffers should be sorted to improve cache efficiency when the data is accessed by the GPU. Our 3D scene exporter and converter tool, PVRGeoPOD, automatically applies sorting to mesh data when generating POD (PowerVR Object Data) files.

Do use all CPU cores

Modern mobile devices usually have more than a single CPU core. To achieve the best performance possible on modern CPU architectures, it is crucial that applications use multi-threading wherever possible. For example, consider having graphics updates on the main thread, while having physics updates running on a separate worker thread. Splitting large chunks of work such as physics, animations, file I/O and so on over multiple threads enables the application to use the CPU more efficiently, and will usually result in a smoother end user experience. If the application is targeting the Vulkan graphics API, it may be possible to split preparation of draw commands (building command buffers) over several threads.

Do prefer lower data precision

Variables in shaders declared with the mediump modifier are represented as 16 bit floating point (FP16) values. Applications should use FP16 wherever appropriate, as it typically offers a significant performance improvement by theoretically double the floating point throughput over FP32 (highp). It should be considered wherever FP32 would normally be used, provided the precision is sufficient and the maximum and minimum values will not overflow, as visual artefacts may be introduced.

Do use Level of Detail (LoD)

Level of Detail is an important consideration for an application, the concept of 'good enough' should be employed here. Application developers must consider the usage of expensive graphics effects and high quality assets against the impact to performance.

Mipmapping is one form of LoD, which was discussed with Rule 7. A second consideration for LoD is geometry complexity. An appropriate level of geometry complexity should be used for each object or portion of an object.

The following are examples of a waste of compute and memory resources:

- using a large number of polygons for an object that will never cover more than a small area of the screen, like a distant background object
- using polygons for detail that will never be seen due to camera angle, or culling – such as objects outside of the view frustum
- using large numbers of primitives for objects that can be drawn with much fewer, with minimal to no loss in visual fidelity. As an example - using many hundreds of polygons to render a single quad.

Shader techniques such as bump mapping should be considered to minimise geometry complexity, but still maintain a high level of perceived detail. This is especially true for techniques such as reflection passes, where higher amounts of geometry may not be visible.

Do not use depth pre-pass

On graphics hardware that employs a deferred rendering architecture such as PowerVR, an application should not perform a depth pre-pass as there is no performance benefit. Performing this operation would be a waste of clock cycles and memory bandwidth. This is because the hardware will detect and remove occluded (opaque) geometry from the pipeline automatically during rasterization, before fragment processing begins.

Do use on-chip memory efficiently for deferred rendering

Graphics techniques such as deferred lighting are often implemented by attaching multiple colour render targets to a frame buffer object, rendering the required intermediate data, and then sampling from this data as textures. While flexible, this approach, even when implemented optimally, still consumes a large amount of system memory bandwidth, which comes at a premium on mobile devices.

Both OpenGL ES (3.x) and Vulkan graphics APIs provide a method to enable communication between fragment shader invocations which cover the same pixel location – through intermediate on-chip buffers. This buffer can only be read from and written to by shader invocations at the same pixel coordinate.

The GLES extension *shader_pixel_local_storage(2)* and Vulkan transient attachments enables applications to store the intermediate per-pixel data in on-chip tile memory. While each method has its own implementation details, they both provide similar functionally and both bring the same benefits. For example the "G-Buffer" attachments in a deferred lighting pass that are only needed once can be stored in tile memory, and then completely discarded when drawing is complete.

Both of the API features described above are extremely beneficial for tile based renderers such as PowerVR graphics cores, as it allows an application to efficiently make use of on-chip tile memory. The intermediate framebuffer attachments are never allocated or written out to system memory - they only exist in on-chip tile memory. This is extremely beneficial for mobile and embedded systems where memory bandwidth is at a premium.

Using these features correctly will result in a significant reduction in system memory bandwidth usage. Additionally most techniques (such as deferred lighting) that write intermediate data out to system memory and then sample from it at the same pixel location can be optimised using these API features.

Do prefer explicit APIs

Vulkan is a new generation graphics and compute API. It is highly efficient, streamlined and modern and designed to take advantage of current and future device architectures. Vulkan works on a wide variety of platforms such as desktop PCs, consoles, mobile devices and embedded devices.

Vulkan is designed from the ground up to take advantage of modern CPU architecture such as multi-core and multi-threaded systems, rendering work can be spread over many logical threads. The Vulkan “Gnome Horde” demo in the PowerVR SDK shows this aspect of the API very nicely.

Vulkan is designed to have minimal driver overhead, but this comes at the cost of a more complex programming paradigm – explicit. In Vulkan it is up to the application developer to handle low level details such as memory allocation for buffers and explicit synchronisation between resources. However, once the API is mastered, a Vulkan graphics application is likely to run much more efficiently and more predictably across various devices compared to legacy graphics APIs.

Our PowerVR SDK includes a Framework, for developers targeting PowerVR platforms. This Framework reduces the need for boilerplate code, provides helpers, and more, making Vulkan development much easier.

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