PowerVR

Performance Recommendations

Filename	:	PowerVR.Performance Recommendations.1.0.28.External.doc
Version	:	1.0.28 External Issue (Package: POWERVR SDK REL_2.10@863987)
Issue Date	:	08 Feb 2012
Author	:	Imagination Technologies Ltd

Contents

List of Figures

1. Introduction

PowerVR SGX is a family of GPU cores from Imagination Technologies designed specifically for shader-based APIs like OpenGL ES 2.0. Due to its scalable architecture, the SGX family spans a huge performance range.

1.1. Document Overview

The purpose of this document is to serve as recommendation and advice for developers who wish to get the best graphics performance from a PowerVR SGX enabled device.

2. Optimal Development Approach

Ideally, a developer should aim to implement their application while obeying all (or as many as possible) of the techniques detailed in Section 3 The Golden Rules. It is crucial to adopt some of the practices detailed in this document from the very start of development in order to save much time and effort later.

Once an application is implemented to a level approaching its final state then a process of iteration should be adopted:

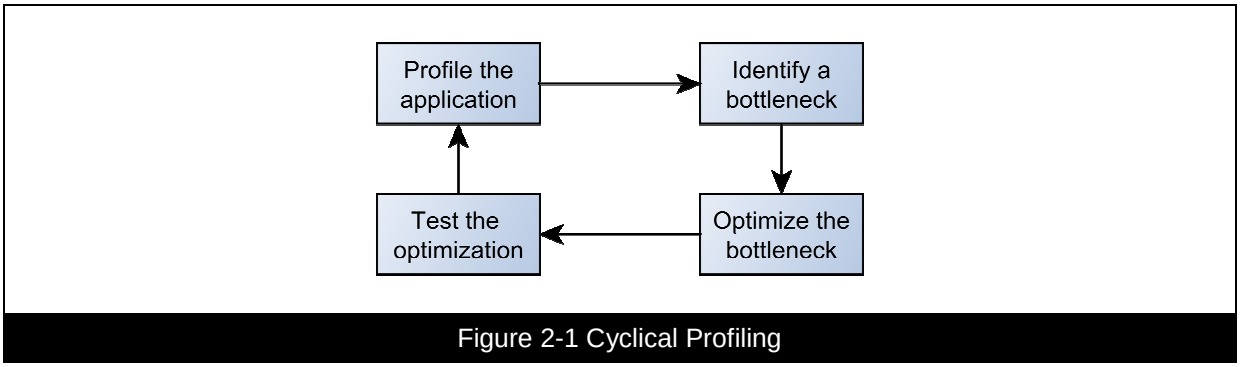

Figure 2-1 Cyclical Profiling

The main benefit of this approach is that time is not wasted and graphics quality not comprised by making changes that do not benefit performance.

2.1. Understanding Rendering Bottlenecks

It is a common misconception that the same actions can speed up any application. For example:

- Polygon Count Reduction

  If the bottleneck of the application is fragment processing or texture bandwidth then the only result of this action will be to reduce the graphical quality of the application without improving rendering speed. In fact, if simpler models actually cause more of the render target to be covered by a material with complex fragments then this can actually slow down an application.

- Reduce Rendering Resolution

  In this case, if the fragment processing workload of your application is not the bottleneck then this will also only serve to reduce the quality of the graphics in your application without improving performance.

In reality, it is only once the limiting factor of an application is determined by profiling with the correct tools that optimization work should be applied. It is also important to realise that once work has been done then the application requires re-profiling in order to determine whether the work actually improved performance and whether the bottleneck is still in the same stage of the graphics pipeline. It may be that the limiting stage in rendering is now in a different place and further optimization should be targeted accordingly.

3. The Golden Rules

As mentioned in Section 2 Optimal Development Approach, developers should seek to implement and observe as many of the techniques and principles mentioned in these rules as possible, right from the start of development in order to produce well-behaved, high performance graphics applications.

3.1. Understand Your Target Device

No two mobile devices are identical and no two graphics architectures are the same.

Even when the GPU architecture being targeted is thoroughly understood it is important to remember that no two System-on-Chips (SoCs) have exactly the same capability; some may have more powerful CPUs, others may have greater availability of bandwidth etc.

Even between two otherwise identical devices, as SoCs share memory between components, it is possible for applications to be slowed down by other applications being run in the background, especially in regards to memory bandwidth.

The Tile Based Deferred Rendering (TBDR) system used in PowerVR hardware already helps to relieve some of these issues by ensuring memory access is kept to a minimum but it is still vitally important that developers take these factors into account. Finally, in order to sensibly benchmark and recognise what bottlenecks may exist, a developer must possess, as has already been stated, a thorough understanding of the architecture being targeted; to this end an 'Architecture Guide for Developers' is provided in the PowerVR Insider SDK and through the Developer section of the Imagination website.

Developers should seek to learn as much information about their target platforms and where they may differ as possible. The manufacturers' websites for devices may be a good place to look for specifications and they may also provide developer community resources that can be helpful.

3.2. The Principle of 'Good Enough'

Real time graphics can only achieve a finite level of image fidelity per frame whilst maintaining a reasonable frame rate – the more complex a frame the slower it can be rendered. This means that a compromise has to be made between image quality or "correctness" and speed of rendering. While making this compromise it is important to remember that rendering is only as valuable as the quality perceived by the viewer and will always be an approximation - favour techniques that help improve the perceived quality rather than use idealistic, "correct" approaches that may slow rendering unacceptably or simply not look as good as simpler solutions. In other words, favour techniques that are "Good Enough" in order to retain acceptable rendering speed.

3.2.1. Examples

Alpha Blended Polygon Rendering Order

Many blend modes used when rendering transparent objects in a scene are submission order-dependent in their output. To get correct, consistent output the fragments rendered using these modes should be drawn in depth order, back to front. Unfortunately, sorting these transparent layers per-fragment is difficult and prohibitively expensive. Developers solve this issue by using "good enough" compromises, such as:

- Sort per polygon; very expensive and problematic
- Split objects into sub-meshes and sort; still expensive and introduces some artefacts
- Sort by object; cheap, but artefacts are easy to uncover

Interpolated Vertex Values vs. Per Fragment Calculations

Many techniques, such as bump-mapping, may be more accurate with per-fragment calculation, but in practice can look entirely acceptable using the results of per-vertex calculations that are interpolated across the fragments in a polygon. This approach, due to there being fewer vertices than fragments in a typical scene can be more efficient for "good enough" results.

Lower Bit-rate Textures

Textures that are compressed or down-sampled can look noticeably different to the developer, particularly in the preview window of a tool such as PowerVR's PVRTexTool GUI application. In a 3D scene this difference often is a lot less obvious, to the extent that a user will not perceive any benefit from the higher bitrate, more expensive versions of the textures. The smaller textures are usually "good enough" for acceptable image quality while making a much more significant difference to bandwidth use and hence frame rate.

3.3. Profile, Profile, Profile!

Profiling tools are provided to developers so they can gain an understanding of what is happening in their application, and how it relates to the hardware the application is running on. They allow developers to determine where bottlenecks are occurring, and enable effort to be concentrated on areas that will improve rendering performance.

Ideally, optimizations should only be targeted at bottlenecks that have been identified through profiling; these optimizations should then be checked by re-profiling. This cyclical process, as described in Section 2 Optimal Development Approach, ensures applications don't fall into the trap of cutting quality for no gain, for example, halving the number of vertices in a scene which is limited by the complexity of the fragment shader. To this end the PowerVR Insider SDK includes a GPU profiling tool called PVRTune; information on identifying bottlenecks with PVRTune can be found in the PVRTune User Manual.

3.4. Promote Calculations up 'The Chain'

In general, fewer vertices exist in a scene than fragments appear on the screen; as such, processing time can be saved by performing a calculation in the vertex shader rather than in the fragment shader. This is promoting a calculation 'up the chain'.

Even more GPU performance can be gained by promoting a calculation off the GPU altogether; for example, pre-building a matrix or pre-transforming an object into view space on the CPU. While the GPU can perform these operations very rapidly, in many cases far more rapidly than the CPU, performing a calculation once on the CPU is much less intensive than performing the operation once per vertex, or once per fragment.

'Pre-baking' takes this concept one step further. It is likely that a fragment shader is already reading from several textures, certain features can be pre-baked into these textures to save time; lighting, for example. This concept can be taken one step further again using the concept of 'lookup textures'. In cases where there is memory bandwidth available a texture can be created to act as a lookup table for a particular mathematical function; instead of calculating the solution to the function a texture read is performed. This saves calculation time at the expense of bandwidth, and possibly, dependent texture reads.

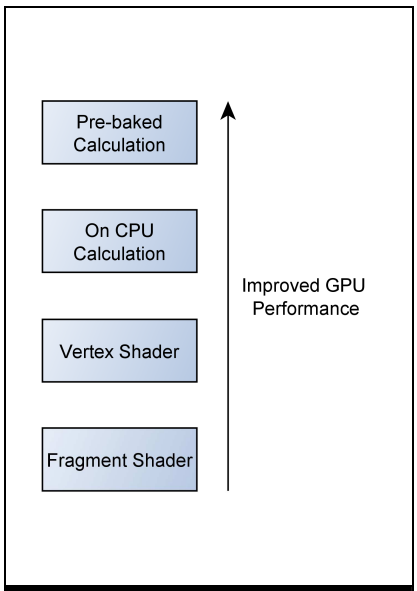

Figure 3-1 Generalisation of 'The Chain'

3.5. Avoid Accessing Render Targets

Accessing render targets from the CPU is very bad for GPU performance as it breaks the parallelism between the CPU and the GPU:

- Any access to the current render target will cause the driver to flush queued rendering commands and wait for rendering to finish.
- The CPU must wait for the flush to complete before it can access the buffer in question
- The GPU must wait for the CPU to finish its access before it can continue working in that buffer and may need to wait for further graphics instructions to be submitted from the CPU.

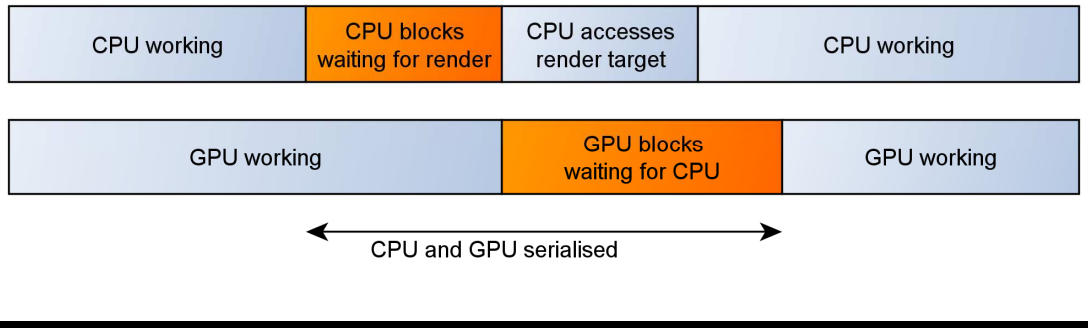

Figure 3-2 Serialised Render Target Access

3.5.1. Accessing Render Targets Safely

If render target access cannot be avoided then steps must be taken to ensure that CPU/GPU parallelisation is unbroken or at least that the negative effects are minimized. The recommended means of doing this is through the use of the EGL_KHR_fence_sync extension and EGLImage surfaces.

For details of these extensions please see their specifications, available on the Khronos website:

- http://www.khronos.org/registry/egl/

EGL_KHR_fence_sync

The EGL_KHR_fence_sync extension introduces the concept of 'sync objects' into EGL. Sync objects are a synchronisation primitive representing events whose completion can be tested for or waited upon.

By inserting a sync object just before 'eglSwapBuffers' is called, it is possible to wait on that 'fence' being hit as a means to determine if the GPU has finished writing to a target. With the addition of a queue of EGLImage surfaces, the CPU and GPU can both access a different render target without blocking each other. This is done using the OES_EGL_image_external extension, an extension designed to grant access to an area of memory shared between Khronos APIs (e.g. OpenGL ES) and the developer's application. An example use case for this would be the writing of video data from an application to a buffer to be used by the OpenGL ES API, or the retrieval of the contents of a framebuffer after writing to it has been completed.

This method should only be performed with EGLImage surfaces, all other methods, particularly those that require (ab)using 'glReadPixels' to force the GPU to finish, will always block and break parallelism as illustrated in Figure 3-2 Serialised Render Target Access.

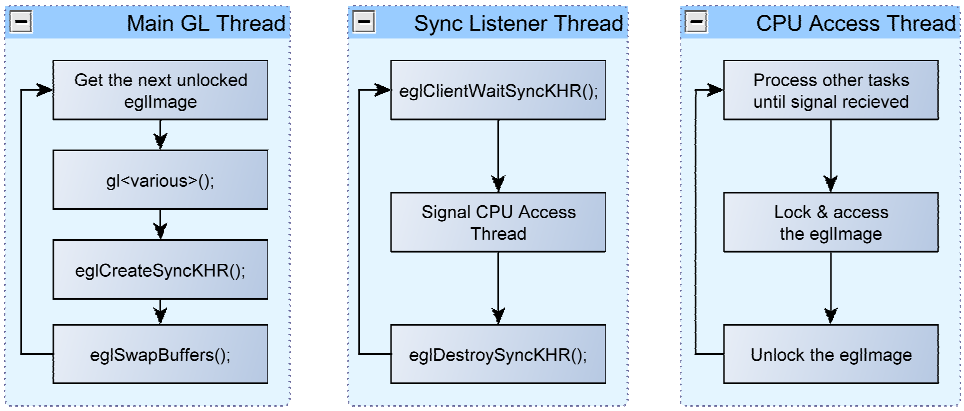

Figure 3-3 Implementation of Render Target Access using EGL_KHR_fence_sync

3.6. Avoid Using Alpha Test/Discard

The OpenGL ES 2.0 fragment shader operation `'discard'` can be used to stop fragment processing and prevent any buffer updates for the current fragment in a shader. Essentially, it provides the same functionality as the fixed function 'alpha test' but in a programmable manner. It can seem like a convenient method to achieve the rendering of complex shapes without using geometry, but it is an expensive operation on all modern graphics hardware and thus is discouraged.

On modern graphics hardware, the use of 'alpha test' requires that the fragment shader be run for a given fragment before visibility can accurately be determined. This affects performance on PowerVR as the visibility information must be fed back from the fragment processing stage to the ISP before the ISP can continue to perform depth and stencil tests for other polygons in that position. This effectively removes some of the benefits of PowerVR's Hidden Surface Removal (and those of 'Early-Z' techniques on other architectures).

For this reason `'discard'` and 'alpha test' should be avoided whenever possible.

It should also be noted that if a shader contains the `'discard'` keyword then any object that shader is applied to will suffer the cost of alpha test, even if the keyword is inside a conditional block that the developer knows will not be hit for a draw call. The GPU cannot know the result of the conditional without executing the fragment shader and so has to assume that the `'discard'` keyword may be hit. The solution to this is to move the use of `'discard'` into a separate shader.

Under fixed function APIs it is essential that 'alpha test' be disabled for objects that do not require it. It is common for 'alpha test' to be switched on at the beginning of a scene and left on for the entire scene. This is strongly discouraged is it may severely harm performance.

The same visual effect as `'discard'` can often be achieved through the use of the correct Alpha Blend Mode, and setting the Alpha value to `'0'` where `'discard'` would be used.

If `'discard'` or 'alpha test' cannot be avoided then objects using these techniques should be submitted after all opaque geometry is submitted.

3.7. Use EXT_discard_framebuffer and Clear

A PowerVR core uses a TBDR architecture that allows much work (such as colour, depth or stencil operations) to be carried out on-chip, without accessing main memory. Commonly, an application doesn't require the information from a frame for a subsequent render so the bandwidth used in storing and retrieving this data is unnecessary.

The EXT_discard_framebuffer extension allows the developer to communicate to the GPU which buffers are not going to be required after a tile has been processed. Additionally, performing a clear operation before rendering a frame tells the GPU that it doesn't need to retrieve stored data before drawing a tile.

Without these calls the GPU cannot know to avoid the wasted loads and flushes in time, even if subsequent rendering will entirely obscure any data that was previously present.

Used together, these two operations can save a great deal of unnecessary bandwidth use.

3.8. Opaque, then Alpha Test, then Alpha Blend

Opaque objects should be submitted first, then alpha test objects (where this technique cannot be avoided), then alpha blended objects. In addition to lowering the number of state changes used in a render, this ordering also takes the greatest advantage of PowerVR's Hidden Surface Removal feature.

3.9. Batch, Batch, Batch!

3.9.1. Minimize State Changes

Good render state management is vitally important when maximum performance is required. Setting OpenGL ES state values multiple times between draw calls, or repeatedly setting the same value again and again adds driver overhead to an application; wherever possible code should be structured to avoid these redundant calls. Where this is hard to achieve, a copy of the current state can be kept in the application and a call only made if the old state and the new state are different.

Ideally, the following rules should be followed:

- Set every OpenGL state at most once between draw calls.
- Set only those states that affect the next draw call.
- Don't set states that already have the desired value.

It should also be noted that, thanks to PowerVR's order independent, pixel-perfect, Hidden Surface Removal, objects do not need to be ordered by depth; this allows an application to sort by render state instead, which improves batching and minimizes state changes further.

3.9.2. Group Meshes

If multiple meshes have static positions and orientations relative to one another, and could use the same render state, they should be combined into a single mesh; this will reduce the number of draw calls, and thus may increase performance.

3.9.3. Texture Atlases

A texture atlas is a single large texture that contains multiple subtextures. With correctly calculated UVs, individual areas in the texture atlas can be used like a separate texture. This approach minimizes the number of times that textures must be rebound and hence reduces the number of draw calls an application requires; effectively batching textures.

There are two common issues that must be considered before using a texture atlas, however. The first of these is when using MIP-maps. During MIP-map generation it is possible that texels from neighbouring areas within the texture atlas can blend into each other. Some solutions to this are to leave a large border around each area within the texture atlas or to place areas with similar borders next to each other.

The second issue is in the use of texture wrap modes: when performing wrapping, OpenGL's texture coordinates only wrap over the entire texture; there is no facility to wrap over a portion of a texture. Solutions to this problem: ensure enough redundancy in the atlas texture to avoid discrepancies, create a shader that will correct the UV mapping or don't use wrapping at all.

Information on the number of OpenGL calls an application is making, and the OpenGL render state etc., all of which relate to batching correctly, can be checked using the PVRTrace application, available as part of the PowerVR Insider SDK.

3.10. Perform Rough Culling

Hidden Surface Removal is a very powerful tool for reducing overdraw and provides significant gains both in terms of power consumption and performance; however, even with this feature, vertices must still be processed even if all fragments resultant from them are obscured; the cheapest triangle to draw, even with HSR, is the triangle that is not submitted. Where it is feasible to do so, rough culling should be performed on the CPU to avoid unnecessary geometry processing; objects that are not going to contribute to a frame should not be submitted.

A minimal approach to this is to cull objects that are behind the camera or otherwise outside the view frustum as these will be clipped away by the GPU anyway. If it's possible to determine whether an object is going to be occluded entirely by another then this will also take work away from the renderer.

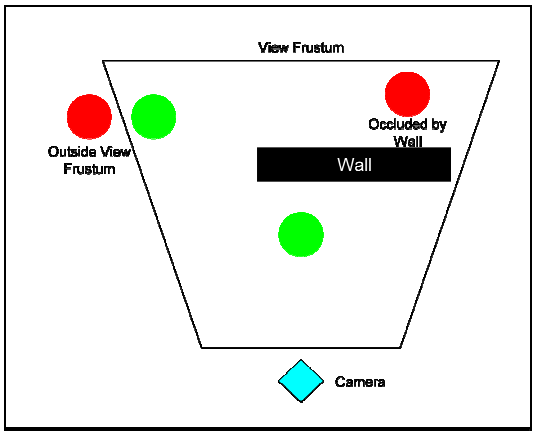

Figure 3-4 Rough Culling

3.11. Target a Sensible Frame Rate

A higher framerate will improve the smoothness of any animations in an application and may improve the feel of user interaction. Forcing a lower frame rate than the maximum that the device is capable of will help with power consumption as less work needs to be done by the device to render frames over the same period of time.

As a rule, if animation is on screen and needs to be updated frequently, such as during game play, then aim to update the screen at a high, constant framerate. If no animation is visible, such as when displaying an idle menu screen then there is no reason to render the same image over and over - consider lowering the frequency of screen updates until user interaction is detected. This will allow the device to expend less power over the same time period for the same result.

If an application is having trouble maintaining a constant framerate, say it varies between 30 and 60fps, then it may be beneficial to restrict rendering to a constant 30fps i.e. the lower end of the varying range, as this is likely to look just as smooth to the user without expending power drawing extra frames that don't enhance the quality of output.

Even if an application can run at a constant high frame rate, but this high framerate doesn't enhance the user experience, then consider lowering the frame rate to reduce power consumption. Alternatively, this may be an opportunity to increase the quality of graphics in the application by increasing model complexity, texture detail, or shader sophistication.

3.12. Favour Stencil Operations

Stencil operations are carried out on-chip, per tile, and so are very cheap on PowerVR hardware. In almost every case a technique that could be performed using functions such as scissoring can also be performed using the stencil buffer. Preferring the use of the stencil buffer in these cases will almost always improve performance, possibly dramatically, but the following questions should be considered first.

3.12.1. Is the Stencil Operation Required?

It is inefficient to use unnecessary calls that won't affect the final output; thus, if an object to be drawn would fit entirely inside a stencil region then the stencil operation should not be used. This holds true for other clipping operations, such as scissor.

3.12.2. Where is the Performance Bottleneck?

As is covered in Section 2 Optimal Development Approach it is essential to verify that the code that you wish to change is actually affecting the rendering speed of the application. If there are very few stencil/scissor operations in your application then it is likely that more value will be gained from optimizing elsewhere. For instance, an application that is limited by overly complex fragment shaders is unlikely to benefit significantly from swapping scissor operations for stencil operations.

3.12.3. Re-Implementing Using Stencil Operations

As an example, a common use case would be the use of rectangles to restrict the area of the screen to be drawn to. This technique, normally performed with scissoring, can easily be solved using stencil operations and this may provide a substantial boost to rendering performance.

The procedure for implementing this technique is as follows:

- Clear the stencil buffer.
- Render rectangles to the stencil buffer only, each rectangle representing an area that is to be drawn to. Each rectangle should be given a unique value within the stencil buffer if overlapping is required. If no overlapping is required, or content will not spill over the edge of a stencil, then the same value may be used. It should be noted that submission order will affect overlap behaviour.
- Render content using the stencil test value corresponding to the desired rectangle so that fragment visibility is determined by what is stored in the stencil buffer.

If more unique values are still required than the stencil buffer can contain, then the render should be resolved up to the stage where the number of available stencil values has run out. The above sequence should then be performed again for the remains of the scene.

Finally, one of the bonuses of using stencil operations for this technique is that, unlike scissor methods, the stencils used can be of an arbitrary shape and size and are not just restricted to rectangles.

4. Optimizing Geometry

4.1. Geometry Complexity

It is important that an appropriate level of geometry complexity be used for each object or portion of an object. It is a waste to use a large number of polygons on an object that will never cover more than a small area of the screen. Likewise, it is a waste to use polygons for detail that will never be seen due to camera angle, or culling; or to use large amounts for objects that may be drawn with much fewer, such as spending hundreds of polygons drawing a single quad. Shader techniques such as bump mapping should be considered to minimize geometry complexity, but still maintain a high level of perceived detail.

4.2. Primitive Type

In general, drawing a mesh as a single, indexed, triangle list will ensure the best performance on PowerVR SGX hardware.

4.3. Data Types

Vertex shaders always expect attributes to be of the type ‘float’, this means that all types except ‘float’ will require a conversion. This conversion is performed in the USSE pipeline and costs a few additional cycles; thus the choice of attribute data type is a trade-off between shader cycles, bandwidth/storage requirements and precision. It is important that type conversion only be considered when a bottleneck has been identified that type conversion can solve, otherwise the increased cost of conversion may harm performance.

Precision requirements should be checked carefully, the ‘byte’ and ‘short’ types are often sufficient, even for position information. For example, scaled to a range of 10m the ‘short’ types give you a precision of 150 µm. Scaling and biasing those attribute values to fit a certain range can often be folded into other vertex shader calculations, e.g. multiplied into a transformation matrix.

4.3.1. ‘Fixed’ Data Types

The ‘fixed’ data type uses the same bandwidth as ‘float’, but requires additional format conversion cycles in the USSE pipeline, thus it should be avoided.

4.4. Interleaving Attributes

Two ways exist to store vertex data in memory, either the data is stored with all the information, position, normals etc., pertaining to a given vertex in a single block, followed by all the information pertaining to the next vertex, and so on; or the data can be stored in a series of arrays, each containing all the information of a particular type for each vertex, for example, an array of positions, an array of normals etc. The first of these two options is called ‘interleaving’.

In general data should be interleaved; this provides better cache efficiency, and thus better performance.

Two major caveats exist to this rule. Interleaving should not be used if several meshes are to share the same array of vertex attributes; in this case putting the instances of this attribute into their own array may result in better performance, and will save bandwidth and storage space due to there being less duplication.

Interleaving should also not be used if a single attribute will be updated frequently, outside of the GPU, while the other attributes remain the same.

4.5. Vertex Buffer Objects

Vertex Buffer Objects (VBO) (and, where available, Vertex Array Objects) are the preferred way of storing vertex and index data; since VBO storage is managed by the driver there is no need to copy an array from the client side at every draw call and the driver is able to perform some transparent optimizations.

Pack all the vertex attributes that are required for a mesh into the same VBO unless a mixture of static and dynamic attributes are being used. Do not create a VBO for every mesh, it is a good idea to group meshes that are always rendered together in order to minimize buffer rebinding, this also has the benefit of improving batching.

For dynamic vertex data one buffer object should be used for each update frequency, and the right usage flag should be set (`STATIC_DRAW`, `DYNAMIC_DRAW`, `STREAM_DRAW`) when submitting data or allocating storage with `glBufferData`.

4.6. Padding

When vertex data is interleaved, each vertex should be aligned to a four byte boundary; when vertex data is not interleaved each element in each array of vertex data should be aligned to a four byte boundary.

5. Optimizing Textures

5.1. Texture Size

It is a common misconception that bigger textures always look better; a 1024x1024 texture that never takes up more than a 32x32 area of the screen is a waste of both storage space and bandwidth. A texture's size should be based on its usage; ideally this will be the same as the number of texels the texture will cover when the object that it is mapped to is viewed from the closest allowable distance.

5.2. Texture Compression

Modern applications have become graphically intensive; certain types of software, such as games or navigation aids, often need large amounts of textures in order to represent a scene with satisfying quality. Texture compression can save or allow better utilization of bandwidth, power, and memory without noticeably losing graphical quality and should be used as much as possible. PowerVR hardware offers a specific form of texture compression called 'PVRTC' which should be used as much as possible.

PVRTC is PowerVR's proprietary texture compression scheme. It uses a sophisticated amplitude modulation scheme to compress textures: texture data is encoded as two low-resolution images along with a full resolution, low bit-precision modulation signal. More information can be found in the whitepaper:

Fenney, S. (2003) 'Texture Compression Using Low-Frequency Signal Modulation' *SIGGRAPH Conference*.

Additionally, it supports both opaque (RGB) and translucent (RGBA) textures (unlike other formats, such as S3TC, that require a dedicated, larger form to support full alpha channels); and boasts a very high image quality for competitive compression ratios: 4 bits per pixel (PVRTC 4bpp) and 2 bits per pixel (PVRTC 2bpp). At time of writing, no other format is available in hardware at such a low bit rate.

5.2.1. Why use PVRTC?

In any given situation, the best texture format to use is the one that gives the required image quality at the highest rate of compression. The smaller the size of the texture data, the less bandwidth is required for texture fetches; this reduces power consumption, can increase performance, and allows for more textures to be used for the same budget. The smallest RGB and RGBA format currently available is PVRTC 2bpp; as such, it should be considered for every texture in an application. Larger formats (such as PVRTC 4bpp) should only be used if the image quality provided by a particular PVRTC 2bpp image does not have sufficient quality.

Storage Footprint vs. Memory Footprint

PVRTC compression reduces the memory footprint of a given texture; this allows applications to fit all their required textures in a constrained amount of texture memory, or to use larger (or more) textures for the same memory budget at, potentially, extra quality. In addition, any savings in memory requirements are very useful for mobile and tablet devices where memory is shared across an entire SoC (System on Chip).

Performance Improvement

The smaller memory footprint of PVRTC means less data is transferred from memory to the GPU allowing for major bandwidth savings. In situations where memory bandwidth is the limiting factor in an application's performance PVRTC can provide a significant boost.

Power Consumption

Memory accesses are one of the primary causes of increased power consumption on mobile devices where battery life is of the upmost importance. The bandwidth savings and better cache performance resulting from the use of PVRTC both contribute to decreasing the quantity and magnitude of memory accesses; which in turn reduce the power consumption of an application.

5.2.2. Image File Compression versus Texture Compression

Developers are familiar with compressed image file formats such as jpg or png; it is important to be aware of the distinction between these forms of ‘storage’ compression and the texture compression discussed in this document.

The primary requirement of ‘storage’ compression schemes is that files compressed using them should occupy as small an amount of storage in a file system as possible; there is no requirement that the data stay compressed while in use. The result is that ‘storage-based’ image file formats tend to produce very small file sizes, often for very high (or lossless) image quality, but at the cost of immediate decompression on use. This immediate decompression, usually to 24/32bpp means that the image, while small on disk, consumes large amounts of bandwidth and memory at runtime.

Texture compression schemes, such as PVRTC are designed to be directly usable by the GPU. The texture data exists in storage, in memory, and when transferred to the graphics hardware itself, in the compressed format. The only step in which full-precision colour values are extracted from a compressed state is when dedicated texture sampling hardware inside the graphics accelerator passes texel values to the shader processing units. A graphical representation of this can be seen in Figure 5-1 Image File Compression vs. Texture Compression.

This allows all the advantages mentioned in Section 5.2.1 above, but puts some limits on the form the compression technique may take. In order to allow for direct use by the graphics accelerator a texture format should be optimized for random access, with a minimal size of data from which to retrieve each texel’s values. Consequently, texture compression schemes are usually fixed bitrate with very high data locality; image file formats are not constrained by these requirements and thus can often achieve higher compression ratios and image quality for a given data size.

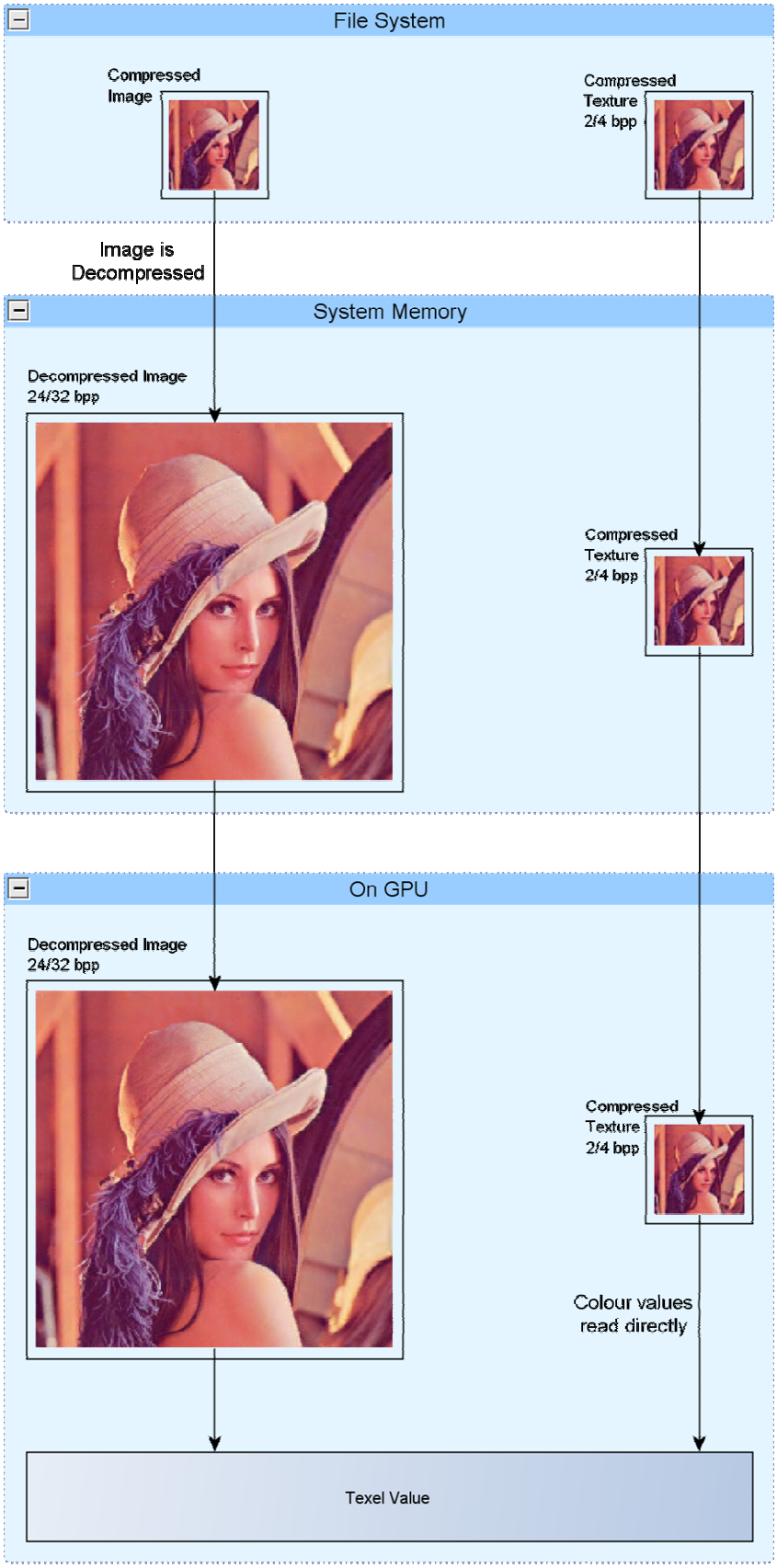

Figure 5-1 Image File Compression vs. Texture Compression

5.3. MIP-Mapping

MIP-Maps 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 MIP-Maps, the GPU can be set up to automatically calculate which level-of-detail comes closest to mapping the texels of a MIP-Map to pixels in the render target, and use the according MIP-Map for texturing.

Advantages

Using MIP-Maps has two important advantages; it increases performance by massively improving texture cache efficiency, especially in cases of strong minification; it also improves image quality by countering the aliasing that is caused by the under-filtering of textures when MIP-Mapping. The single drawback of MIP-Mapping is that it requires approximately 1/3 more texture memory per image; depending on the situation, this cost may be minor when compared to the benefits to rendering speed and image quality.

There are some exceptions where MIP-Maps should not be used. Specifically, MIP-Mapping should not be used where filtering cannot be applied sensibly, such as for textures that contain non-image data such as indices, or depth textures; it should also be avoided for textures that are never minified, for example, UI elements where texels are always mapped one-to-one to pixels.

Generation

Ideally MIP-Maps should be created offline using a tool like PVRTexTool (available as part of the PowerVR Insider SDK). It is, however, possible to generate MIP-Maps at runtime using the function ‘glGenerateMipmap’ and this can be useful for updating the MIP-Maps for a render to texture target; this will not work, however, with PVRTC textures which must have their MIP-Maps 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 of ‘glGenerateMipmap’.

Filtering

Finally, it should be noted that the lack of filtering between MIP-Map levels can lead to visible seams at MIP-Map transitions, a form of artifacting called ‘MIP-Map banding’; ‘Trilinear Filtering’, using the filter mode ‘GL_LINEAR_MIPMAP_LINEAR’, can effectively eliminate these seams, for a price (see Section 5.4.1 Texture Filtering), and thus achieve an even higher image quality.

5.4. Texture Sampling

5.4.1. Texture Filtering

'Texture Filtering' is used to increase the image quality of textures used in 3D scenes; however, it comes at a cost. Filtering works by taking multiple texture fetch values and combining them in order to produce as good a sampling value as possible to use in fragment calculations. Retrieving multiple values requires more data to be fetched, possibly from disparate areas of memory and so cache performance and bandwidth use can be affected. For instance, whenever two MIP-Map levels must be blended together for tri-linear filtering, the texture unit in the GPU must spend time and bandwidth fetching and filtering the required data from the two MIP-map levels in question; this can cause the processing of a fragment to stall while the data is fetched and adds to the total amount of memory that must be transferred across the bus in order to render a frame.

For independent texture reads, texture sampling can begin before the execution of a shader and so the latency of the texture fetch can be avoided. For dependent reads the cost can further be amortised thanks to the hardware scheduler in PowerVR GPUs; particularly if the shader in question involves a lot of mathematical calculation. This latency can be hidden by swapping in another thread on the GPU; this thread will process as much as possible with the original thread being swapped back once the fetch is complete. Further information on the functioning of the 'Coarse Grain Scheduler' and thread scheduling within PowerVR hardware can be found in 'SGX Architecture Guide for Developers' available on the Imagination website.

The three main techniques for 'texture filtering' are bilinear, trilinear, and anisotropic; each giving increased image quality than the previous, at an increasing cost. Performance can be gained by using an appropriate level of filtering, following The Principle of 'Good Enough', don't use anisotropic if trilinear is acceptable, don't use trilinear if bilinear is acceptable, don't filter at all – choose nearest or point texture sampling - if it's not necessary.

5.4.2. Dependent Texture Reads

A 'dependent texture read' is a texture read in which the texture coordinates depend on some calculation within the shader instead of on a 'varying'. As the values of this calculation cannot be known ahead of time it is not possible to pre-fetch texture data and so stalls in shader processing occur.

Vertex shader texture lookups always count as dependent texture reads, as do texture reads based on the '.zw' channels of a varying.

The cost associated with a dependent texture read can be amortised to some extent by hardware thread scheduling, but they should still be avoided wherever possible for good performance.

5.4.3. Wide Floating Point Textures

For textures that exceed 32 bits per texel, each additional 32 bits is counted as a separate texture read; this also applies to half float texture with 3 or 4 components as well as float textures with 2 or more components. These larger formats should be avoided unless necessary for a particular effect.

5.5. Demystifying NPOT

If a 2D texture has dimensions which are a power-of-two (i.e width and height are 2^n and 2^m for some m and n), then the texture is said to be a POT texture (power-of-two); if they are not it is said to be an NPOT texture (non-power-of-two). This section seeks to clarify the status of NPOT textures on PowerVR SGX cores.

5.5.1. SGX Support

NPOT textures are supported as required by the OpenGL ES specifications. However:

- NPOT textures are not supported in OpenGL ES 1.1 implementations
- NPOT textures are supported in OpenGL ES 2.0 implementations, but only with the wrap mode of GL_CLAMP_TO_EDGE.
    - The default wrap mode in OpenGL ES 2.0 is GL_REPEAT; this must be specifically overridden in an application to GL_CLAMP_TO_EDGE for NPOT textures to function correctly.
    - If this wrap mode is not correctly set then an 'invalid texture' error will occur, likewise a driver error may occur at runtime, on newer drivers, to highlight the need to set a wrap mode.

5.5.2. GL_IMG_texture_npot

An extension exists (GL_IMG_texture_npot) to provide some of the functionality found outside of the core OpenGL ES specification. This extension allows the use of the following filters for NPOT textures:

- LINEAR_MIPMAP_NEAREST
- LINEAR_MIPMAP_LINEAR
- NEAREST_MIPMAP_NEAREST
- NEAREST_MIPMAP_LINEAR

It also allows the calling of 'glGenerateMipmapOES' with an NPOT texture to generate NPOT MIP-maps.

Like all other OpenGL extensions, the application should check for this extension's presence before attempting to load and use it.

5.5.3. Guidelines

Finally, a few additional points should be considered when using NPOT textures:

- POT textures should be favoured over NPOT textures for the majority of use cases as this gives the best opportunity for the hardware and driver to work optimally.
- A 512x128 texture will qualify as a POT texture, not an NPOT texture; rectangular POT textures are fully supported.
- 2D applications (such as a browser or other application rendering UI elements where an NPOT texture is displayed with a one-to-one texel to pixel mapping) should see little performance loss from the use of NPOT textures other than possibly at upload time.
- To ensure that texture upload can be optimally performed by the hardware, use textures where both dimensions are multiples of 32 pixels.
- The use of NPOT textures may cause a drop in performance during 3D rendering. This can vary depending upon MIP-map levels, size of the texture, the texture's usage and the target platform.

5.6. Texture Uploading

When a texture is uploaded through the use of 'glTexImage2D' the input data is usually in linear scanline format; internally PowerVR hardware uses its own layout to improve memory access locality and improve cache efficiency. Reformatting of the data is done on chip by dedicated hardware and thus is very fast, however, it is still recommended that a few steps be taken to minimize the cost of this reformat.

- Textures should be uploaded during non-performance critical periods, such as initialisation; this helps avoid the framerate dips associated with additional texture loading.
- Avoid uploading texture data mid-frame to a texture object that has already been used that frame.
- Consider performing a 'warm-up' step after texture uploads have been performed, once again, this helps avoid the framerate dips associated with texture loading

5.6.1. Texture Warm-up

The 'warm-up' step mentioned above ensures that textures are fully uploaded immediately. By default, 'glTexImage2D' does not perform all the processing required to upload immediately; instead the texture is fully uploaded the first time it is used. It is possible to force an upload by drawing a series of triangles off screen or otherwise obscured with the texture object in question bound and so marked for use. Performing this for all textures in a scene will avoid the cost and potential stutters when they are uploaded on first use.

5.6.2. Texture Formats and Precision

As stated in Section 6.4.7 Samplers**Error! Reference source not found.**, in general, textures should be read at 'lowp'. The exceptions to this are half float textures which should be read as 'mediump', and float and depth textures which should be read as 'highp'.

5.7. Render To Texture

The preferred method for rendering to textures on OpenGL ES 2.0 is through the use of Frame Buffer Objects (FBO) with textures as attachments. The only situation where FBOs are not recommended is when accessing render targets from the CPU, for further information on this please see Section 3.5.1 Accessing Render Targets Safely.

For maximised performance, FBOs should be rendered to in series, submitting all calls for one FBO before moving to the next. This serves to minimize state changes, as well as reducing unnecessary memory bandwidth usage caused by flushing partially completed renders when the target FBO is changed. Re-using of FBO targets should be approached with caution – if the previous contents of an FBO is required for some stage of a render that hasn't been completed then this contents will need to be preserved in an expensive copy operation. Due to this, a technique that may seem to be saving memory could actually be using substantially more.

5.8. Mathematical Look-ups

Sometimes it can be a good idea to encode the results of a complex function into a texture and use it as a look-up table instead of performing the calculations in a shader. However, this will only provide a boost in performance if a bottleneck has been identified in the processing of the shader in question, and bandwidth is free to perform the texture lookup. If the function parameters (and thus the texture co-ordinates in the lookup table) vary wildly between adjacent fragments then cache efficiency will suffer; this can be mitigated with the use of MIP-Maps but at the cost of accuracy. Profiling should be performed to determine if the results of using look-up tables are acceptable, as well as to determine what, if any, effect their use will have on performance.

6. Optimizing Shaders

6.1. Choose the Right Algorithm

For complex shaders that run for more than a few cycles, picking the right algorithm is usually more important than low-level optimizations. It is highly recommended that a fast, well designed, algorithm be favoured over small performance tweaks to a poor algorithm. Bear in mind, that, although increasingly powerful, mobile graphics hardware is not designed to handle some of the latest techniques in desktop and console shaders. As such, a reduction in complexity will likely be needed from some of these techniques for mobile shader implementations.

6.2. Know Your Spaces

A common mistake in vertex shaders is to perform unnecessary transformations between model space, world space, view space and clip space. If the model-world transformation is a rigid body transformation, i.e. it only consists of rotations, translations, and mirroring, lighting and similar calculations can be performed directly in model space. Transforming uniforms such as light positions and directions to model space is a per-mesh operation, as opposed to transforming the vertex position to world or view space once per vertex and so is an optimization. In cases where a particular space must be used, e.g. for cube map reflections, it's often best to use this single space throughout.

6.3. Flow Control

PowerVR hardware offers full support for flow control in both vertex and fragment shaders without the need to explicitly enable an extension.

When conditional execution depends on the value of a uniform variable, this is called 'static flow control', and the same shader execution path is applied to all vertex or fragment instances in a draw call. 'Dynamic flow control' refers to conditional execution based on per-fragment or per-vertex data, e.g. textures or vertex attributes.

'Static flow control' can be used to combine many shaders into one big 'uber-shader'. Thorough profiling should be done when taking this approach however as a performance advantage may not be gained. Beware of conditional branches that use Alpha Test/Discard (see 3.6 Avoid Using Alpha Test/Discard).

Using dynamic branching in a shader has a non-constant overhead that depends on the exact shader code; dynamic branching is, therefore, unpredictable in its effect on performance. One thing to note is that the branching granularity on PowerVR SGX hardware is one fragment or one vertex; this means areas of fragments don't have to be coherent in terms of branching.

In general, the following rules should be applied:

- Make use of conditionals to skip unnecessary operations when the condition is met in a significant number of cases.
- If the product of two complex functions is required, and that product sometimes evaluates to zero, use the less complex function, or the function that most often returns zero, as a condition for executing the other.

6.4. Demystifying Precision

Unlike other graphics architectures, PowerVR SGX is designed with support for the multiple precision features of graphics APIs such as OpenGL ES 2.0. Three precision modifiers are included in the API spec for OpenGL ES 2.0, ‘mediump’, ‘highp’, and ‘lowp’; lower precision calculations can be performed faster, but need to be used carefully to avoid troubles with visible artefacts being introduced. The safest method of arriving at the right precision for a given value is to begin with ‘highp’ for everything (except samplers) then reduce the precision of specific variables to as low as possible while checking that the visual output is still as expected. The following guidelines may help in this:

6.4.1. Highp

Float variables with the ‘highp’ precision modifier will be represented as 32 bit floating point values; this precision should be used for all vertex position calculations, including world, view, and projection matrices, as well as any bone matrices used for skinning. It should also be used for most texture coordinate and lighting calculations (though in some cases ‘mediump’ may be sufficient), as well as any scalar calculations that use complex built-in functions such as ‘sin’, ‘cos’,’ pow’, ‘log’, etc.

6.4.2. Mediump

Variables declared with the ‘mediump’ modifier are represented as 16 bit floating point values covering the range [65520, -65520]. This precision level typically offers only minor performance improvements over ‘highp’, with throughput being identical most of the time. It can, however, reduce storage space requirements and thus it can be very useful for texture coordinate varyings.

6.4.3. Lowp

A variable declared with the ‘lowp’ modifier will use a 10 bit fixed point format, allowing values in the range [-2, 2] to be represented to a precision of 1 / 256. This precision is useful for representing colours and any data read from low precision textures, such as normals from a normal map. Care must be taken not to overflow the maximum or minimum value of ‘lowp’ precision, especially with intermediate results.

Swizzling

Swizzling is the act of accessing or re-ordering the components of a vector out of order. Some example of swizzling can be found below:

```
a = var.brg;                          // Swizzled – Out of order access
b = vec3(var.g, var.b, var.r);        // Swizzled – Out of order access
c = vec3(vec4);                       // Not swizzled – Dropping a component does not change
                                      // access order
d.gr = a.gr + b.gr                    // Not swizzled – This will be optimized to a
                                      // non-swizzled form
```

Swizzling costs performance when performed on ‘lowp’ variables due to the additional work required to move vector components when they in ‘lowp’ form, and thus should be avoided.

6.4.4. Conversion Costs

When performing arithmetic on multiple precisions within the same calculation it is likely that values will have to be ‘packed’ or ‘unpacked’. ‘Packing’ is the act of taking a higher precision value and placing into a lower precision variable; ‘unpacking’ is the reverse, taking a lower precision value and placing it into a higher precision variable. Given the following example:

```
mediump float out;
lowp float x;
highp float y;
out = x * y;
```

Potentially, the ‘lowp’ value ‘x’ will be ‘unpacked’ to a ‘highp’ and then the resulting multiplication will be ‘packed’ back into the ‘mediump’ variable ‘out’. This one line of code may end up containing two additional instructions. Where possible precisions should be kept the same for an entire calculation as each ‘pack’ and ‘unpack’ has a cost associated with it.

This means that simply reducing the precision of variables in shaders does not necessarily reduce their processing cost – it is essential that tools such as PowerVR's PVRUniSCoEditor are used to analyse shaders for optimal processing.

6.4.5. Attributes

The per-vertex 'attributes' passed to a vertex shader should use a precision appropriate to the data-type being passed in, so, for example, 'highp' would be unrequired for a float whose maximum value never goes above 2 and for which a precision of 1/256 would be acceptable.

6.4.6. Varyings

Varyings' represent the outputs from the vertex shader which are interpolated across a triangle and then fed into the fragment shader. Each 'varying' requires additional space in the parameter buffer, and additional processing time to perform interpolation; to keep this to a minimum as few a number of 'varyings' as possible should be used. One caveat to this exists, as PowerVR SGX hardware performs 'highp' mathematics on scalar values, when using 'highp' multiple varyings should not be packed into a single vector. This becomes an even worse choice when dealing with texture co-ordinate varyings; texture co-ordinate varyings, regardless of precision, which are packed into the '.zw' channel of a 'vec4' will always be treated as undesirable dependent texture reads (see 5.4.2. Dependent Texture Reads).

6.4.7. Samplers

'Samplers' are used to sample from a texture bound to a certain texture unit. The default precision for sampler variables is 'lowp', and generally this is good enough. Two main exceptions exist to the 'lowp' rule, if the sampler will be used to read from either a depth or float texture then it should be declared with 'highp'; if the sampler will be used to read from a half float texture then it should be declared as 'mediump'.

6.4.8. Uniforms

Uniform variables represent values that are constant for all vertices or fragments processed as part of a draw call. Similar to redundant state changes, redundant uniform updates in between draw calls should be avoided. Unlike attributes and varyings, uniform variables can be declared as arrays, however, care should be taken when using uniform arrays; while a certain number of uniforms can be stored in registers on-chip, large uniform arrays will be stored in memory and accessing them comes at a bandwidth and execution time cost.

Uniform Calculations

The PowerVR shader compiler is able to extract calculations based on uniforms from the shader and perform these calculations once per draw call. If this functionality is desired, it is important that the order of operations is chosen so that the uniforms are processed first, such as in the example below.

```
uniform highp mat4 modelview, projection;
attribute vec4 modelPosition;

// Can be extracted
gl_Position = (projection * modelview) * modelPosition;

// Can not be extracted
gl_Position = projection * (modelview * modelPosition);
```

6.5. Scalar Operations

When using 'highp' precision PowerVR SGX hardware operates on scalar not vector values; this allows for greater efficiency, but different optimization rules apply.

Vectorising computations that are naturally scalar should be avoided, likewise when mixing scalar and vector calculations, care should be taken to keep the calculation scalar for as long as possible. For example:

```
highp vec4 v1, v2;
highp float x, y;

// Bad
v2 = (v1 * x) * y;

// Good
v2 = v1 * (x * y);
```

6.6. Sparse Matrices

If it is already known that many elements of a transformation matrix are zero, don't perform a full matrix transform.

For example, given a typical projection matrix in the form of:

A	0	0	0
0	*B*	0	0
0	0	*C*	*D*
0	0	*E*	0

If the vertex being transformed is already in view space an additional full transformation would be both a waste of cycles and unnecessary since dividing the matrix by a positive constant will not change the transformation result in homogeneous co-ordinates. In this case it is sufficient to store just four values. Similarly, non-projective transformation matrices usually have the fourth row fixed at (0, 0, 0, 1); if this holds true then it is possible to store the matrix as three 'vec4' rows, replacing the matrix-vector multiplication with three dot products as shown below:

```
attribute highp vec3 vertexPos;
uniform highp vec4 modelview[3]; // first three rows of modelview matrix
uniform highp vec4 projection;   // = vec4(A/D, B/D, C/D, E/D)

void main()
{
	// transform from model space to view space
	highp vec3 viewSpacePos;
	viewSpacePos.x = dot(modelview[0], vec4(vertexPos, 1.));
	viewSpacePos.y = dot(modelview[1], vec4(vertexPos, 1.));
	viewSpacePos.z = dot(modelview[2], vec4(vertexPos, 1.));

	// use view space position in calculations
	...

		// transform from view space to clip space
		gl_Position = viewSpacePos.xyzz * projection;
	gl_Position.z += 1.0;
}
```

7. Related Materials

Software

- PVRTune
- PVRTrace
- PVRTexTool

Documentation

- PVRTune User Manual
- PVRTrace User Manual
- PVRTexTool User Manual
- PVRTC & Texture Compression User Guide
- PowerVR Architecture Guide for Developers

8. Contact Details

For further support contact:

devtech@imgtec.com

PowerVR Developer Technology
Imagination Technologies Ltd.
Home Park Estate
Kings Langley
Herts, WD4 8LZ
United Kingdom

Tel: +44 (0) 1923 260511
Fax: +44 (0) 1923 277463

Alternatively, you can use the PowerVR Insider forums:

www.imgtec.com/forum

For more information about PowerVR or Imagination Technologies Ltd. visit our web pages at:

www.imgtec.com