# POWERVR SGX

# OpenGL ES 2.0 Application Development Recommendations



Filename : POWERVR SGX.OpenGL ES 2.0 Application Development Recommendations.mht

Version : 1.1f (POWERVR SDK 2.05.25.0295)

Issue Date : 27 Jan 2009

Author : POWERVR

# Contents

# List of Figures

# 1. Introduction

POWERVR SGX is a new 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.

This document contains recommendations and advice for developers who wish to use version 2.0 of the OpenGL ES API on POWERVR SGX enabled devices. Therefore references to hardware features and characteristics apply to members of the POWERVR SGX family targeted at the mobile and embedded markets, programmed using the OpenGL ES APIs.

## 1.1. What's new?

### 1.1.1. Vertex and fragment Shaders

The most significant departure from the previous generation is the introduction of vertex and fragment shaders, replacing a large part of the fixed-function vertex and fragment pipeline with fully programmable stages. This gives developers a huge amount of flexibility to create complex and convincing visual effects, and to offload work to the GPU that previously had to be performed by the CPU. A large part of this document is devoted to guidelines for shader programming.

Some parts of the pipeline – such as rasterization, visibility testing, and texture filtering – remain fixed-function to guarantee maximum efficiency.

### 1.1.2. Unified Shader Architecture

The architecture of the POWERVR SGX family is based around its fully programmable multi-threaded Universal Scalable Shader Engine (USSE), which performs all the processing for both vertex and fragment shaders. This unification of vertex and fragment processing has two very important implications:

- **Load Balancing:** In an architecture with separate units for vertex and fragment processing precious cycles are often wasted due to the workloads not exactly matching the hardware capabilities. Reducing the vertex workload will not improve performance if the fragment shader is the bottleneck. On POWERVR SGX however, a reduction in vertex processing will result in more cycles available for fragment shading, and vice versa, thus always resulting in maximum utilization of the USSE core.
- **Mostly Identical Capabilities:** Both vertex and fragment shaders support full flow control. Both can read from textures. The same precisions are available in both shader types. Some limitations are imposed by the API and by the inherent differences between vertices and fragments, though.

The USSE pipeline also performs some "fixed-function" tasks such as alpha blending and conversion of vertex attributes to floating point values. While this implies a performance penalty when using these features, it increases the overall GPU efficiency because there are no specialized hardware units which would be idle when those features are not used.

### 1.1.3. Multiple Precisions and Scalar Processing

To achieve best performance for a variety of tasks, the USSE supports multiple precisions. The GLSL ES precision modifiers, `lowp`, `mediump`, and `highp`, map to 10-bit fixed point, 16-bit float and 32-bit float types, respectively. While `lowp` computations are performed as 3 and 4 component vector operations, `mediump` operations use 2 component vectors. When using `highp` precision, the USSE operates on scalar values.

This scalar processing has a significant influence on low-level shader optimizations. Scalar code can be a lot more efficient than vector code since you only need to calculate those vector components that contribute to the final result. But keep in mind that you don't get calculations "for free" by squeezing multiple scalars into a vector.

### 1.1.4. New Texture Formats

In addition to the texture formats already supported by POWERVR MBX (including common formats like 16 and 32-bit RGBA and PVRTC compressed textures), members of the POWERVR SGX family can handle 16 and 32-bit floating point textures.

POWERVR SGX also supports the ETC compressed texture format which is expected to be implemented by a wide range of devices, and introduces support for cube map textures and limited non-power-of-two (NPOT) textures as required by OpenGL ES 2.0.

### 1.1.5. Stencil Buffer

One of the fixed-function features missing from POWERVR MBX, stencil buffer support has been added to POWERVR SGX to allow techniques such as stencil shadows or constructive solid geometry (CSG).

### 1.1.6. Improved Anti-Aliasing

POWERVR SGX further improves the anti-aliasing performance and quality of the previous generation MBX family by offering 4-sample sparse grid multisampling anti-aliasing (MSAA) which offers quality that often comes close to 16-sample ordered grid anti-aliasing.

Anti-aliasing is essential for achieving the best image quality. Since the POWERVR architecture can keep the multisample buffers entirely on-chip and only writes out the resolved framebuffer to memory, POWERVR SGX does not suffer from the large framebuffer memory overhead typical immediate mode renderers (IMR) have to bear when using multisampling.

## 1.2. What's not-so-new (but important)?

While POWERVR SGX is in many ways a massive leap forwards from the MBX family, the fundamentals of the proven concept have been retained.

### 1.2.1. Perfect Hidden Surface Removal

POWERVR graphics cores employ a method called Tile Based Deferred Rendering (TBDR). All the geometry data and state required for a frame is captured into a scene buffer in memory, and fragment processing is deferred until the scene has been completely submitted.

Because the whole scene is known at the time of rendering, the POWERVR architecture can perfectly compute the visibility of any surface before performing any fragment processing, independent of the order in which scene elements were submitted. This means applications do not have to resort to depth sorting a scene to improve rendering performance. With this method, only those parts of surfaces that are visible in the final image will be computed by the fragment pipeline.

For rendering the viewport is divided into tiles, rectangular regions which are rendered individually using on-chip colour and depth buffers. The use of on-chip buffers ensures that no external bandwidth is used for framebuffer blending and visibility testing, significantly lowering overall bandwidth requirements.

### 1.2.2. Designed for Efficiency and Low Power Consumption

In mobile devices, battery life is of paramount importance. The POWERVR SGX family therefore employs sophisticated power saving techniques to keep battery drain at a minimum. While POWERVR SGX provides a feature set that rivals that of recent desktop PC hardware, the power consumption of those mobile devices is more than two orders of magnitude below that of high-end PC graphics cards. Obviously this also has an impact on performance, and you should not expect to run the latest, hundreds-of-instructions PC game shaders on a mobile device at interactive framerates, even with a screen at just VGA resolution.

### 1.2.3. Massive Depth/Stencil Fillrate

Because the visibility determination requires an enormous amount of depth and stencil tests, POWERVR graphics cores are designed to perform these at a very high rate. This, along with the bandwidth savings provided by the on-chip depth buffer as well as the addition of stencil functionality, makes POWERVR SGX a perfect match for techniques such as stencil shadows.

# 2. Golden Rules

**Understand your target and benchmark your application**

Attend to the API, OS and platform specifics and measure performance with different settings. No two devices are identical.

**Do not access the framebuffer from the CPU**

Any access to the framebuffer will cause the driver to flush queued rendering commands and wait for rendering to finish, removing all parallelism between the CPU and the different modules in the graphics core.

**Batch your primitives to keep the number of draw calls low**

Try to minimise the number of calls used to render your scene as these can be expensive. Using branching in your shaders may help to have better batching.

**Perform rough object culling on the CPU**

Your application has more knowledge about the scene than the GPU. Whenever you have the opportunity to quickly detect that objects are invisible, don't render them!

**Perform calculations per vertex instead of per fragment whenever possible**

The number of vertices processed is usually much lower than the total number of fragments, so operations per vertex are considerably cheaper than per fragment.

**Avoid `discard` in the fragment shader**

POWERVR architectures offer performance advantages that are negated when discard is used.

**Use texture compression and mipmapping**

Texture compression and mipmapping reduce memory page-breaks and will make better use of the texture cache.

**Use vertex buffer objects and indexed geometry**

Vertex and index buffers will benefit from driver and hardware optimisations for fast memory transfers.

**Target a fixed framerate**

The smoothest animations can be achieved by running at a constant framerate. By having an upper limit for the framerate you can save power when scene complexity is low.

# 3. Optimisation Strategies

Modern 3D graphics cores are a very complex system consisting of many sub-blocks with very specific and complex performance characteristics. This section of the document aims to provide a basic overview of the POWERVR SGX structure. At the same time all potential bottlenecks are identified together with detection methods and possible solutions.

When it comes to finding bottlenecks and optimising performance, the most important rule is: benchmark it! Every scene is different and stresses different parts of the hardware. The *PVRScope* utility which is part of the POWERVR SDK is an important tool to help you find bottlenecks and discovering where there is headroom for additional scene complexity.

## 3.1. SGX Hardware Architecture

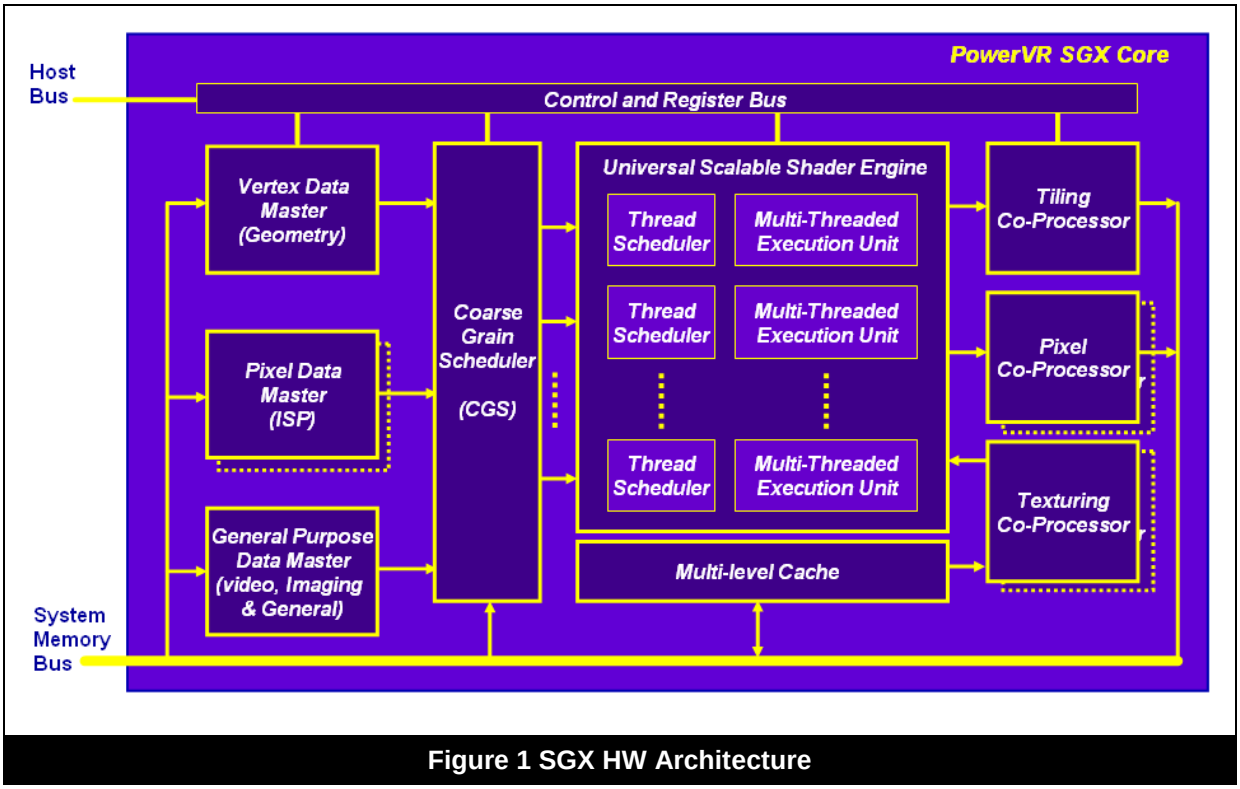


**Figure 1 SGX HW Architecture**

1. When tracing a single triangle through the POWERVR SGX pipeline it starts of at the Memory Interface.
2. The geometry data is submitted by the scheduler into the Universal Scalable Shader Engine (USSE) with a vertex shader program to generate its output.
3. The output is passed onto the Tiling Co-Processor, which is responsible for clipping (when one or more clip-planes are enabled), culling (backface, off screen and small polygon), and tiling (determining which tiles and macro tiles contain the triangles processed).
4. The Tiling Co-Processor writes its compressed data stream into memory. The Image Synthesis Processor (ISP), per tile, fetches all the relevant triangles. After a set up operation, this triangle data is processed to do the actual depth/stencil testing.
5. A Visibility Tag List is then, per tile, passed onto the scheduler to perform the fragment shader operations by the USSE. Results of the USSE are processed by the Pixel Co-Processor and written to the on-chip tile buffer.

SGX hardware has been designed to be very efficient keeping the processing units always busy. Instructions requiring data which is not available yet are parked into a thread list until the data is ready for processing. While these threads are waiting other operations are performed. Still each block shown in the overview can become a bottleneck by stalling units in front of it and starving units behind it.

We will now address each block separately and indicate its performance characteristics and how to identify whether the block is responsible for reducing performance. Note that during the rendering of complex frames a lot of different situations will occur, thus it is not unlikely to encounter a variety of different bottlenecks throughout a frame. At the same time it is important to realise that the front-end (Tiling Co-Processor and vertex shader) and the back-end (ISP and everything following it) of the chip are effectively decoupled by a parameter buffer. Hence, they will have little impact on each other since the buffering absorbs the respective stalls.

## 3.2. CPU

The CPU is one of the most common bottlenecks. It is caused by the many operations executed by the application, the API runtime and the OS. A lot of those operations have nothing to do with 3D graphics. For instance, in games, a large part of the CPU time is spent on computing physics, AI, sound, and other things. The best way to detect whether the CPU is a bottleneck is by profiling where the application time is spent. If very little time is spent in the graphics driver, then the chances are you have a CPU bottleneck. A more basic way of identifying the CPU bottleneck is to check how the application scales with a different CPU, or a different CPU clock. This is also handy to rule out other limitations such as system bandwidth, file access, memory paging, etc…

## 3.3. Memory Bus

Heavy memory copies in your application can cause this bottleneck. If your application uses a lot of dynamic vertex data and vertex throughput is quite low then you might need to take measures to solve this problem. Try limiting the amount of dynamic data streamed to the graphics core to check if this is really the bottleneck. Some options are to reduce the vertex size, or to split off dynamic and non-dynamic vertex elements. Also avoid constant uploading of large textures or reading back data from the graphic core since both can stall the HW and flood the memory bus.

## 3.4. Vertex Shader (USSE)

The USSE should only rarely be a bottleneck at this moment considering POWERVR SGX's excellent vertex processing capabilities. It is more likely that the vertex throughput is constrained by memory throughput. Detection of this bottleneck can be performed by simplifying the vertex shader programs, e.g. reducing the number of lights and so on. Remember to do vertex shader throughput testing using a low render resolution to avoid the impact of other bottlenecks (e.g. fillrate) on your testing. Vertex texturing can have an impact on performance,

### 3.4.1. Vertex Processing FIFO

POWERVR SGX has a cache for processed vertices. This cache will hold a few previously transformed vertices so keeping them together will increase the Vertex Shader throughput. The smaller the transformed vertex format the more effective the FIFO will be so try to reduce the amount of data passed from the vertex shader to the fragment shader (varyings).

## 3.5. Tiling Co-Processor

### 3.5.1. Clipping and Culling

Clipping and Culling is very fast and is rarely the bottleneck. It is quite difficult to isolate from other bottlenecks though. If you are concerned that your application might be clipping or culling limited contact the POWERVR Developer Technology team who will be happy to assist you.

### 3.5.2. Tiling

Tiling is the process of setting up the pointers and geometry data per tile to be processed by the rasterizer. Tiling is very fast but the amount of time required to do tiling depends on the number of polygons submitted and the coverage (how many tiles are covered per polygon).

## 3.6. Image Synthesis Processor (ISP)

### 3.6.1. Setup

The ISP will do the depth sorting and stencil tests and will submit the pixels to be processed by the USSE unit. Generally this should not be a bottleneck due to the reduction in polygon count obtained by culling (back face, off-screen and small polygon culling), the lack of accesses to the external depth buffer and high parallelism (several pixels processed at the same time). The fragment shader should, on most occasions, be complex enough to absorb this bottleneck. If you think this might be your bottleneck avoid polygons that are “very” large and use just a very simple fragment shader (or have colour writes disabled), e.g. very large stencil volumes since they will have a high ISP Setup Load.

### 3.6.2. Z-Load/Store

Due to POWERVR SGX’s tile based deferred rendering a Z buffer is not usually required to be stored in external memory. Whenever possible, try to avoid retaining the same Z Buffer contents across different renders (e.g., not clearing a depth renderbuffer), as it might require the HW to store the Z buffer. This will typically result in a loss of performance and an increase in bandwidth requirements.

## 3.7. Fragment Shader (USSE)

### 3.7.1. Setup

Avoid using a high number of varyings since this will result in a high setup load.

### 3.7.2. Texture Fetches

SGX has a ‘latency optimal’ architecture where instructions waiting for a texture sampler will be parked until the data is ready, giving preference to other instructions. This effectively reduces the performance penalty usual in systems where the pipeline has to wait for texture data to be ready.

To reduce the chance of being texture limited, use texture compression whenever possible and always use mipmapping to make a more effective use of texture caching and to reduce the possibility of memory page-breaks.

### 3.7.3. Shader Instructions

Fragment shaders with a large number of instructions should be expected to be a bottleneck when applied to a large amount of pixels. SGX can perform a certain number of internal instructions (which might comprise several floating point operations) in a single cycle. The actual performance is linked to render-target resolution and clock speed. A bottleneck in fragment processing can be verified by reducing the number of arithmetic instructions used in the fragment shader, or by reducing the rendering resolution. If performance is severely limited by this you might need to simplify the shader effects to improve performance. The larger the area that is covered by a fragment shader the more important any instruction optimisation will become.

## 3.8. Parameter Buffer

The parameter buffer memory is written to by the Tiling Co-Processor and read from by the ISP. Follow the general guidelines given in this paper to avoid overflowing this buffer as an overflow of this buffer will penalise performance. If you suspect your application is parameter buffer limited contact the Developer Technology team for further investigation and recommendations.

To optimise parameter buffer usage, reduce the vertex format if possible, check that the vertex indexing is effective (i.e. there is good vertex sharing), sort the triangle list when creating models, and try to reduce the amount of varyings passed from the vertex shader to the fragment shader.

## 3.9. Texture Cache

A potential texture cache bottleneck can be identified by reducing the texture size, reducing the number of textures, changing the texture format (float to integer or compressed), and/or reducing the texture filter settings. As general advice, avoid performing very random texture accesses; these are especially expensive when doing dependent per-pixel perturbed texture reads. Highly random

accesses *will* thrash the texture cache. Use mipmapped textures whenever possible. Also use compressed textures which will fit better in cache memory.

# 4. Render State Management and Batching

## 4.1. Minimize the Number of State Settings and Draw Calls

### 4.1.1. Avoid Redundant Render State Settings

Setting OpenGL ES state values multiple times between draw calls – or setting the same value again – adds driver call overhead to your application. You should structure your code in a way that avoids these redundant calls. Where this is hard to achieve because state is set in different parts of your application, you can keep a copy of the current state setting on the application side and only call the related GL function if the old state and the new state are different.

Ideally, you should follow these simple rules:

- Set every GL 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

### 4.1.2. Texture Atlases

By using texture atlases you can keep the number of texture changes low. Texture atlases consist of multiple texture images, usually of the same size, arranged into a single texture. If you can, use them to group texture images which are used interchangeably with the same shader program.

One issue with texture atlases you need to be aware of is that texture filtering, especially with mipmaps, may cause texels from neighbouring images to blend into each other. To avoid or reduce artifacts from this you need to either leave large enough borders around each image or place images with sufficiently similar edges next to each other. Using texture atlases also clashes with using texture repeat modes.

### 4.1.3. Mesh Groups and Dynamic Geometry

If you have multiple meshes which have static positions and orientations relative to each other, and which could use the same render state, shader, and textures (including texture atlases), always combine them into a single mesh. Often this can be done in the modelling stage, but it also applies to dynamically generated geometry.

For example, if you're rendering a HUD with simple textured rectangles place all required texture images into one texture atlas, gather the rectangles to render and submit them in a single draw call. But do keep transparent and opaque objects separate as described in section 4.3.

## 4.2. Cull Invisible Objects

Although POWERVR SGX is very efficient at removing invisible objects, not submitting them for rendering at all is still faster. Your application has more knowledge of the scene contents and positions of objects than the GPU and OpenGL ES driver, and it can use that information to quickly cull objects based on occlusion or view direction. Especially when you're using complex vertex shaders it is important that you keep the amount of vertices submitted reasonably low.

To perform this culling it is important that your application uses efficient spatial data structures. Bounding volumes can help to quickly decide whether an object is completely outside the view frustum. If your application uses a static camera, perform view frustum culling off-line.

## 4.3. Rendering Order

The order in which objects are submitted for rendering can have a huge impact on the number of state changes required and therefore on performance. Additionally, blended objects often need to be rendered back-to-front to get the desired result.

### 4.3.1. Opaque First, Blend Last

Opaque objects are those which are rendered without framebuffer blending or `discard` in the fragment shader. Always submit all opaque objects first, before any transparent ones. Follow these four steps:

1. Separate your objects into three groups: opaque, using `discard`, using blending.
2. Render opaque objects sorted by render state.
3. Render objects using `discard` sorted by render state (but avoid `discard` if possible).
4. Render blended objects sorted back-to-front.

### 4.3.2. Sorting Opaque Objects by Render State

Due to the advanced hidden surface removal mechanism of POWERVR SGX you don't need to sort opaque objects by depth for good efficiency as is required on other architectures. In fact, it would be just a waste of CPU cycles to do so. Sort these objects in a way that minimizes render state changes instead. For example, by grouping objects based on the shader program they're using you can achieve that each shader program used is set exactly once.

Not all state changes are equally costly. Always submit all commands to draw one frame to a render target in one go, don't switch render targets mid-frame. Per frame, sort objects by the shader program used. Inside those groups try to minimize texture and uniform changes as well as all other state changes.

### 4.3.3. Sorting Transparent Objects by Depth

Some of the combinations of OpenGL ES blend equations and functions are *associative*, meaning that the resulting colour in the framebuffer does not depend on the order in which overlapping transparent objects were drawn. The most obvious examples of such blend modes are *additive blending* and *multiplicative blending*, as shown here:

```
glEnable(GL_BLEND);

// Additive Blending:
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE, GL_ONE);

// Multiplicative Blending:
glBlendFunc(GL_ZERO, GL_SRC_COLOR);
```

In these cases it does not matter which one of two overlapping objects you render first. However most other blend modes are order-dependent and require rendering objects from back to front in order to get correct results.

Back to front sorting isn't always trivial, and depending on how objects are shaped and how they overlap, it may sometimes be necessary to split meshes and even triangles to get the right order. Fortunately in many scenes blended objects are limited to flat polygons like window panes, particles, glow and lens flare effects, or leaves and other foliage. Typically just sorting by the view space Z of the object centre (which is equal to the last element of the third row in the modelview matrix of an object) already yields a satisfying render order.

Because this sorting can be quite CPU intensive when you want to render a large number of transparent objects, it is worth looking into methods to reduce this load. The most important observation here is that the depth order of objects usually stays roughly the same from frame to frame. Thus it is a good idea to start with the sort order from the previous frame and apply a sort algorithm like insertion sort with linear run-time on already sorted lists.

In some cases, like a fast-moving particle system or a group of uniformly coloured objects such as grass blades, the visual impact of rendering these objects in wrong order may end up being hardly visible to the viewer, and sorting the objects might not be necessary.

Note that order-independent transparency can only work when depth writes are disabled (`glDepthMask(GL_FALSE)`), otherwise transparent objects that are close could incorrectly hide objects further away if rendered first.

# 5. Vertex data

When it comes to passing vertex data to OpenGL ES there are a number of choices to be made:

### 5.1.1. Primitive Type

OpenGL ES 2.0 supports three primitive types based on triangles – triangle lists, triangle strips, and triangle fans – which can be used to render solid objects. While triangle strips, with the help of degenerate connecting triangles, are equally flexible as triangle lists, triangle fans are too limited to be of much use. All three types can be drawn either indexed (using `glDrawElements`) or non-indexed (using `glDrawArrays`).

For best performance on POWERVR SGX you should always draw a mesh as a single indexed triangle list.

### 5.1.2. Interleaved Attributes or Separate Arrays?

There are several ways of storing vertex data in memory. One is to interleave it so all data for one vertex is followed by all data for the next vertex. Another is to keep each attribute in a sequential array, one array after another or in completely separate locations. It is also possible to mix these two approaches. Interleaved attributes can be considered array-of-structs (AoS) while sequential arrays represent a struct-of-arrays (SoA).

In general interleaved vertex data provides better performance because all data required to process each vertex can be gathered in one sequential read which greatly improves cache efficiency.

However if there is a vertex attribute array you want to share between several meshes but don't want to duplicate, putting this attribute into its own sequential array can result in better performance. The same is true if there is one attribute that needs frequent updating while the other attributes remain unchanged.

### 5.1.3. Client Side Arrays or Vertex Buffer Objects?

Vertex Buffer Objects (VBO) are the preferred way of storing vertex and index data. Since their storage is server (driver) managed there is no need to copy in 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 there is an attribute you want to share between meshes. You don't have to create one VBO for every mesh, it's a good idea to group meshes that are always rendered together in order to minimize buffer rebinding.

For dynamic vertex data you should have one buffer object for each update frequency. Don't forget to set the right usage flag (`STATIC_DRAW`, `DYNAMIC_DRAW`, `STREAM_DRAW`) when submitting data or allocating storage with `glBufferData`.

### 5.1.4. Attribute Data Types

OpenGL ES 2.0 offers a choice of six different data types for vertex attributes: `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `FLOAT`, and `FIXED`.

Vertex shaders always expect attributes as float, which means that all types except `FLOAT` 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 and storage requirements, and precision. Find out where the bottleneck in your application is before making a final decision on the attribute data types.

Check your precision requirements carefully. The byte and short types are often sufficient, even for position information. For example, scaled to a range of 10 m the short types give you a precision of 150 µm, more than enough for many meshes. 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.

`FIXED` uses the same bandwidth as `FLOAT`, but requires additional format conversion cycles in the USSE pipeline, thus you should avoid it. The only case where you might want to use `FIXED` is if you are dynamically calculating vertex attributes on a platform without FPU using fixed-point arithmetic and you're CPU limited.

Make sure that you align all attributes to 4 byte boundaries. For attributes that are less than 32 bits wide this may mean you have to add padding bytes. Better yet, try to pack different attribute vectors with less than 4 components together to minimize the space wasted. Furthermore, you could attempt to compress vertex attributes that show a strong correlation.

# 6. Using Textures

## 6.1. Texture Size

A common misconception is that bigger textures always look better. Using a 1024x1024 texture for an object that never covers more than a small part of the screen just wastes storage space. Choose your textures' sizes based on the knowledge of how they will be used. Ideally you would choose the texture size so there is about one texel mapped to every pixel the object covers when it is viewed from the closest distance allowed.

Obviously memory constraints limit the amount of texture data you can use, and it is rarely possible to use all the texture detail you would want. But when you reduce texture sizes to try and squeeze more textures into memory, try to reduce texture resolution uniformly. It is often better to aim for a scene with uniform – though lower – texture detail across all surfaces than to have highly detailed textures clash with blurry ones.

### 6.1.1. NPOT Textures

Non-power-of-two (NPOT) textures are supported by POWERVR SGX to the extent required by the OpenGL ES 2.0 core specification. This means that mipmapping is not supported with NPOT textures, and the wrap mode must be set to CLAMP_TO_EDGE.

The main use of NPOT textures is for screen-sized render targets necessary for post-processing effects. Due to the mipmapping restriction it is not recommended to use NPOT textures for normal texture mapping of meshes.

## 6.2. Texture Formats and Texture Compression

POWERVR SGX offers a variety of texture formats you can choose from. However, all except the basic formats listed in section 6.2.1 require specific OpenGL ES extensions that may or may not be supported on a given implementation based on SGX. You always need to check whether the appropriate extension is listed in the OpenGL ES extension string before you use any of these formats.

Whenever you are accessing multiple identically-sized textures with the same texture coordinates, think about packing them into fewer textures to reduce the number of texture fetch operations. For example, if you have a normal map and a height map, consider packing them together into one RGBA texture, using the RGB components for the normal and A for the height. But take into account that e.g. RGB565 + A8 needs less bandwidth than RGBA8.

The *PVRTexTool* Utility that comes with the POWERVR SDK can be used to convert common image file formats to the texture formats listed below, including texture compression and mipmap generation.

### 6.2.1. Basic OpenGL ES 2.0 Formats

These are the basic 8 bit per channel (bpc) texture formats plus the 16 bit RGB and RGBA formats as mandated by the OpenGL ES 2.0 specification. Note that the RGB8 format is not directly supported by the hardware and will be expanded to RGBA8 at texture upload time.

| Short Name | GL Format | GL Type | bpp |
|---|---|---|---|
| A8 | ALPHA | UNSIGNED_BYTE | 8 |
| L8 | LUMINANCE | UNSIGNED_BYTE | 8 |
| LA8 | LUMINANCE_ALPHA | UNSIGNED_BYTE | 16 |
| RGB8 | RGB | UNSIGNED_BYTE | 32* |
| RGBA8 | RGBA | UNSIGNED_BYTE | 32 |
| RGB565 | RGB | UNSIGNED_SHORT_5_6_5 | 16 |
| RGBA5551 | RGBA | UNSIGNED_SHORT_5_5_5_1 | 16 |
| RGBA4444 | RGBA | UNSIGNED_SHORT_4_4_4_4 | 16 |

These formats are typically used for colours or transparency, but can also represent surface normals, shininess, or any other surface property where a relatively low precision is acceptable.

Texture bandwidth is often an important performance bottleneck. The compressed texture formats listed below require fewer bits per pixel, thus you should always try them first, and only fall back to 16, then 32 bit RGBA formats if the compressed image quality is not acceptable.

### 6.2.2. Float and Half Float Texture Formats

Float and half float textures are only supported given the presence of the extensions `GL_OES_texture_float` and `GL_OES_texture_half_float`, respectively. Some OpenGL ES 2.0 implementations on POWERVR SGX may not support these extensions, so you have to check for their presence in the extension string before using those texture formats.

| Short Name | GL Format | GL Type | bpp |
|---|---|---|---|
| A16f | ALPHA | HALF_FLOAT_OES | 16 |
| L16f | LUMINANCE | HALF_FLOAT_OES | 16 |
| LA16f | LUMINANCE_ALPHA | HALF_FLOAT_OES | 32 |
| RGB16f | RGB | HALF_FLOAT_OES | 48 |
| RGBA16f | RGBA | HALF_FLOAT_OES | 64 |
| A32f | ALPHA | FLOAT | 32 |
| L32f | LUMINANCE | FLOAT | 32 |
| LA32f | LUMINANCE_ALPHA | FLOAT | 64 |
| RGB32f | RGB | FLOAT | 96 |
| RGBA32f | RGBA | FLOAT | 128 |

When using float and half float textures, the sampler variables in the shader used to read from these textures should be declared with mediump (half float) or highp (float) precision modifiers. Remember that these texture formats are very demanding on memory bandwidth, so try to limit their use as much as possible. Clever packing of values into lower precision textures can sometimes work just as well as float textures.

### 6.2.3. Compressed Texture Formats

POWERVR supports a proprietary texture compression format called PVRTC. It boasts very high quality for competitive compression ratios. As with S3TC, this compression is block-based but benefits from a higher image quality than S3TC as data from adjacent blocks are also used in the reconstruction of original texture data (Figure 2). PVRTC supports opaque and translucent textures in both 4 bits-per-pixel and 2 bits-per-pixel modes. This reduced memory footprint is advantageous for embedded systems where memory is scarce, and also considerably minimizes the memory bandwidth requirements (while improving cache effectiveness). For more information about PVRTC please read the POWERVR white paper about PVRTC at http://www.powervr.com/Downloads/Factsheets/PVRTextureCompression.pdf.

**Skybox texture, uncompressed**

**S3TC 4bpp**

**PVRTC 4bpp**

**Figure 2 Comparison of PVRTC and S3TC for a skybox texture**

POWERVR SGX also supports Ericsson Texture Compression (ETC). The ETC format is expected do be supported on a wide range of OpenGL ES 2.0 implementations from multiple vendors, offering good portability. However, it only includes a single 4 bpp RGB format and is thus less flexible than PVRTC.

Both PVRTC and ETC formats require an extension to be present in the extension string. The four PVRTC sub-formats are supported if the extension `GL_IMG_texture_compression_pvrtc` is present, while ETC requires the extension `GL_OES_compressed_ETC1_RGB8_texture` in the extension string.

We highly recommended that you use the *PVRTexTool* utility to generate compressed versions of all your textures and compare them to the originals to check whether the compression quality is acceptable.

Compressed textures have to be passed to the GL using `glCompressedTexImage2D`. They are not renderable, and they do not support mipmap generation. So if you want to use mipmaps you need to generate them off-line and pass each level to `glCompressedTexImage2D`.

| Short Name | GL Base Internal Format | GL Compressed Format | bpp |
|---|---|---|---|
| PVRTC2 | RGB | COMPRESSED_RGB_PVRTC_2BPPV1_IMG | 2 |
| PVRTC2 | RGBA | COMPRESSED_RGBA_PVRTC_2BPPV1_IMG | 2 |
| PVRTC4 | RGB | COMPRESSED_RGB_PVRTC_4BPPV1_IMG | 4 |
| PVRTC4 | RGBA | COMPRESSED_RGBA_PVRTC_4BPPV1_IMG | 4 |
| ETC | RGB | ETC1_RGB8_OES | 4 |

### 6.2.4. How Texture Formats Affect Shaders

All reads from RGB and RGBA basic or compressed texture formats as listed above can be passed as a `lowp vec4` value from the Texture Co-processor to the USSE pipeline. Thus these formats are interchangeable. However, *float*, and *half float* textures need higher precision and may return fewer components. So when you change the texture format bound to a sampler variable across those groups, the driver may have to patch the shader code to accommodate for the changed register requirements. You should try to avoid this overhead by designing your shaders with the texture formats you're going to use in mind.

Because float texture formats require higher precision, you need to declare sampler variables with the `mediump` (half float) or `highp` (float) precision modifiers for the shader to work correctly when using these formats. Additionally you should use only those components you really need in the shader and mask out the rest. The latter is also true for LUMINANCE and ALPHA formats: Only use those channels that actually exist in the texture you plan to use.

For example, the following GLSL ES snippet shows how you should access an RGB32f texture in a shader:

```
uniform highp sampler2D sFloatTexture; // highp to sample from float texture

varying highp vec2 TexCoords;

void main()
{
	highp vec3 color = texture2D(sFloatTexture, TexCoords).rgb; // use RGB only
	// ...
}
```

## Use Mipmaps!

Mipmaps are smaller, pre-filtered variants of a texture image, representing different levels-of-detail (LOD) of a texture. By using a minification filter mode that uses mipmaps the hardware can be set up to automatically calculate which level-of-detail comes closest to mapping one texel of a mipmap to one pixel in the render target, and use the according mipmap for texturing. This automatic selection only works in fragment shaders, though. In vertex shaders you need to calculate the level-of-detail parameter yourself and pass it to the `texture2DLod` GLSL ES function.

Using mipmaps has two important advantages. It increases performance by massively improving texture cache efficiency, especially in cases of strong minification. At the same time using pre-filtered textures improves image quality by countering aliasing that would be caused by severe under-filtering of the texture content. This aliasing usually shows itself as a strong shimmering noise on minified textures.

The one drawback of mipmaps is that they require about 1/3 more texture memory, but this cost can often be considered minor compared to the gains. There are some exceptions where mipmaps should not be used, though. This includes any kind of textures where filtering can't be applied sensibly, such as textures containing indices. Also, textures that are never minified – e.g. 2D elements such as fonts or a HUD which are mapped 1:1 to screen pixels – do not need mipmaps.

Mipmaps can be created off-line with a utility such as *PVRTexTool* which can be found in the POWERVR SDK. Alternatively you can save the file storage space for mipmaps and generate them at runtime in your application. The OpenGL ES function `glGenerateMipmap` will perform this task for you, but it will not work with PVRTC or ETC textures. Those should always be off-line generated. This function is also useful when you want to update mipmaps for render target textures.

In combination with certain texture content, especially with high contrast, the lack of filtering *between* mipmap levels can lead to visible seams at mipmap transitions. This is called *mipmap banding*. Trilinear filtering (GL_LINEAR_MIPMAP_NEAREST) can effectively eliminate these seams and thus achieve higher image quality. However this quality comes at a cost, as outlined below in section 6.6.1.

### 6.2.5. Mipmap Selection Caveats

In all modern GPUs the level-of-detail parameter which determines the mipmap level selection is calculated based on the difference in texture coordinates for a 2x2 fragment block. The same mipmap level is then used for all four fragments. This is never a problem for non-dependent texture reads, or for texture reads based on linearly modified varyings. It can, however, lead to blocking artefacts when the texture coordinates are the result of a highly non-linear function, especially if there are sudden changes in the curvature of that function.

Fixing these block artefacts can be tricky, and usually it is easiest to try to limit the frequency of the texture coordinate function. When the texture coordinate function itself depends on a texture lookup, as is often the case with techniques such as environment mapped bump mapping (EMBM) this can easily be achieved by a little positive LOD bias applied to the first texture lookup. The bias is an optional parameter to the GLSL ES built-in function `texture2D`.

## 6.3. Texture Upload

When you upload textures to the OpenGL ES driver via `glTexImage2D`, the input data is usually in linear scanline format. Internally, though, POWERVR SGX uses a twiddled layout (i.e. following a plane-filling curve) to greatly improve memory access locality when texturing. Because of this different layout uploading textures will always require a somewhat expensive reformatting operation, regardless of whether the input pixel format exactly matches the internal pixel format or not.

For this reason we recommend that you upload all required textures at application or level start-up time in order to not cause any framerate dips when additional textures are uploaded later on.

You should especially avoid uploading texture data mid-frame to a texture object that has already been used in the frame.

## 6.4. Render to Texture

OpenGL ES 2.0 offers two ways of rendering to textures. One is using EGL pbuffer surfaces; the other is attaching a renderable texture to a framebuffer object (FBO). The latter approach is the recommended one for POWERVR SGX.

Make sure that you always clear the colour and depth buffers when rendering to a texture.

## 6.5. Math lookups

Sometimes it can be a good idea to encode the results of a complex function in a texture and use it as a lookup table instead of performing the necessary calculations in a shader. This is a trade-off between shader workload, bandwidth and memory usage, and precision.

Before you attempt to use a math lookup texture, always make sure that you're actually shader limited. For small functions using a lookup table is rarely worth the effort. For example, replacing 3D vector normalization with a normalization cube map lookup may not help performance at all.

If the function parameters, i.e. the texture coordinates, vary wildly between adjacent fragments, texture cache efficiency suffers. Using mipmaps as mentioned above in section 0 will help improve this somewhat, but the loss of high frequency detail, accompanied by a loss of accuracy, may or may not be desirable, depending on the function represented. As mipmap generation is easy, you can always compare both methods and pick the best one.

## 6.6. Texture Sampling Performance

Typically texture sampling performance heavily depends on the memory bandwidth available. However, there are other influences which limit the rate at which SGX's Texture Co-processor can fetch and filter texture samples.

Texturing performance of POWERVR SGX scales with the number of texture units. As this number varies between different members of the SGX family, the following paragraphs describe performance on a per-texture-unit basis.

### 6.6.1. Trilinear Filtering

All basic texture filtering modes which do not blend between two mipmap levels, including the commonly used bilinear filtering with mipmaps (GL_LINEAR_MIPMAP_NEAREST), can typically be performed taking only one cycle.

Trilinear filtering (`GL_LINEAR_MIPMAP_LINEAR`) can effectively reduce so-called *mipmap banding*, i.e. visible seams along mipmap transitions, and thus improve image quality. However it comes at a performance cost: Whenever two mipmap levels need to be blended together the texture unit needs an additional cycle to fetch and filter the required data. In the worst case, trilinear filtering achieves half the texturing performance compared to bilinear filtering. If you are using complex math-heavy shaders however, the texture unit may have cycles to spare and you can get trilinear filtering for a low bandwidth cost.

### 6.6.2. Dependent Texture Reads

Dependent texture reads are those in which the texture coordinates depend on some calculation in the shader instead of being taken directly from varying variables. Vertex shader texture lookups always count as dependent reads.

Dependent reads require two USSE cycles. SGX variants with an equal number of USSE and texture units may therefore be USSE limited when performing dependent reads. In this case the processing cost of trilinear filtering can be masked most of the time.

### 6.6.3. Wide Floating Point Textures

For textures which exceed 32 bits per texel, each additional 32 bits can be counted as a separate texture read. This applies to *half float* textures with 3 or 4 components as well as to *float* textures with 2 or more components. This means that e.g. a dependent read from an RGB float texture will require 6 cycles to complete.

# 7. Shaders

Shaders are the core feature of OpenGL ES 2.0, offering an enormous flexibility to application developers.

## 7.1. Algorithms and Shader Length

For complex shaders that run for more than a few cycles per invocation, picking the right algorithm is usually more important than low-level optimizations. Doing research on the latest and greatest shader techniques can be a time-consuming task. To help you with this, the *ShaderX* series of books features plenty of articles that are an excellent source of algorithms and ideas for amazing shader effects. The same is true for the *GPU Gems* series. However, many of these examples target high-end PC hardware and need adaptation, not just porting to GLSL ES but also a reduction in complexity, to be useful for mobile and embedded devices.

Keep in mind that these devices are not designed to handle the complex shaders used in the latest PC and console games at full framerates. Because the number of GLSL ES lines is not always indicative of the number of USSE instructions generated by the compiler for that shader, it is hard do give a recommendation based on shader source length. Future versions of the *PVRUniSCo* offline shader compiler will be able to return instruction statistics for each GLSL line. To find the best balance between shader complexity and performance you will have to benchmark your shaders.

## 7.2. Choosing the Right Precision

To achieve high throughput in a variety of tasks, POWERVR SGX was designed with support for multiple precisions. GLSL ES 1.00 supports three precision modifiers that can be applied to float and integer variables. Because choosing a lower precision can increase performance but also introduce precision artefacts, finding the right balance is very important. The safest method for arriving at the right precision is to start with `highp` for everything (except samplers) and then gradually reduce precision according to the following rules until the first precision problems start to appear.

**highp**

Float variables with the `highp` precision modifier will be represented as 32 bit floating point values, adhering to the IEEE754 single precision standard with a few exceptions. 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, as well as any scalar calculations that use complex built-in functions such as `sin`, `cos`, `pow`, `log`, etc.

With `highp` precision, the USSE pipeline acts on scalar values.

**mediump**

Variables declared with the `mediump` modifier are represented as 16 bit floating point values (using a sign bit, 5 exponent bits and 10 mantissa bits), 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 is can be very useful for texture coordinate varyings.

With `mediump` precision, the USSE pipeline acts on two-component vectors.

**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 with a precision of 1 / 256. The USSE pipeline processes lowp variables as 3- or 4-component vectors. The `lowp` precision is useful mainly for representing colours and any data read from low precision textures, such as normals from a normal map. Be careful not to exceed the numeric range of `lowp` floats, especially with intermediate results. Swizzling the components of `lowp` vectors is expensive and should be avoided.

## 7.3. Attributes

Attributes are the per-vertex inputs to a vertex shader and should normally use `highp` precision. Each attribute that is fed from a vertex attribute array in memory requires bandwidth. To minimise the bandwidth required, pack attributes together and choose their data types as outlined in section 5.1.4.

POWERVR SGX supports up to 8 vec4 attributes in a vertex shader.

## 7.4. Varyings

Varyings represent the outputs from the vertex shader which are interpolated across a triangle and then fed into the fragment shader. Interpolation of varyings is always perspective correct. Each varying variable requires memory space in the parameter buffer and processing cycles for interpolation. Always try to use as few varyings as possible. By choosing a lower precision for varyings you can reduce the space and memory bandwidth required to store the whole scene in the parameter buffer. `mediump` may be sufficient for texture coordinates, especially if the textures are relatively small, while colours and normals can usually be stored as `lowp` `vec3`/`vec4`.

POWERVR SGX supports up to 8 varying vectors between vertex and fragment shaders.

## 7.5. Samplers

Samplers are used to sample from a texture bound to a certain texture unit. The default precision for sampler variables is `lowp`, which is exactly what you need in most cases. There are only a few exceptions to this where you need to explicitly declare a higher precision: If you intend to use a sampler to read from a *float* texture, that sampler should be declared with `highp` precision. Similarly, for a *half float* texture you should choose `mediump`. This is explained in more detail in section 6.2.4.

POWERVR SGX supports up to 8 texture samplers in both vertex and fragment shaders.

## 7.6. 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, try to avoid redundant uniform updates in between draw calls. Unlike attributes and varyings, uniform variables can be declared as arrays. Be careful with the number of uniforms you use, though. 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.

POWERVR SGX supports up to 512 uniform scalars in the vertex shader and up to 64 uniform scalars in the fragment shader.

### 7.6.1. Uniform Calculations

The shader compiler for POWERVR SGX is able to extract calculations based on uniforms from the shader and perform these calculations once per draw call in the USSE pipeline. This can be very handy on platforms with weak floating point support and allows you to offload operations such as matrix-matrix multiplications from the CPU. Make sure the order of operations is chosen so the uniforms are processed first, though:

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

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

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

## 7.7. Scalar Operation

When using highp precision SGX operates on scalars, not vectors. This means that you can achieve higher efficiency, but different optimisation rules apply. For example, on a vector-based architecture you try to vectorise code as much as possible. This can in fact be detrimental for a scalar architecture. Do not vectorise computations that are naturally scalar. Be careful with the order of operations when you mix scalar and vector computations. Try to keep it scalar as long as possible. For example, when you calculate the product of two scalar values and a vector, multiply the two scalars first.

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

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

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

### 7.7.1. Sparse Matrices

If you already know that many elements of a transformation matrix are 0, don't perform a full matrix transform. For example, a typical projection matrix looks like this:

| | | | |
|---|---|---|---|
| *A* | 0 | 0 | 0 |
| 0 | *B* | 0 | 0 |
| 0 | 0 | *C* | *D* |
| 0 | 0 | *E* | 0 |

So if for some reason you had to transform your vertex position to view space already, another full matrix transform would be a waste of cycles. In fact, since dividing the matrix by a positive constant will not change the transformation result in homogeneous coordinates, it is sufficient to store just four values.

Similarly, non-projective transformation matrices usually have the fourth row fixed at (0, 0, 0, 1) which means you don't have to store that row at all. Instead you can just store three vec4 rows and replace 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;
}
```

Note that if D is negative, you need to negate the projection vector and subtract 1.0 from gl_Position.z instead in order to avoid negative W values.

## 7.8. 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 (consisting only of rotations, translations, and mirroring) you can perform lighting and similar calculations 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. However, if you have to use a certain space, e.g. for cubemap reflections, it's often best to use this single space throughout. If you use view space, remember to use the method presented in section 7.7.1 for efficient projection to clip space.

## 7.9. Flow Control

POWERVR SGX offers full support for flow control constructs (`if-else`, `for` and `while` loops) in both vertex and fragment shaders. You can use these without explicitly enabling a language extension in GLSL ES.

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 shader. This is not generally a win, though, and you should benchmark thoroughly when deciding whether to put multiple paths into one shader.

Using dynamic branching in a shader has a certain, non-constant overhead that depends on the exact shader code. Therefore using dynamic branching is not always a performance win. The branching granularity on SGX is one fragment or one vertex. This means you don't have to worry a lot about making branching coherent for an area of fragments. Apply the following rules when deciding where to apply conditional execution:

- Make use of conditionals to skip unnecessary operations when the condition is met in a significant number of cases.
- If you have to calculate the product of two rather complex functions that sometimes evaluate to 0, use the one that is less complex and more often returns 0 as a condition for executing the other.
- Don't unroll loops manually. The compiler knows when loop unrolling and predication is cheaper than a jump.
- Don't use texture lookups without explicit LOD in a conditionally executed part of a shader

## 7.10. Discard

The GLSL ES fragment shader operation `discard` can be used to stop fragment processing and prevent any buffer updates for this fragment. It provides the same functionality as the fixed function alpha test in a programmable fashion.

On POWERVR SGX `discard` is an expensive operation because it requires a fragment shader pass to accurately determine visibility of fragments. This visibility information then has to be fed back to the Image Synthesis Processor (ISP) before continuing to perform the depth and stencil test for other polygons.

For this reason you should avoid `discard` whenever possible. Often the same visual effect can be achieved using the right alpha blend mode and forcing the alpha value to 0 where `discard` would be used. If you really need to use `discard`, make sure that you render objects using this shader after all opaque objects have been submitted.

## GL State That Affects Shader Execution

In addition to the vertex and fragment shaders, some operations which are considered “fixed-function” by the OpenGL ES specification are also executed by POWERVR SGX’s USSE pipeline. The instructions necessary to perform these operations are generated by the driver at draw call time and injected into the USSE code generated by the shader compiler. This means that these operations may cost additional shader cycles.

### 7.10.1. Vertex Attribute Type Conversion

The only vertex attribute data type in OpenGL ES 2.0 that directly maps to one of the three precision types POWERVR SGX supports in shaders is `FLOAT`. All other types (`BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `FIXED`) need to be converted to float before the vertex shader can use them. Where required, the driver inserts the appropriate conversion instructions at the start of a vertex shader, depending on the set of active attribute arrays at draw call time. Section 5.1.4 describes how to choose the right attribute data type.

### 7.10.2. Texture Formats

Always make sure that the precision with which you have declared a sampler variable matches the texture format you are using. This means that basic or compressed texture formats should always be used with a `lowp` sampler, while half float textures need a `mediump` sampler and float textures should be accessed using a `highp` sampler. Not following these guidelines will result in inefficient shader execution because the driver will have to add conversion instructions to the shader. This is further explained in section 6.2.4.

### 7.10.3. Framebuffer Blending and Colour Mask

Framebuffer blending and colour masking are also performed by the USSE pipeline and thus may add to the fragment shader execution cost. Sometimes these operations can be merged with the last fragment shader instructions in which case there is no additional cost.

## 7.11. Dos and Don’ts

To recap this section on shaders, here is a short list of tips that should help you write optimised shaders for POWERVR SGX.

- Move calculations that can be performed per vertex, such as linear transformations on varyings, from the fragment shader to the vertex shader
- Perform per-vertex calculations in model space if possible, then transform straight to clip space
- Keep your fragment shaders short
- Use `highp` for vertex position and transformation matrices
- Use `highp` or `mediump` for texture coordinates
- Use `lowp` for normals and colours as long as the range is sufficient
- Declare samplers with the precision that matches the texture format
- Use conditionals to skip unnecessary computations
- Don’t unroll loops
- Don’t try to vectorize scalar `highp` calculations
- Don’t perform a full matrix-vector multiplication with sparse matrices
- Don’t use `discard`
- Don’t swizzle `lowp` vectors

# 8. Depth Buffer

Unlike many other GPUs, POWERVR SGX uses a 32 bit floating point depth buffer. When used correctly, it can provide a much higher depth precision than commonly used 24 bit fixed point buffers and solve many Z-fighting problems.

### Inverting Viewport Depth

Typical projection matrices – e.g. like the ones generated by `glFrustum` or `gluPerspective` – in combination with the default viewport depth range of [0, 1] (which can be set using `glDepthRange`) map view space positions at a near Z plane distance to a window Z coordinate of 0, while positions at a far Z plane are mapped to 1. While this mapping is ok for a fixed point depth buffer, it is less than ideal for the floating point depth buffer in POWERVR SGX. For the latter an inverted mapping (near → 1, far → 0) is much more desirable. Note that, since in a fixed point format the representable values are distributed evenly, an inverted mapping does not adversely affect precision on other hardware.

This inverted mapping can be achieved in multiple ways. One is to simply invert the viewport Z axis by calling:

```
glDepthRangef(1, 0);
```

Another way is to negate the Z component of the `gl_Position` output variable. If your vertex shader is using a projection matrix to calculate the clip space position this can be folded into the matrix by negating its third row.

Along with the inverted mapping, you need to invert the depth compare function from the default `LESS` to `GREATER` (or `GEQUAL`), and to negate any parameters passed to `glPolygonOffset`.

### Infinite View Frustum

Because of the logarithmic distribution of values in the floating point format, it is actually possible to generate a projection matrix with the far clip plane moved “to infinity” and still get a high relative precision of depth values. Doing this and negating the third matrix row results in the following projection matrix, where n is the near plane distance and width and height represent the size of the view frustum at distance n.

| | | | |
|---|---|---|---|
| $\frac{2n}{width}$ | 0 | 0 | 0 |
| 0 | $\frac{2n}{height}$ | 0 | 0 |
| 0 | 0 | 1 | $2n$ |
| 0 | 0 | -1 | 0 |

To get the highest depth precision it is still crucial that you place the near clipping plane as far out as is possible without causing clipping artefacts.

The static method `Mat4::PerspectiveFloatDepth` which is part of the PVRTools library in the POWERVR OpenGL ES SDK can be used to generate such a projection matrix. Note that this matrix contains just three variable components, so if you’re using a separate projection matrix in a vertex shader follow the guidelines presented in section 7.7.1.

# 9. Target a fixed framerate

It is important to understand that the smoothness of animation is largely determined by the lowest framerate and that high peaks and average framerates do not serve to even out stuttering and dips to 10 fps or less. As those maximum fps do not contribute much to the perceived smoothness, it makes sense to limit the framerate to a fixed value that can be achieved at all times and let the hardware idle and save power when the workload is low.

Most LCDs are updated at a rate of around 60 Hz. By setting a swap interval of 2 using `eglSwapInterval`, you can limit the framerate to a maximum of ~30 fps which is often considered sufficient for smooth animation for all but the fastest moving games.