# PowerVR

# Performance Recommendations

| | | |
|---|---|---|
| Filename | : | PowerVR.Performance Recommendations |
| Version | : | PowerVR SDK REL_16.2@4266559a External Issue |
| Issue Date | : | 12 Oct 2016 |
| Author | : | Imagination Technologies Limited |

## Contents

# List of Figures

# 1. Introduction

PowerVR SGX and PowerVR Rogue are Graphics Core architectures from Imagination Technologies designed specifically for shader-based APIs like OpenGL ES 2.0 and 3.0. Due to their scalable architectures, the PowerVR 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 or PowerVR Rogue enabled device. Throughout the document, the specific recommendations for PowerVR SGX and PowerVR Rogue are marked as appropriate.

## 1.2. The Golden Rules

The golden rules are a set of more generic performance recommendations that developers should seek to implement and observe as many of the techniques and principles mentioned in these rules help to produce well-behaved, high performance graphics applications. These rules are detailed in the document entitled "PowerVR Performance Recommendations: The Golden Rules", which is supplied with the PowerVR SDK.

## 1.3. Optimal Development Approach

It is crucial to adopt the practices identified 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 near-final state, the process of iteration depicted in Figure 1 should be adopted. The main benefit of this approach is that time is not wasted and graphics quality is not comprised by making changes that do not benefit performance.

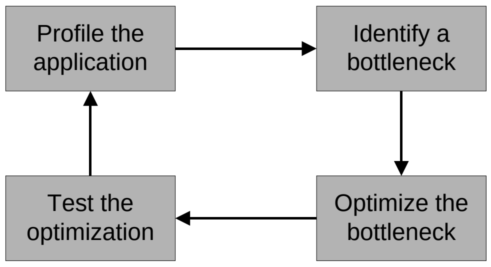


**Figure 1. Cyclical profiling**

## 1.4. 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 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 at the same stage of the graphics pipeline. It may be that the limiting stage in rendering is now at a different place and further optimization should be targeted accordingly.

## 1.5. Optimizing Graphics Applications for PowerVR GPUs

### 1.5.1. Know Your Target

Before diving into tools or performance recommendations, it is important to consider the capabilities and characteristics of the target device.

**Graphics Architecture**

A basic understanding of how API calls are processed by the driver, inserted into the graphics hardware's command stream and converted into coloured pixels goes a long way. It provides an immediate appreciation of why certain graphics API calls are costly and how submitted calls will map to the graphics hardware's processing pipeline.

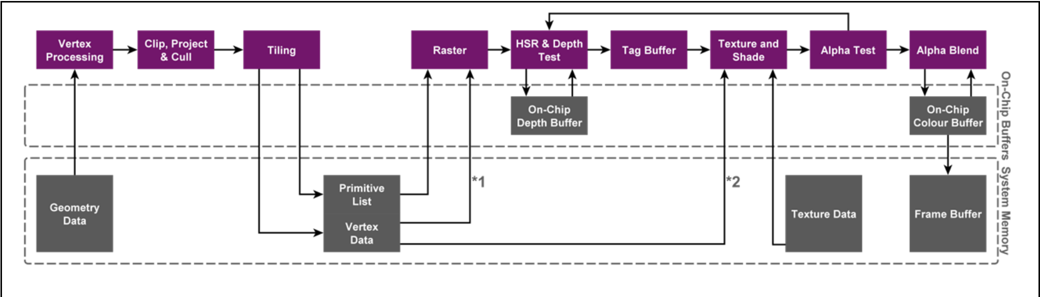


**Figure 2. PowerVR GPU TBDR Graphics Architecture**

It is recommended to read the following documents to familiarize yourself with the PowerVR graphics architecture:

- PowerVR Hardware Architecture Overview for Developers
- PowerVR Series5 Architecture Guide for Developers
- PowerVR Series6 Compiler Instruction Set Reference

These documents are packaged with the PowerVR SDK & Tools, and can also be found online at https://community.imgtec.com/developers/powervr/documentation/.

**Mobile Graphics APIs**

Mobile graphics APIs are a subset of their desktop counterparts, with imposed restrictions and specific features to suit the performance characteristics of mobile devices and the batteries that power them. Although the latest APIs, such as OpenGL ES 3.1 and the Android Extension Pack, have brought many of the desktop and console features to low power devices, there are still differences that need to be considered.

Many of the recommendations in this document (as well as those in the PowerVR Performance Recommendations: The Golden Rules and PowerVR Supported Extensions documents) apply to all mobile graphics architectures. Of course, these documents also detail PowerVR specific behaviour and describe OpenGL ES extensions exposed by the PowerVR reference driver for advanced hardware features.

## 1.5.2. Analysing an Application's Performance

**The Process of Optimizing Graphics Applications**

Optimizing graphics applications seems like a straightforward process. Although the steps in the diagram below may seem obvious, both beginners and experienced developers have in the past made simple mistakes that have cost large amounts of development time to resolve. Developers tend to run their analysis tools, identify a bottleneck, modify their application and consider the work done. One of the most important stages of optimisation though is to verify the change has actually improved performance. Without analysing performance after a modification, it's easy for new (and possibly worse) bottlenecks to creep their way into a renderer.

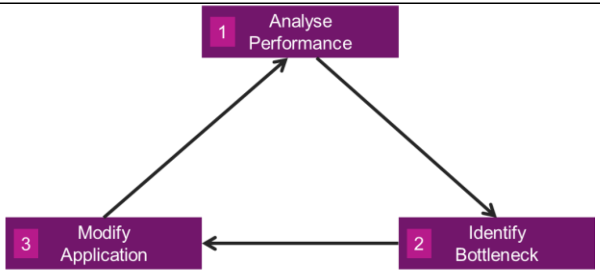


**Figure 3. Optimization Process for Graphics Applications**

**The Right Tools for the Job**

The PowerVR SDK includes profilers, debuggers and a variety of other analysis tools to help developers track down issues. Here's a quick overview of the key utilities:

- **PVRMonitor:** Renders a real-time overlay of CPU and GPU stats on Android devices integrating PowerVR GPUs.
- **PVRTune:** Remote GPU performance analyser (server on the target, GUI analysis tool on your development machine). Captures timing data and counters, such as hardware unit loads and throughput data, in real-time.
- **PVRShaderEditor:** Off-line shader editor & performance analyser. Generates disassembly in real-time for Series6 and Series6XT GPUs.
- **PVRTrace:** OpenGL ES 1.x, 2.0 & 3.x capture and analysis tool.

For more information regarding our entire suite of development tools, please visit our PowerVR Tools landing page at https://community.imgtec.com/developers/powervr/tools/.

# 2. Optimizing Geometry

## 2.1. 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. For example, 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. Techniques such as “Level of Detail” should be used. This is especially true for things such as reflection passes where higher amounts of geometry may not be visible.

## 2.2. Primitive Type

For optimal performance on PowerVR Graphics Cores, a mesh with static attribute data should:

- Use indexed triangle lists;
- Interleave VBO attribute data;
- Not include unused attributes

For optimal vertex shader execution performance, meshes transformed by the same vertex shader (even if compiled into different shader programs) must have the same VBO attribute data layout.

On some devices, padding each vertex to 16 byte boundaries may also improve performance.

## 2.3. 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 is considered as bandwidth is always at a premium.

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 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.

### 2.3.1. “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.

## 2.4. 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 as 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 Graphics Core, while the other attributes remain the same. In this instance, data that will not be updated should be interleaved, while data that will be updated is held in a separate array.

## 2.5. Vertex Buffer Objects

Vertex Buffer Objects (VBO) 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.

As the TBDR tends to process multiple frames at a time, the driver has to internally allocate multiple buffers for dynamic VBOs so that each frame has a unique dynamic buffer associated with it. Because dynamic VBOs cause the driver to behave in this way it is generally better for performance to resubmit vertex data that changes on a per-frame basis. If there is a mesh where only some of the vertex data is dynamic (for example, a skinned character in a game) then a VBO should be created that contains the static data and use calls to `glVertexAttribPointer()` to resubmit the dynamic vertex data. On a similar note, a VBO that will never change should always set `STATIC_DRAW` while a VBO whose contents will change should never set it.

# 3. Optimizing Textures

## 3.1. 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 reduces cache efficiency. A texture's size should be based on its usage; there should be a 1 pixel to 1 texel mapping when the object that it is mapped to is viewed from the closest allowable distance.

Before considering reducing the resolution of your texture assets to save storage space, you should apply texture compression. If the quality of the lossy texture compression is unacceptable, you can then consider using an 8 or 16 bit per pixel uncompressed format. If you still need to reduce the storage space of your assets, you should then consider reducing the resolution of your images.

## 3.2. 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.

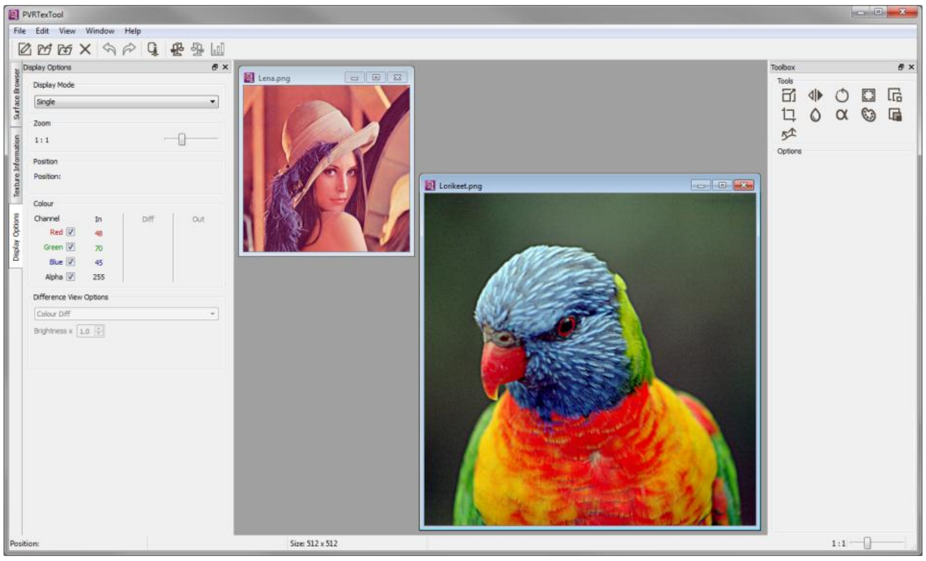
PVRTC is PowerVR’s proprietary texture compression scheme. It uses a sophisticated amplitude modulation scheme to compress textures: texture data is encoded as two low-resolution images along with a full resolution, low bit-precision modulation signal. More information can be found in the whitepaper:

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. Is also 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.

### 3.2.1. PVRTexTool

PVRTexTool (Figure 4) is a utility for compressing textures, which is an important technique that ensures the lowest possible texture memory overhead at application run-time. The PVRTexTool package includes a library, command-line and GUI tools, and a set of plug-ins. Plug-ins are available for Autodesk 3ds Max, Autodesk Maya, and Adobe Photoshop.



Display Mode: Single
Zoom: 1 : 1
Position:

| Channel | In | Diff | Out |
|---|---|---|---|
| Red ✓ | 48 | | |
| Green ✓ | 70 | | |
| Blue ✓ | 45 | | |
| Alpha ✓ | 255 | | |

Difference View Options: Colour Diff
Brightness x 1.0

Position:
Size: 512 x 512
1:1

Figure 4. PVRTexTool GUI

Each component is capable of converting to a variety of popular compressed texture formats such as PVRTC and ETC, as well as all of the core texture formats for a variety of different APIs. They also include a number of advanced features to pre-process the image data, for example, border generation, colour bleeding and normal map generation.

Textures can be saved to DDS, KTX, or PVR, which is Imagination's PowerVR Texture Container format which benefits from full public specification, support for custom metadata, as well as complete and optimized resource loading code in the PVRTools. Key features include:

- Supports all core texture formats in OpenGL ES and DirectX 11.1
- PVRTC, ETC and DXT texture compression
- Outputs to PVR, KTX, or DDS files
- Pre-processing textures for efficient rendering
- Normal map generation
- Composition and visualization of cube maps
- Optimized font to texture creation
- Creation of texture arrays

The latest version of the tool can be downloaded here.

*Note: Texture arrays are allocated as contiguous blocks of memory. Modifying any texel within any element of the array will cause the driver to ghost the entire texture array. The KHR_debug logging will report when these ghosting events occur.*

### 3.2.2. 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 on all PowerVR Graphics Cores is PVRTC 2bpp and, 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. On the latest PowerVR GPU cores, ASTC compression is also available.

**Performance Improvement**

The smaller memory footprint of PVRTC means less data is transferred from memory to the Graphics Core 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.

### 3.2.3. Image File Compression vs. 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 Graphics Core. 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.

This allows all the advantages mentioned in Section 3.2.2, 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.

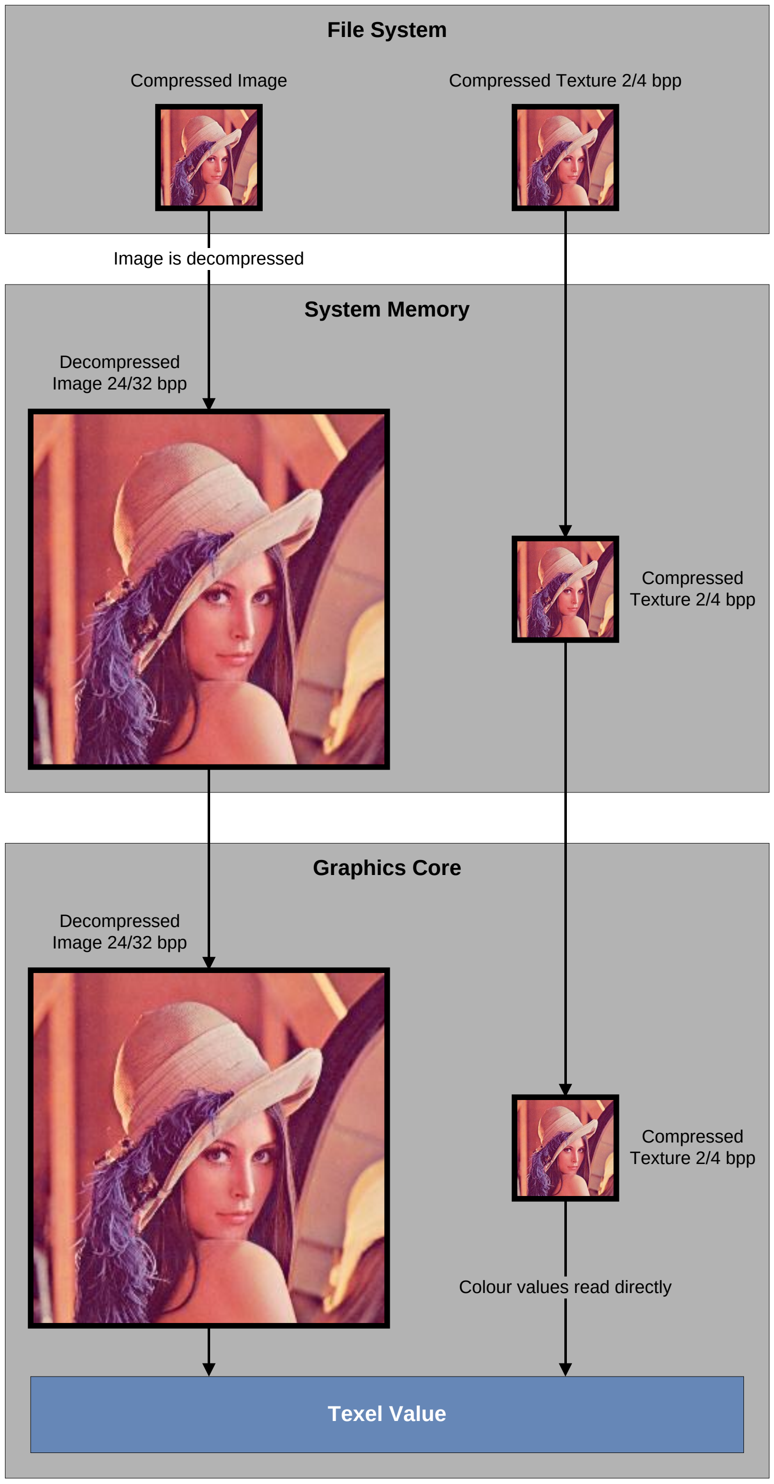

File System
Compressed Image
Compressed Texture 2/4 bpp
Image is decompressed
System Memory
Decompressed Image 24/32 bpp
Compressed Texture 2/4 bpp
Graphics Core
Decompressed Image 24/32 bpp
Compressed Texture 2/4 bpp
Colour values read directly
Texel Value

**Figure 5. Image file compression vs. texture compression**

## 3.3. 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 Graphics Core 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 right MIP-map for texturing.

### 3.3.1. Advantages

Using MIP-maps has two important advantages, namely 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 that do not use MIP-mapping.  The single limitation 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 in terms of 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.

### 3.3.2. Generation

Ideally MIP-maps should be created offline using a tool like PVRTexTool, which is available as part of the PowerVR Graphics 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`.

### 3.3.3. 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 3.4.1), and thus achieve an even higher image quality.

## 3.4. Texture Sampling

### 3.4.1. 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 trilinear filtering, the texture unit in the Graphics Core 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 on Series5 and Series5XT Graphics Cores, 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 Graphics Cores, particularly if the shader in question involves a lot of mathematical calculation. This latency can be hidden by swapping in another thread on the Graphics Core. 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 the “PowerVR Hardware Architecture Guide for Developers”.

The three main techniques for texture filtering are bilinear, trilinear, and anisotropic, where each gives 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” (see “PowerVR Performance Recommendations: The Golden Rules”). Also, not using anisotropic if trilinear is acceptable. Not using trilinear if bilinear is also acceptable.

### 3.4.2. Dependent Texture

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 in fragment shaders where the texture read is based on the `.zw` channels of a varying. On some driver and platform revisions `Texture2DProj()` also qualifies as a dependent texture read if given a `Vec3` or a `Vec4` with an invalid `w`.

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.

Dependent texture reads are significantly more efficient on PowerVR Rogue Graphics Cores than SGX. However, there are still small performance gains to be had. For this reason, applications should always calculate coordinates before fragment shader execution unless the algorithm relies on this functionality.

### 3.4.3. 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.

## 3.5. 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 in OpenGL ES.

### 3.5.1. OpenGL ES Support

NPOT textures are supported as required by the OpenGL ES specifications. However, it is necessary to point out the following:

- 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.

### 3.5.2. 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.

### 3.5.3. 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, where 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, texture usage and the target platform.

## 3.6. Texture Uploading

When a texture is uploaded through the use of `glTexImage2D` the input data is in linear scan-line 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 frame rate dips associated with additional texture loading.
- Avoid uploading texture data mid-frame to a texture object that has already been used for that frame.
- Consider performing a “warm-up” step after texture uploads have been performed. Once again, this helps avoid the frame rate dips associated with texture loading.

### 3.6.1. Texture Warm-up

The warm-up step mentioned before 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.

### 3.6.2. Texture Formats and Precision

In general, textures should be read at `lowp` (see Section 4.5.7). 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`.

## 3.7. Render to Texture

The preferred method for rendering to textures on OpenGL ES 2.0 is through the use of Frame Buffer Objects (FBOs) with textures as attachments.

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. For optimal performance, attachments should be unique to each FBO and attachments should not be added or removed once the FBO has been created.

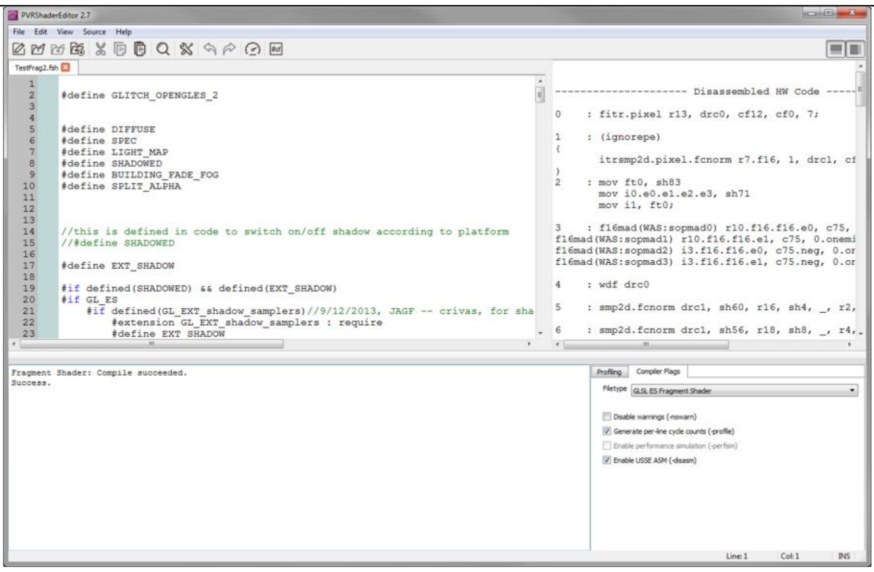
## 3.8. Mathematical Look-ups

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 coordinates in the look-up table, vary wildly between adjacent fragments then cache efficiency will suffer. As a significant amount of work must be saved for this to be an optimisation, profiling should be performed to determine if the results of using look-up tables are acceptable.

# 4. Optimizing Shaders

## 4.1. PVRShaderEditor

To demystify shader optimization, we provide a GUI utility called PVRShaderEditor (Figure 6) to share a wealth of off-line performance analysis data for developers as shaders are being written.



```

#define GLITCH_OPENGLES_2


#define DIFFUSE
#define SPEC
#define LIGHT_MAP
#define SHADOWED
#define BUILDING_FADE_FOG
#define SPLIT_ALPHA



//this is defined in code to switch on/off shadow according to platform
//#define SHADOWED

#define EXT_SHADOW

#if defined(SHADOWED) && defined(EXT_SHADOW)
#if GL_ES
    #if defined(GL_EXT_shadow_samplers)//9/12/2013, JAGF -- crivas, for sha
        #extension GL_EXT_shadow_samplers : require
        #define EXT_SHADOW
```

```
--------------------- Disassembled HW Code -----

0     : fitr.pixel r13, drc0, cf12, cf0, 7;

1     : (ignorepe)
{
        itrsmp2d.pixel.fcnorm r7.f16, 1, drc1, cf
}
2     : mov ft0, sh83
        mov i0.e0.e1.e2.e3, sh71
        mov i1, ft0;

3     : f16mad(WAS:sopmad0) r10.f16.f16.e0, c75,
f16mad(WAS:sopmad1) r10.f16.f16.e1, c75, 0.onemi
f16mad(WAS:sopmad2) i3.f16.f16.e0, c75.neg, 0.or
f16mad(WAS:sopmad3) i3.f16.f16.e1, c75.neg, 0.or

4     : wdf drc0

5     : smp2d.fcnorm drc1, sh60, r16, sh4, _, r2,

6     : smp2d.fcnorm drc1, sh56, r18, sh8, _, r4,
```

```
Fragment Shader: Compile succeeded.
Success.
```

Filetype: GLSL ES Fragment Shader

- [ ] Disable warnings (-nowarn)
- [✓] Generate per-line cycle counts (-profile)
- [ ] Enable performance simulation (-perfsim)
- [✓] Enable USSE ASM (-disasm)

**Figure 6. PVRShaderEditor GUI**

Additionally, we provide shader disassembly for PowerVR Rogue GPUs within the tool so you can see the exact GPU instructions that have been generated by the compiler for your shader. Key features of the tool include:

- Syntax highlighting for GLSL ES, GLSL, PFX, HLSL and OpenCL Kernels
- Supports PowerVR Series5, Series5XT and Series6 offline GLSL ES compilers
- Per-line cycle count estimates (PowerVR Series5, Series5XT and Series6 GPUs)
- Simulated performance estimates (PowerVR Series5 and Series5XT GPUs)
- Series6, Series6XT and Series6 FP16 disassembly

The latest version of the tool can be downloaded here.

## 4.2. 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.

## 4.3. 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 is often best to use this single space throughout.

## 4.4. 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”, on the other hand, 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. A better solution when an uber-shader is desired is to use pre-processor defines to create separate shaders from one larger shader at build time, effectively creating many smaller shaders from a single original source file.

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. In general, the following specific points should be considered:

- Make use of conditionals to skip unnecessary operations when the condition is met in a significant number of cases.
- Do not branch to `discard` (see “PowerVR Performance Recommendations: The Golden Rules”).
- **Series5 and Series5XT only**: Avoid branching to a texture read as samplers in dynamic branches qualify as “dependent texture reads” and will harm performance.

## 4.5. Demystifying Precision

PowerVR hardware is designed with support for the multiple precision features of graphics APIs such as OpenGL ES 2.0 and OpenGL ES 3.0. Three precision modifiers are included in the API spec for OpenGL ES 2.0 onwards, namely `mediump`, `highp`, and `lowp`. Lower precision calculations can be performed faster, but need to be used carefully to avoid trouble with visible artefacts being introduced. The best method of arriving at the right precision for a given value is to begin with `lowp` or `mediump` for everything (except samplers) then increase the precision of specific variables until the visual output is as desired.

### 4.5.1. Highp

Float variables with the `highp` precision modifier will be represented as 32 bit floating point values, whereas integer values range from $2^{31}$-1 to -$2^{31}$. 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 where the precision, or range, of `mediump` is not sufficient. It should also be used for any scalar calculations that use complex built-in functions such as `sin`, `cos`, `pow`, `log`, etc.

### 4.5.2. Mediump

Variables declared with the `mediump` modifier are represented as 16 bit floating point values covering the range [65520, -65520]. The integer values cover the range [$2^{15}$-1, -$2^{15}$]. This precision level typically offers a performance improvement over `highp`, and should be considered wherever `highp` would normally be used, provided the precision is sufficient and maximum and minimum values will not be overflowed.

### 4.5.3. Lowp

A variable declared with the `lowp` modifier will use a 10 bit fixed point format on Series5, allowing values in the range [-2, 2] to be represented to a precision of 1/256. The integer values are in the range of [$2^9$ -1, -$2^9$]. 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.

### 4.5.4. Swizzling

Swizzling is the act of accessing or reordering the components of a vector out of order. Some examples of swizzling can be found next:

```
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 on Series5 (lowp only) and Series5XT (all precisions) due to the additional work required to reorder vector components. As PowerVR Series6 is scalar based, swizzling is a significantly cheaper operation.

### 4.5.5. 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.

### 4.5.6. 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.

**Packing Varyings**

Packing multiple varyings together, for example packing two `Vec2` into a single `Vec4` should suffer no performance penalty and will save varyings. Exclusively on PowerVR Series5 and Series5XT, co-ordinate varyings which are packed into the `.zw` channel of a `Vec4` will always be treated as a dependent texture read and should be avoided (see Section 3.4.2).

### 4.5.7. 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`. On the other hand, if the sampler will be used to read from a half float texture then it should be declared as `mediump`.

### 4.5.8. 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. This is because 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.

**Constant Calculations**

The PowerVR shader compiler is able to extract calculations based on constant values (for example uniforms) from the shader and perform these calculations once per draw call.

### 4.5.9. 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 while unpacking is the reverse and involves taking a lower precision value and placing it into a higher precision variable.

Where possible precisions should be kept the same for an entire calculation as each pack and unpack has a cost associated with it. This cost can be further amortised by writing shaders in such a way that all higher precision calculations are performed together, at the top of the shader, and all lower precision calculations performed at the bottom. This ensures that variables are not repeatedly packed and unpacked. It also ensures that variables are not all unpacked into `highp` thereby losing any benefit of using lower precision.

## 4.6. Scalar Operations

It is very easy to accidently vectorise a calculation. Hence, one should be wary of vectorising scalar operations where it cost more cycles for the same output. For example:

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

// Bad
v2 = (v1 * x) * y; // vector * scalar followed by vector * scalar totals 8 scalar muladds

// Good
v2 = v1 * (x * y); // scaler * scalar followed by vector * scalar totals 5 scalar muladds
```

## 4.7. “Const” Data in Shaders

If used correctly the `const` keyword can provide a significant performance boost. For example, a shader that declares a `const` array outside of the `main()` block can perform significantly better than the same shader with the array not marked as `const`, even if the array could be treated as such. Another example would be the use of a `const` value to reference an array member. In this example, if the value is `const` the Graphics Core can know ahead of time that the number will not change and data can be pre-fetched prior to the shader being ran.

# 5. Optimizing Specific Techniques

## 5.1. Multiple Render Targets (Series6 only)

Multiple Render Targets (MRTs) are available in a variety of APIs, and are supported on PowerVR hardware from Series6 onwards. By using MRTs properly, developers can take advantage of the tile-based architecture of PowerVR hardware, keeping all render targets entirely on-chip for a significant performance boost. In order to benefit from this feature the combined bit rate of all MRTs should be no more than 128bits per pixel.

## 5.2. Efficient Sprite Rendering

Rendering sprites efficiently may seem like a trivial exercise. However, without careful consideration an application may be unresponsive and sluggish due to poor graphics performance. Traditional sprite render tends to see textures drawn, using alpha blending, on to quads. These quads will consist of large areas of alpha, either full alpha, or partial alpha. Areas of full alpha are traditionally discarded using either the `discard` keyword or alpha testing, while areas of partial alpha undergo blending. Both of these have some form of impact on performance versus fully opaque objects meaning that a large number of sprites being drawn inefficiently can seriously harm performance.

**Comment [RA-J1]:** Render"ing" ?

The `discard` keyword (see "PowerVR Performance Recommendations: The Golden Rules") should be avoided in favour of the much faster alpha blending. Even when favouring alpha blending, performance can still be affected if there are a large number of sprites. One method to minimise the impact of several layers of blended sprites is to increase the geometry complexity of the sprites in order to reduce the amount of wasted transparent fragments. For example, if a sprite is circular in shape and is rendered using the most optimal fitting quad, 22% of the fragments processed are redundant. Significant performance improvements can be gained by reducing the wasted transparency by increasing geometry complexity.

PowerVR hardware has excellent vertex processing capabilities and is designed to handle large amounts of geometry data, far in excess of what is present in most sprite based applications. As such, increasing complexity should have minimal performance impact and any impact this may have is most likely outweighed by the savings of rendering less transparency. If we increase the complexity of the previous case of a perfectly fitting quad around a circular sprite to that of a dodecagon (twelve sided polygon) we can reduce the amount of wasted fragment processing to just 3%.

Assuming radius $\boldsymbol{r}$ of 64

$$A = 1 - \frac{\pi r^2}{(2r)^2} \qquad vs \qquad A = 1 - \frac{\pi r^2}{12\left(2-\sqrt{3}\right)r^2}$$

$$A = 0.214 \qquad\qquad A = 0.029$$

$$A = \mathbf{21.4\%} \qquad\qquad A = \mathbf{2.9\%}$$

**Figure 7. Increasing complexity and reducing processing**

## 5.3. Draft MSAA Performance on Series6+

Using MSAA (multisample anti-aliasing) in your render is costly because when there is a blended edge, this sets the pixels on the edge to be "on edge blend". On edge blend is a very costly operation, as the blend is done for each sample by a shader. In contrast, on opaque edge is done by dedicated hardware, and is a much cheaper operation as a result. On edge blend is also sticky, which means that once an on-screen pixel is marked, all subsequent blended pixels are blended by a shader, rather than by dedicated hardware. This is very costly in a trace because it starts with a large amount of on edge blends. These pixels are then stuck on edge for the whole render, unless fully covered by an opaque. In order to avoid these costs, submit all opaque/no blend geometry first, which keeps the pixels “off edge” for as long as possible. Also, developers should be extremely reserved with the use of blending, as blending has lots of performance implications, not just for MSAA.

In general, MSAAs are considered to cost very little. This is true for typical games and UIs, as those have low geometry counts but very complex shaders. The complex shaders typically hide the cost of MSAA and have less blend workload.

## 5.4. Draft glClears & glColorMask

It is key to avoid a partial clear (partial colour mask) at the start of a frame. This is due to two reasons:

1. The previous frame has to be read in. This is performed by a full screen primitive reading it in as a texture.
2. This texture must be masked out by the partial clear, which is done by submitting another full screen primitive as a blend.

It is important to note that this results in two overdraws before we even begin working on the frame. If we then change the colour mask to full and glClear, this counts as a state change for colour mask. A state changes requires a flush on the tiler and the clear becomes another full screen primitive. This adds yet another overdraw. In the case of one full clear (no partial colour masks) at the start of frame, we go down the “fast clear” path. This marks the whole frame as a set colour and does nothing, so no fullscreen primitive is required, resulting in no pixels being drawn at all.

*Note: PVRTrace GUI emulates this behaviour.*

## 5.5. Draw*Indirect and MultiDraw*IndirectEXT

### 5.5.1. Draw*Indirect

A standard OpenGL ES draw call requires the programmer to pass the parameters of the draw via the function’s arguments. With the Draw*Indirect calls, the programmer can instead pass in a structure containing the draw parameters. The great thing about this structure is that it doesn’t have to be populated by the CPU – a programmer can use the GPU and SSBOs to populate the structure. This enables the application to issue a draw without any CPU-side involvement.

**Example Use Case: Batched Draws**

For optimal performance, applications should batch draws by state to reduce the number of API calls. The problem here though is that a separate draw call needs to be issued for each object in that batch and draw calls have a CPU overhead we would like to avoid. With Draw*Indirect, developers can populate an SSBO with the vertex data of all draws that share the same state. With this SSBO, only a single Draw*Indirect needs to be made.

**Example Use Case: Particle Systems**

Another use case could be a particle system where the developer doesn’t want to allocate a big array for particles up front. Instead, they could use a compute shader to determine how many particles need to be rendered each frame. For really complex particle systems, they could also remove particles from the render if they are obscured by opaque objects.

### 5.5.2. MultiDraw*IndirectEXT

These API calls are very similar to Draw*Indirect. The key difference is that an array of Draw*IndirectCommand structs can be passed into each draw call.

**Example Use Case: GPU-side Occlusion Culling**

In complex 3D navigation systems, draw calls tend to be grouped by map tiles. If a map tile intersects the view frustum, all draw calls within the tile are issued to the GPU. This can be optimized with occlusion queries to further reduce the number of draw calls that are issued. With MultiDraw*IndirectEXT you can do even better though. A compute shader can be used to populate an array of Draw*IndirectCommand structures. Those can then be used to issue a single draw call for many objects sitting in many different tiles.

## 5.6. PBO Texture Uploads

PBOs are Pixel Buffer Objects. They were introduced in OpenGL ES 3.0 and enable applications to map GL driver allocated textures into the application's address space. Once mapped, the application can then read from or write to the texture from the CPU.

### 5.6.1. Optimal Texture Updates with PBOs

PBOs can be used to reduce the number of memcpy's required to transfer data to GPU accessible memory. For example, if we wanted a very fast upload of texture data from file we could create a PBO and directly load the contents of the file into this memory. If, however, the file was loaded into application memory first and then memcpy'd it into the PBO, we'd be performing as many memcpy's as a glTexImage2D would.

**Comment [RA-J2]:** Memcpys, memcpies, memcopies, memcopys..?

**Comment [RA-J3]:** Memcpys, memcpies, memcopies, memcopys..?

**Comment [RA-J4]:** Memcpys, memcpies, memcopies, memcopys..?

**When are transfer (TQ) tasks kicked?**

If TexStorage has been used to define the texture, transfer tasks for PBO writes will be kicked when TexImage is called. Note that the PBO must be unmapped before any GL calls are issued for the texture. Failing to do so will result in an error. If TexStorage hasn't been used, the transfer task will be deferred to the first draw call that uses the modified texture.

**If there is already a copy of the texture in GPU memory, will the data be copied to the PBO so it's preserved?**

Yes. By default, the driver will have to TQ copy the mapped region of the texture from twiddled GPU memory to the driver's PBO buffer.

If you do not need to preserve the mapped region, you can specify the GL_MAP_INVALIDATE_RANGE_BIT access flag when calling glMapBufferRange. If the entire texture can be invalidated, then you can use the GL_MAP_INVALIDATE_BUFFER_BIT flag.

## 5.7. Rogue DDK

### 5.7.1. glTexStorage2D & glTexStorage3D

glTexStorage2D & glTexStorage3D were introduced in OpenGL ES 3.0. They provide a mechanism to define immutable-format textures, i.e. textures where the format and dimensions of all levels cannot be altered after their creation. The main benefit of immutable-format textures is that they reduce the amount of validation the driver has to perform. Texture format validation is performed up front and only once for all texture levels.

### 5.7.2. Depth/Stencil Load/Store

When rendering to an FBO, depth and stencil are saved (Z/stencil store) by default. If you call glInvalidateFramebuffer(GL_DEPTH | GL_STENCIL), the Z/stencil STORE is skipped. If you call glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT), the Z/stencil LOAD is skipped.

## 5.8. Physically Based Rendering & Per-Pixel LOD Rogue Performance

Physically based rendering (PBR) is a forward and deferred render compatible lighting model that aims to better represent real-world light behaviour. It is more costly to calculate than traditional diffuse, specular, and ambient lighting, but it is very appealing to artists as it makes it easier to specify complex material properties. PBR art pipelines are rapidly becoming the norm in AAA titles.

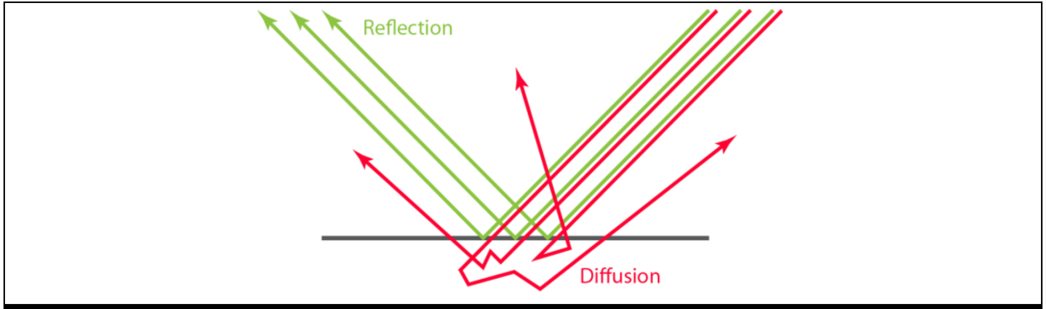


**Figure 8. PBR Theory**

### Per-Pixel Texture LOD

PBR pairs each object in a scene with a roughness/gloss map. This texture allows artists to alter the roughness/glossiness across an object rather than having a consistent texture across the entire surface. An example use is to add areas of dull rust to a shiny pistol, or to describe the properties of a rubber grip, all within a single draw call.

To add an element of reflectivity, environment maps are applied to all objects. Each environment map contains progressively blurrier surfaces towards the bottom of the chain. The sampled roughness value is used to calculate which MIP level of the environment map should be sampled.

### Why is this a problem for Rogue?

Rogue sub-divides a fragment shader USC task into 2×2 blocks of spatially aligned pixels. A primary reason for doing this is so gradients can be calculated across a pixel-quad to determine how texture filtering should be applied. It is also optimized for the standard rendering case where a LOD value is calculated for a pixel-quad based on the calculated gradients. This allows the GPU to batch texture sample operations for the pixel-quad into a single TPU request. When texture LOD is specified per-pixel, the GPU has to assume that each pixel in the quad has a unique LOD. This causes the USC to issue a TPU request for each pixel instead of the entire quad, which in turn causes 1/4 TPU throughput.

### The Workaround

There is a GLSL workaround to avoid the ¼ speed path. However, it introduces dynamic branching and additional instructions.

**Comment [RA-J5]:** Consistency with ¼ in previous paragraph

```
#version 310 es

in mediump float LOD;
in mediump vec3 TexCoords;

uniform lowp samplerCube EnvMap;

layout (location = 0) out lowp vec4 oColour;

mediump vec4 envSample(lowp samplerCube envMap_, mediump vec3 texCoords_, mediump float LOD_){
        mediump vec4 mip0;
        mediump vec4 mip1;

        if(LOD_ <= 4.0){
                if(LOD_ <= 2.0){
                        mip1 = textureLod(envMap_, texCoords_, 1.0);
                }else{ // LOD_ > 2.0
                        mip1 = textureLod(envMap_, texCoords_, 3.0);
                }
        }else{ // LOD_ > 4.0
                if(LOD_ <= 6.0){
                        mip1 = textureLod(envMap_, texCoords_, 5.0);
                }else{ // LOD > 6.0
                        mip1 = textureLod(envMap_, texCoords_, 7.0);
                }
        }

        if(LOD_ <= 3.0){
                if(LOD_ <= 1.0){
                        mip0 = textureLod(envMap_, texCoords_, 0.0);
                }else{ // LOD_ > 1.0
                        mip0 = textureLod(envMap_, texCoords_, 2.0);
                }
        }else{ // LOD_ > 3.0
                if(LOD_ <= 5.0){
                        mip0 = textureLod(envMap_, texCoords_, 4.0);
                }else{ // LOD_ > 5.0
                        mip0 = textureLod(envMap_, texCoords_, 6.0);
                }
        }

        bool isEven = ((int(LOD_) & 1) == 0);
        mediump float fractVal = fract(LOD_);
        mediump float invFractVal = 1.0 - fractVal;
        mediump float mixVal = isEven ? fractVal : invFractVal;
        return mix(mip0, mip1, mixVal);
}

void main() {
        oColour = envSample(EnvMap, TexCoords, LOD);
}
```

## 5.9. VAOs, UBOs, Transform Feedback Buffers and SSBOs in OpenGL ES

This section provides a quick reference for the mentioned items.

### 5.9.1. VAOs – Vertex Array Objects

Encapsulates bound vertex state e.g. `glVertexAttribPointer`. Binding a VAO applies all of the encapsulated state to the global GL state. The ID Zero ('0') is reserved by GL to represent the default vertex array object. Always use VAOs when working with SGX or Rogue.

*Note: VAOs only store the bound `GL_ELEMENT_ARRAY_BUFFER`. Any binding to `GL_ARRAY_BUFFER` or other buffer types are separate and distinct from VAOs.*

**APIs**

- **OpenGL ES 1.x & 2.0:** Extension (`GL_OES_vertex_array_object`) This is discussed in greater detail in the PowerVR Supported Extensions document).
- **OpenGL ES 3.0:** Core.

### 5.9.2. UBOs – Uniform Buffer Objects

Instead of uploading uniform data from client memory (e.g. `glUniformMatrix4fv` & `glUniform1f`), this feature allows you to store uniform data in a buffer object (UBO). Applications can map and unmap buffers to modify them. There are a number of slots available for UBOs, so an application can use more than one for each draw. Apps can also use client side uniform data at the same time as UBOs. This enables applications to easily split data into separate buffers depending on the frequency the data is updated e.g. static data in one UBO, frequently changing data in another. There should always be used when working with Rogue, but not when working with SGX.

**APIs**

- **OpenGL ES 1.x:** Not exposed.
- **OpenGL ES 2.0:** Exposed (IMG_uniform_buffer_object).
- **OpenGL EWS 3.0:** Core.

### 5.9.3. Transform Buffer Objects

These buffers are used for transform feedback. When these are bound, post-transform vertex data is automatically resolved to the buffer by the GPU. The buffers can be written to/read by the GPU, but cannot be accessed by the CPU. You should always use these buffers when working on Rogue, but they are not available for SGX architectures.

**APIs**

- **OpenGL ES 1.x & 2.0:** Not exposed.
- **OpenGL ES 3.0:** Core.

### 5.9.4. SSBOs – Shader Storage buffer Objects

SSBOs are similar to UBOs. For example, storage blocks are defined in GLSL and SSBOs are bound to SSBO binding points. Unlike UBOs, SSBOs:

- Can be written to by the GPU.
- Can be used as compute kernel input/output.
- Can be much larger than UBOs (MBs instead of KBs).
- Have variable storage up to the range bound for the given buffer. The actual size of the array, based on the range of the buffer bound, can be queried at runtime in the shader using the length function on the unbounded array variable.

Although SSBOs have similar features to UBOs and transform feedback buffers, the flexibility comes at a cost. SSBO GPU reads may be more costly than UBOs, as data is fetched like a buffer texture instead of being pre-loaded into shader registers (as UBOs would). Unlike transform feedback buffers that are written to automatically when bound, SSBOs need to be written to explicitly in shader code. SSBOs are not available on SGX, and when you use Rogue it is recommended you favour UBOs and transform feedback buffers instead, as they take optimal paths through the pipeline. SSBOs are best suited for draw indirect/dispatch compute indirect use cases.

**APIs**

- **OpenGL ES 1.x, 2.0 & 3.0:** Not available.
- **OpenGL ES 3.1:** Core.

## 6. 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