BROADCOM®

OpenGL ES Quick Start Guide

Table of Contents

Chapter 1: Introduction

1.1 Overview

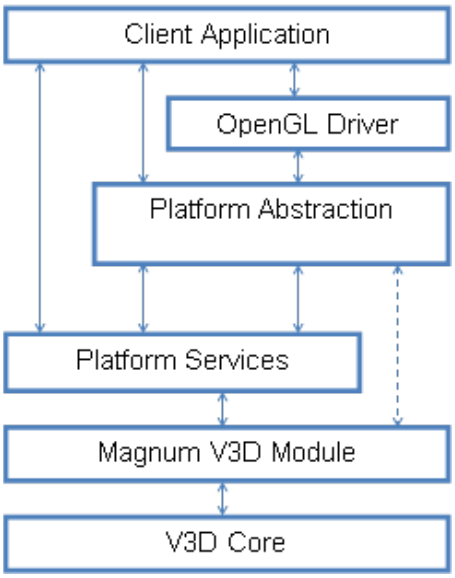
OpenGL ES is an API for 3D graphics hardware. It is designed specifically with embedded systems in mind, and can be used to generate color images of two and three-dimensional objects at high frame rates. OpenGL ES is a subset of the OpenGL APIs commonly available on desktop devices such as PCs and, as such, many of the references and books which describe OpenGL can also be useful for OpenGL ES.

OpenGL ES is supported on Broadcom platforms containing a VideoCore block. VideoCore IV based devices support OpenGL ES 1.1 and 2.0 whereas VideoCore V based devices support OpenGL ES 1.1, 2.0 and 3.X.

OpenGL ES 1.1 is a legacy API designed for low-end devices which have a fixed-function 3D pipeline. The newer versions, OpenGL ES 2.0 and OpenGL ES 3.X, are more flexible and allow developers to implement their own custom graphics pipelines via shader-programs. For best performance, flexibility and forwards compatibility, we recommend that developers adopt OpenGL ES 2.0 or above.

All versions of OpenGL ES are device and platform independent. The GL does not export APIs to obtain system surfaces or to select display configurations. However, bundled with the OpenGL ES drivers is another API called EGL which, in collaboration with the native OS, does provide these services.

This document describes how to use the OpenGL ES implementations with Nexus and should be read in conjunction with the examples described below. The OpenGL ES driver is not part of the main Nexus build. It consists of two components: a platform layer and the driver. The platform layer provides a bridge between the platform-independent driver and Nexus. The platform layer in the distribution is a default implementation which should be sufficient for most applications. However, it is possible to override the platform layer. This is an advanced topic beyond the scope of this document. The Magnum V3D module interfaces with the hardware and marshals access from multiple processes.



1.2 Further Reading

OpenGL ES is an industry standard maintained by the Khronos Group. They provide information about each of the APIs including their specifications, “man” pages and extension registries.

- See www.khronos.org for up-to-date specifications and documentation on OpenGL ES and the other Khronos APIs.

1.3 Locations

The OpenGL ES examples and driver code are held in the "BSEAV" source tree. In this section we give all paths relative to the folder in which "nexus" is located. We will be mostly concerned with files in the following folders:

- *BSEAV/lib/gpu/applications/nexus* contains the example applications.
    - We will call this folder $NEXUS_APPS in the rest of this document.
- *BSEAV/lib/gpu/v3d/* contains the driver code for VideoCore IV. In particular:
    - *BSEAV/lib/gpu/v3d/driver/interface/khronos/include* contains the OpenGL ES and EGL header files.
    - *BSEAV/lib/gpu/v3d/platform/nexus* holds the platform layer for Nexus.
- *BSEAV/lib/gpu/vc5* contains the driver code for VideoCore V. In particular:
    - *BSEAV/lib/gpu/vc5/driver/interface/khronos/include* contains the OpenGL ES and EGL header files
    - *BSEAV/lib/gpu/vc5/platform/nexus* holds the platform layer for Nexus.
- *obj.<chip>.<rev>.<cfg>/nexus/bin* is the folder into which your binary files will be compiled. This includes the executable for the examples as well as the shared libraries for the driver and platform layer. We will call this directory $NEXUS_BIN in the rest of this document.
    - <chip> is the board that your are building for e.g. 97278
    - <rev> is the chip revision building for e.g. B0
    - <cfg> is the configuration of the board e.g. SFF, 32 or 64

1.4 Example: Cube

To run a 3D example, you must first build nexus, and then the platform and OpenGL ES driver shared libraries. The supplied Makefile automates this process for you.

The cube example creates and draws a simple cube and rotates it. If you have not already run the platform script for your board, then:

```
source BSEAV/tools/build/plat <chip> <rev> <cfg>
```

For example:

```
source BSEAV/tools/build/plat 97278 B0 32
```

To build a release version of the cube example follow these steps:

```
cd $NEXUS_APPS/cube
unset NEXUS_MODE
unset NXCLIENT_SUPPORT
make DEBUG=n
```

This will build the three shared objects:

- Nexus library: nexus.so

- OpenGL ES driver: libv3ddriver.so
- Platform layer: libnxpl.so

They will be copied into $NEXUS_BIN. To run the application on the board use the “nexus” script thus:

```
cd $NEXUS_BIN
nexus cube
```

The “nexus” script installs the appropriate Nexus kernel modules, sets LD_LIBRARY_PATH and PATH, and then invokes the application.

The cube example can be run at different resolutions and with different screen depths. So, for example, to run at 720p, 32 bits-per-pixel, multisampled:

```
nexus cube d=1280x720 bpp=32 +m
```

To build a debug version of the cube application:

```
cd $NEXUS_APPS/cube
make DEBUG=y
```

Note that the performance of the debug version will be significantly degraded.

1.4.1 Kernel Mode

In Nexus kernel mode, the bulk of the Nexus implementation is built into a kernel module. Applications run in “proxy” mode, which means that API calls to Nexus are mapped to IOCTL calls into the kernel. To build the application in Nexus kernel mode:

```
cd $NEXUS_APPS/cube
export NEXUS_MODE=proxy
unset NXCLIENT_SUPPORT
make clean
make DEBUG=n
```

To run:

```
cd $NEXUS_BIN
nexus cube
```

As before the “nexus” script will insert the Nexus kernel module and set up LD_LIBRARY_PATH and PATH automatically.

1.5 Other Examples

The examples folder contains the following applications:

- cube, cube_pixmap and cube_s3power
- earth_es2
- benchmark, poly_rate and poly_rate_es1
- eglimage
- dice
- font_es2
- video_texturing

All these examples can be built in the same way as for “cube”.

Chapter 2: Basic Usage

When creating new applications, the Makefiles from the examples can be used as a basis. To help understand the build process, the following sections describe the include files and libraries needed to build OpenGL ES applications.

2.1 Include Files

Cube is a very simple OpenGL ES 2.0 example, but it can serve as a template for more complex applications. All OpenGL ES applications need to include the appropriate headers:

For EGL:

```
#include <EGL/egl.h>
```

For OpenGLES 1.1:

```
#include <GLES/gl.h>
```

For OpenGL ES 2.0:

```
#include <GLES2/gl2.h>
```

For OpenGL ES 3.X:

```
#include <GLES3/gl3.h>  // For ES 3.0 or
#include <GLES3/gl31.h> // For ES 3.1
```

Applications which use extensions to the OpenGL ES API will also need to include the appropriate extension headers e.g.

```
#include <GLES2/gl2ext.h>
```

or for OpenGL ES 1.1:

```
#include <GLES/glext.h>
```

OpenGL ES applications on a Nexus platform will also need to include the platform layer header:

```
#include "default_nexus.h"
```

Make sure that your compile flags specify the include paths for the OpenGL ES, EGL and the Nexus platform layers as detailed in Section 1.3.

2.2 Libraries

In addition to Nexus, OpenGL ES applications will need to link against:

- *m* – the math library
- *rt* – the Posix real-time extensions library
- *pthread* – the Posix thread library
- *v3ddriver* – the OpenGL ES and EGL driver
- *nxpl* – the nexus platform layer for OpenGL ES

2.3 Lifecycle of an Application

A typical Nexus OpenGL ES application follows the steps summarized below:

- Initialize a Nexus platform and display.
- Register the display with the platform layer.
- Create a native window.
- Initialize EGL and create surfaces.
- Create an OpenGL ES context and make it current.
- Initialize OpenGL ES including compilation and linking of shaders.
- For each frame:
  - Clear buffers.
  - For all geometry
    - Set up GL State.
    - Submit geometry.
    - Reset GL state.
  - Swap buffers.
- Finalize GL. Deleting all objects.
- Release the OpenGL ES context and finalize EGL.
- Destroy the native window.
- Unregister the Nexus display from the platform layer.
- Close the display and finalize nexus.

For details on creating, configuring and closing the Nexus display, refer to the Nexus documentation in nexus/docs or refer to the example code. Once a nexus display has been created, it should be registered with the OpenGL ES nexus platform layer via:

```
NXPL_PlatformHandle nxpl_handle = 0;
NXPL_RegisterNexusDisplayPlatform(&nxpl_handle, nexus_display);
```

This tells the OpenGL ES driver that we are giving it exclusive use of the display. The application should not use the display for anything else until it is unregistered from the platform layer. Next, we can use the display to create a “native window” which is used to describe to the EGL the properties of the rendering region:

```
NXPL_NativeWindowInfo   win_info;
win_info.x      = 0;
win_info.y      = 0;
win_info.width   = WIDTH;     /* Width of window       */
win_info.height  = HEIGHT;    /* Height of window      */
win_info.stretch = STRETCH;   /* Stretch to fit display */

native_window = NXPL_CreateNativeWindow(&win_info);
```

In the next step, we need to configure the EGL. This is handled in “cube” by the function InitEGL(). It is a mechanical process, which will look the same in most applications. Using EGL, the application can request a “config” with a certain number of bits for the color, depth and stencil buffers. It can also request multisampling. EGL will return a number of configs matching the application requirements, and the application is responsible for selecting the configuration that most closely reflects its requirements.

Once a configuration has been selected, the application can create one or more window surfaces using `eglCreateWindowSurface()` passing in the native window created above. Then a GL context is created using `eglCreateContext()` and made current using `eglMakeCurrent()`.

The application can now issue GL calls to clear buffers and render geometry. Calling OpenGL ES functions when there is no context will lead to application failure.

There are two EGL functions which deserve special mention here:

- eglSwapBuffers() swaps out the currently displaying frame and replaces it with the next frame. This should be done at the end of every rendered frame. This operation does not block and does not involve any data copies.
- eglSwapInterval(egl_display, N) specifies to the EGL that the display should not be updated more than once every N vertical sync periods. The default is N = 1 which allows for a display update every vertical sync. If an application cannot deliver results at this speed, then N = 2 or greater might help to smooth the animation. N = 0 will run the graphics at full speed without waiting for vertical syncs. This is useful for benchmarking and performance measurements, but leads to frame tearing and is not desirable for production code.

Chapter 3: Advanced Usage

This chapter describes some more advanced use cases which go beyond simple stand-alone 3D applications.

3.1 Mixing 2D and 3D

In many cases, it is worth considering migrating 2D operations into the 3D domain. The 3D hardware can efficiently render 2D content. You might also want to run separate 2D and 3D applications together simultaneously, each application with its own 'window' in the display area.

There are several ways in which 2D and 3D rendering can be mixed using OpenGL and Nexus. We recommend using Nexus surface compositor through the NxClient library as the most flexible solution. This is also the route which requires the least development effort since much of the functionality is already provided for you.

Other possibilities include rendering to an off-screen pixmap and blitting that pixmap in 2D (see cube_pixmap). You could also write a custom platform layer that does composition of the 3D rendered buffers.

3.2 CPU access to GPU data

To access what you have just rendered, the temptation is to call glReadPixels(). Unfortunately this function is almost always written for test purposes rather than used as a critical component of your draw loop.

glReadPixels() is slow as it forces implicit flushes through the pipeline, which could lead to trips to RAM with client/server synchronization. For most vendors, it will imply a conversion from an internal GPU format to a CPU raster format which is slow.

A much cheaper method to access pixel data is via pixmap rendering. A pixmap has the native image format of the windowing system, which should always be suitable for CPU access.

Pixmaps are off-screen surfaces which can be rendered into, independently of the display, and which are usable by the native platform. Hence, they can be used to render 3D imagery which can then be composited into a 2D display. The main disadvantage of a pixmap surface is the need to synchronize the 3D and 2D drawing. In this example, this is done by executing a glFinish() at the end of 3D rendering. This is a blocking API and GL will wait until all rendering has completed before returning.

An alternative to glFinish() is to use "fences". Fences are placed into the command stream and are signalled when they and all preceding commands have completed and all their effects realised e.g. changes to the framebuffer. Applications can wait on fences to determine if they have been signalled. See the "video_texturing" example for more information.

NOTE: if an application wishes to render to an off-screen buffer and use the results exclusively within OpenGL ES, for example as a texture, then it should use "framebuffer objects" and not pixmaps. Pixmaps are held in a CPU friendly format whereas framebuffer objects use the native GPU format.

3.3 Multi-process

The cube example above was built in "exclusive display" mode. In this case, the application has exclusive access to the display and no other applications will be able to use it. OpenGL ES applications can also be built to run in "multi-process" mode where multiple applications communicate to a compositor (by default "nxserver") with each application appearing in its own "window".

To build the client and "nxserver" app:

```
cd $NEXUS_APPS/cube
unset NEXUS_MODE
export NXCLIENT_SUPPORT=y
make clean
make
```

To run the applications under the server on the board:

```
cd $NEXUS_BIN
nexus nxserver &
```

To run the multi-process example applications e.g.

```
nexus.client cube d=300x300 &
nexus.client earth d=300x300 o=400x300 &
```

3.3.1 Kernel Mode

To build the server application:

```
export NEXUS_MODE=proxy
export NXCLIENT_SUPPORT=y
cd $NEXUS_APPS/cube
make clean
make DEBUG=n
cd $NEXUS_APPS/earth_es2
make DEBUG=n
```

To run the server application:

```
cd $NEXUS_BIN
nexus nxserver &

nexus.client cube d=200x200 o=400x100 &
nexus.client earth d=350x350 &
```

Chapter 4: Performance

4.1 Introduction

This section describes some general guidelines for maximizing performance from Broadcom's VideoCore IV and VideoCore V GPU based 3D graphics solutions when using OpenGL ES. We assume some working knowledge of 3D graphics and the OpenGL ES family of APIs.

4.2 General Principles

Performance problems for 3D applications can usually be attributed to one of:

- Using too much GPU or CPU compute
- Using too much memory bandwidth
- Wasting GPU or CPU cycles
- Hitting bottlenecks

Memory bandwidth is typically heavily impacted by:

- Reading textures
- Reading and writing framebuffers
- Copying vertex and image buffers

Identifying the performance barriers in an application is the key to optimization. If the GPU is lightly loaded, then efforts need to be concentrated on the CPU and vice versa.

Use standard tools for monitoring CPU usage e.g. "top" on a Linux platform will show CPU loading. The Broadcom GPUMonitor tool is an invaluable help in analyzing the behavior of the GPU. It can help identify where your program is bottlenecked to ensure that your optimization efforts are best directed.

4.3 Programming Guidelines

The following hints and guidelines should help you to maximize the performance of your OpenGL ES application on the GPU. Not all of them are orthogonal, and to maximize performance, it may be necessary to make some trade-offs.

4.3.1 Rendering Order

As a rule, draw scenes in front-to-back order. The GPU can avoid executing fragment shaders if a nearer fragment has already been written to that pixel. This is called early-z culling.

Transparent objects, however, generally need to be drawn in a back-to-front order for correct results. If the scene contains transparent or blended objects, then draw all the opaque objects first in front-to-back order, and then submit the transparent objects back-to-front.

Note that even a 2D application can take advantage of this optimization. For example, a set of “windows” with a particular z-order could be rendered front-to-back assigning a unique depth value to each window depth.

It can sometimes be useful to split objects into their opaque and transparent components. Continuing the window example, it might be useful to have an opaque inner region drawn separately to a transparent window border.

One caveat to this recommendation is that the cost of sorting your scene into depth order must not be so high as to outweigh the benefits of doing so. Moreover, in general, arbitrarily shaped 3D objects cannot be depth sorted.

4.3.2 Vertex Buffer Objects (VBO)

When drawing objects, the data can be sourced either from client-side buffers or from vertex-buffer objects (VBOs). Client side data is copied every time the object is drawn (because the application might change the data and the API has no way to know if this has happened), whereas vertex buffer objects are copied once at submission time and retained in device memory for re-use in subsequent draw commands. Data in a VBO can still be changed, but you need to tell the API when you are doing so.

Typically the objects in a scene change relatively infrequently, so it makes sense to submit the data for the objects once, and only update it when required. Many applications can work with a relatively small set of prototype objects which can be instantiated at different sizes and orientation according to parameters (e.g. transformation matrixes) passed to the shader programs. Different shader programs can be used to create different effects and appearances.

The GPU works best when vertex data is presented in an interleaved format. For example laid out in memory as:

Vertex 1	X
	Y
	Z
	U
	V
Vertex 2	X
	Y
	Z
	U
	V
Vertex 3	X
	…

4.3.3 Minimizing Vertex Counts

OpenGL ES can draw points, lines and triangles. Most applications use triangles as the main primitive. They can be submitted either as a list (3 vertices per triangle), a triangle strip (3 vertices + 1 vertex per subsequent triangle) or a triangle fan (3 vertices + 1 vertex per subsequent triangle).

Strips and fans involve fewer vertices and hence are less costly to submit and to process. Fans tend to be useful only for a fairly restricted set of problems

Strips are useful for drawing a contiguous set of triangles, but can also be used to draw an arbitrary set of triangles. To insert gaps into a triangle strip, you can replicate vertices to create a couple of zero-area triangles which will generate no pixels in the result. This only makes sense if the number of jumps is small.

In OpenGL ES 3.0 and above, it is possible to draw several objects with a single draw command when using glDrawElements(), glDrawElementsInstanced() or glDrawRangeElements(). A sentinel index value can be used to mark the end of one object. The fan or strip will then restart with the subsequent vertices. See the OpenGL ES 3.0 specification for more details on “PRIMITIVE_RESTART_FIXED_INDEX.”

4.3.4 Minimizing API Calls

There is an overhead, beyond the usual function-call cost, involved with every API call e.g.

- Error checking
- Binding the correct thread state
- Copying data

Therefore, it is valuable to minimize the number of calls to GL.

To reduce drawing calls, it is useful to render a few large objects instead of lots of smaller ones. OpenGL ES 3.0 and above provide a number of flexible drawing commands that help with minimizing draw calls:

- glDrawArraysInstanced()
- glDrawArraysIndirect()
- glDrawElementsInstanced()
- glDrawRangeElements()
- glDrawElementsIndirect()

Minimizing state changes is also important to maintaining fast graphics. Group objects according to the shaders and other GL state associated with them, and draw them together to avoid needless state changes.

Applications should set up immutable or infrequently changed state once at the beginning of time e.g.

- GL state such as clear color
- Load and submit static textures
- Load static geometry and create VBOs
- Create and compile shaders

The glGetError() function can be used in debug and development builds, but it should be removed from the release build. It might be useful to query the error status once per frame to detect e.g. out of memory conditions.

4.3.5 Avoiding Stalls

The GPU driver is able to run several frames ahead of what appears on the screen due to its pipelined design. Typically one buffer is being displayed; a second is ready for display whilst a third is being rendered to. This is called triple-buffering and it allows the display and rendering processes to be asynchronous of each other.

When an application calls eglSwapBuffers(), this instructs the driver that the rendering for the current frame is complete and that it should be displayed at the next opportunity e.g. during a vertical sync of the device. The swap-buffers API will only block if there are no free buffers to render into. The application can therefore continue to prepare frames in advance. This helps applications to keep their graphics animations smooth. It also helps keep the hardware busy.

However, there are some APIs that defeat this scheme: glReadPixels, glCopyTexImage, glCopyTexSubImage, glFinish, eglWaitClient and eglWaitGL. These calls force the graphics pipeline to be flushed, either explicitly as part of their function, or implicitly because they are reading the results of rendering. So, for example, the glReadPixels function needs to grab pixels from the currently bound color buffer. To obtain these pixels, it must wait for all pending rendering to complete. This could involve flushing several frames through the pipeline. The application is therefore stalled while waiting for the data to return.

There is a second cost involved. In order to be able to “continue where it left off”, all the buffers associated with the current framebuffer must be saved to memory and then restored. This is a large cost both in terms of storage and for bandwidth. If it is known that the content of buffers is not needed in future, some of this cost can be avoided by using the discard extension discussed in section “Discarding Buffers” below.

Switching between render targets has a similar overhead. Applications should try to minimize the number of times they swap from one render target (e.g. FBO, Pbuffer or main window) to another.

Flushes can also be caused if the driver’s internal resources are exhausted. This can happen, for example, if a texture is updated many times within a single frame. This might happen as the result of incremental updates to an “atlas” or “font” texture. Poor performance will result. In this circumstance it is better, if possible, to create the font/atlas up front, or to restrict the number of modifications to a few per frame.

4.3.6 Buffer Clear

On the VideoCore GPUs, a full buffer clear is essentially a zero-cost operation. To take advantage of this feature, avoid using partial scissored or masked clears. If possible, clear the color, depth and stencil buffers together as a single, full frame clear.

4.3.7 Framebuffer Depth

Always consider the bandwidth implications of your choice of framebuffer. If you don’t need an alpha channel on your graphics buffer, consider using RGB565 mode instead of RGBA8888, as a 16-bit framebuffer will use half the bandwidth of a 32-bit one.

Be aware, however, if you are rendering to a small resolution that is going to be stretched onto the display, the default mode of operation is to dither the RGB565 output. When the dither pattern is later stretched by the display subsystem, it becomes very noticeable. For small resolutions, you should use RGBA8888 rendering for this reason (or disable dithering in RGB565 mode).

4.3.8 Texture Mapping

Textures can be used in OpenGL ES for a variety of purposes e.g.

- Coloring objects

- Lighting
- Bump mapping
- Displacement mapping
- Complicated function map

Careful consideration should be given to the use of texture mapping. The choice of texture filter mode and formats can have a significant impact on overall performance.

Textures should be submitted and modified infrequently. Ideally, load textures once at the start of the application. When a glTexImage2D or glTexSubImage2D function is called, the new texture data must be copied by the driver. Moreover, if the previous contents of the texture is still live (because it is used in a previous, as yet unrendered, frame), a copy of the entire texture must be made (see “Avoiding Stalls”).

Select the smallest texture which delivers the quality needed in your application. There is a trade-off between quality and performance:

- RGBA8888 (32-bits per texel) is high quality but uses a lot of bandwidth
- RGB565 (16 bits per texel) may be an acceptable option
- ETC1 (4-bits per texel) gives good results on smooth natural images, but is RGB only – if alpha is required a separate texture may be needed
- ETC2/EAC is an improved compressed format that performs better on synthetic images and also allows for an alpha channel. This is only available from VideoCore V.

Compressed texture formats help with both memory bandwidth and footprint.

Unless you know *a priori* that a texture will appear on-screen at a certain size, it is important to use mip-maps. They improve the quality of the results and can significantly reduce the bandwidth cost of texturing. A mip-map is a stack of down-sampled copies of the original texture image which are selected so as to match the projected screen size of the texture. This process helps to make texture memory accesses coherent and as small as possible. This makes the texture caches more effective and reduces the bandwidth load.

When using mip-mapping, use linear_mipmap_nearest (bilinear) instead of linear_mipmap_linear (trilinear) where possible. Bilinear interpolation reads and combines four texels per pixel whereas trilinear requires eight.

If textures are frequently being modified, the cost of generating mip-maps may become prohibitive.

4.3.9 Discarding Buffers

When rendering to the default (displayed) framebuffer, the GL does not preserve the output buffers from one frame to the next. However, when drawing to an FBO, the situation is different. After rendering to an FBO, the GL does not know whether the contents of the associated depth and stencil buffers will be needed in future. The application might want to render to FBO A, then FBO B and finally return to where it left off in FBO A to complete its work (note: we would not normally recommend this behaviour).

Oftentimes, however, applications wish to render to an FBO and then forget everything except the final color output. So the depth and stencil buffers have done their work and can be discarded. The “discard framebuffer” extension in OpenGL

ES 2.0 allows applications to declare their intent with respect to these buffers. Without this extension, the driver has no way to knowing what the application will do in the future, so it is forced to save and reload buffers that are not needed.

In OpenGL ES 3.0 and above, this functionality has been incorporated in the core as the glInvalidateFramebuffer() command.

4.3.10 Conditional Code

The hardware units that run shader programs are single-instruction multiple-data (SIMD) machines. Conditional code on this style of architecture is often implemented using predicated instructions. This means that both arms of an if-then-else are executed – some GPU elements will be enabled during the "if" part and others during the "else" part.

For this reason, conditionals in shader programs should be avoided where possible. It can be better to use several variants of a shader program (perhaps generated programmatically or by use of the preprocessor) than to set a uniform and use 'if' statements in the program to choose different paths. For this reason "Uber-shaders" (which implement multiple functionalities) are especially bad.

Note that this behavior is highly dependent on the specific GPU architecture and on the shader compiler. Benchmarking is recommended.

4.3.11 Early Calculation

It is usually sensible to calculate constant values as early in the pipeline as possible. So, for example, you might calculate quantities in the vertex shader and pass the results as varyings to the fragment shader. There are usually far more fragments than vertices so the GPU compute cost is significantly reduced.

Similarly, a single calculation in the application passed as a uniform to multiple vertex shaders reduces the total number of evaluations.

However, a word of caution is in order. If your application is CPU limited, there may be some benefit to calculating values in the vertex shader even if the work is replicated for each vertex. The GPU load may be greater, but you can free up some CPU cycles and these may be more valuable to you.

4.3.12 Use OpenGL ES 2.0+

OpenGL ES 1.1 is a legacy API. Nearly all embedded GPUs in the market today implement OpenGL ES 2.0 or greater. When starting new projects, we recommend using the latest version of OpenGL ES available; it is more flexible and more efficient.

The VideoCore architecture is designed to implement OpenGL ES 2.0 and above. Therefore, programming in this API is "closer to the metal". OpenGL ES 1.1 is effectively translated by the driver software into an equivalent OpenGL ES 2.0 set of vertex and fragment shaders. Because this process is general, it is not able to take advantage of application specific knowledge to optimize these shaders.

As a concrete example, consider an application that is doing some 2D composition of images. In OpenGL ES 1.1, we would use the model-view matrix (a 4x4 matrix of floats) to position each draw. The driver does not know that the application is dealing only with 2D objects so has to produce a generic vertex shader to handle matrices. When writing this application in OpenGL ES 2.0, the programmer can craft a bespoke shader that takes a position and scale as inputs (4 floats) which is consequently smaller and performs less computation.

4.3.13 Other Considerations

The following are some other factors that could help with improving application performance.

- Ensure blending is disabled whenever it is not required. Blending operations can impact performance slightly, so do not use them unless you need to.

Chapter 5: Power Management

5.1 Driver Side Power Management

The driver can detect periods of activity and auto power down and up. To best utilize this mode, construct the application so it redraws in response to application events (key-presses/animation timers/etc.) rather than fixed time redraw based off vsync.

5.2 S3 Power Management

In S3 mode, the CPU is not running and the auto power cannot be achieved. For this, a slightly different approach is required.

The VideoCore GPU is a frame renderer, and as such cannot be put into an S3 state without first making sure that any currently pending renders are complete. This requires sending a signal of intent to the application to enter S3, which in turn would suspend any further issue of OpenGLES commands.

Power down is achieved by calling the respective Nexus/kernel functions.

```
NEXUS_Platform_SetStandbySettings();
system("echo mem > /sys/power/state");
```

A simple example of S3 power management is included in the reference software ‘cube_s3power’. In this, the respective power down code is contained at the display loop.

Chapter 6: Video Texturing

As of Reference Software release 17.4, the video texturing example has been substantially reworked. It no longer makes use of proprietary extensions, but does make use of “fences” for synchronization between the video and 3D frames.