OpenGL ES Quick Start

May 5, 2015

Broadcom Corporation • 5300 California Avenue • Irvine, CA, 92617 • Phone: 949-5300 • Fax: 949-926-5203

STB_XXX

January 16, 2013

Revision History

Doc Number	*Revision Date*	*Description/Author*
STB_XXX	04/08/11	Initial version.
	03/12/12	Revised.
	16/01/13	Revised, added performance, video & new platforms.
	07/03/13	Revised, added S3 power management support.
	04/04/13	Revised, added CPU access to GPU data
	05/04/15	Revised, build flags for NXCLIENT

Contents

Section 1: Introduction

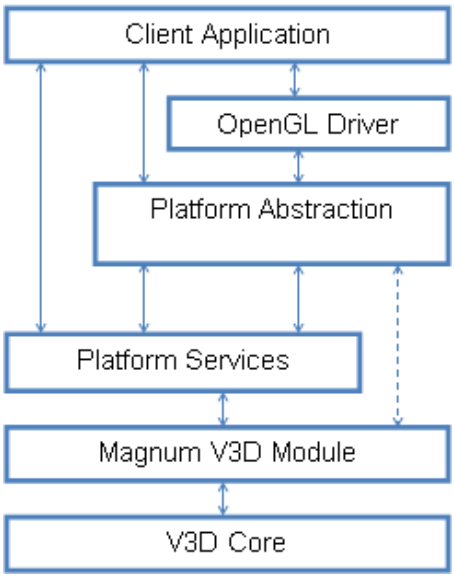
Overview

OpenGL ES is an API for 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.

Currently, there are two versions of the OpenGL ES API. The older legacy version, OpenGL ES 1.1, provides a fixed-function 3D pipeline. The newer version, OpenGL ES 2.0, is more flexible and allows developers to provide shader-programs for the vertex and fragment (pixel) stages of the pipeline. For best performance, flexibility and forwards compatibility, we recommend that developers adopt OpenGL ES 2.0.

OpenGL ES itself is device and platform independent. Thus, it does not provide 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. There are three main components: the OpenGL driver itself, platform layers and a Magnum V3D module. The platform layers provide a bridge between the platform-independent driver and the underlying platform services, in this case Nexus, but potentially any underlying system. The platform layer provided 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.



Further Reading

OpenGL ES 1.1 and 2.0 and the EGL are standardized APIs. The standards are maintained by the Khronos Group and they maintain 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 the OpenGL ES 1.1 and 2.0 and EGL standards.

Locations

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

- *rockford/applications/khronos/v3d/nexus* contains the example applications.
- *rockford/middleware/platform* contains the platform layer code. In particular:
    - *rockford/middleware/platform/nexus* holds the platform layer for Nexus.
- *rockford/middleware/v3d* contains the driver implementation. In particular:
    - *rockford/middleware/v3d/interface/khronos/include* holds the OpenGL ES and EGL headers files used by Open GL ES applications.
- *nexus/bin* contains the shared libraries and example executables once they have been built.

Section 2: Basic Usage

The Examples

The examples folder contains some basic applications, and some which demonstrate the use of more complex features:

- cube: draws a spinning cube.
- earth_es2: is a basic animation of the Earth, Moon and a star-field.
- eglimage: shows how to texture using an EGL image texture source.
- video_texturing: shows how to texture using a Nexus video source.
- poly_rate and poly_rate_es1: are simple benchmarks for measuring polygon rates in ES 1.1 and 2.0.
- benchmark: complex benchmark tests described in the V3D_AppNote_Benchmark application note.

Build and Run Cube Example

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. For this example, we will be using single-process, user-mode, so we ensure that the environment variable NEXUS_MODE is unset. Multi-process modes are discussed later in the document.

The cube example creates and draws a simple cube and rotates it. To build a release version of the cube example follow these steps:

```
cd rockford/applications/khronos/v3d/nexus/cube
unset NEXUS_MODE
make
```

This will build the three shared objects:

- Nexus library: nexus.so
- OpenGL ES driver: libv3ddriver.so
- Platform layer: libnxpl.so

They will be automatically copied into *nexus/bin*. To run the application use the “nexus” script in *nexus/bin* thus:

```
cd nexus/bin
nexus cube
```

The Nexus script installs the appropriate Nexus kernel modules and then invokes the application.

The cube example can 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
```

Creating Your Own Applications

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.

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 2.0 applications need to include the headers for OpenGL ES and for EGL via:

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

Applications which use extensions to the OpenGL ES API will also need to include:

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

OpenGL ES 1.1 applications should include `<GLES/gl.h>` and `<GLES/glext.h>` respectively instead.

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:

- *rockford/middleware/v3d/interface/khronos/include:* for the OpenGL ES and EGL headers
- *rockford/middleware/platform/nexus:* for the nexus platform layer.

Libraries

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

- pthread system threading library
- v3ddriver the OpenGL driver
- nxpl the Nexus platform layer

Ensure that the following folders are in the compiler's link library path:

- *rockford/middleware/v3d/lib_$(PLATFORM)_debug/* or *rockford/middleware/v3d/lib_$(PLATFORM)_release/*
- *rockford/middleware/platform/nexus/lib_$(PLATFORM)_debug* or *rockford/middleware/platform/nexus/lib_$(PLATFORM)_release*

where $(PLATFORM) will be the name of the target board e.g. 97425.

Lifecycle of an Application

A typical single-process, exclusive-display-mode, Nexus OpenGL ES application follows the steps summarized below. See the multi-process section later in this document for details of running multiple applications simultaneously.

- 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
- For each frame:
    - Clear buffers.
    - For all geometry
        - Set up GL State.
        - Submit geometry.
        - Reset GL state.
    - Swap buffers.
- Finalize GL.
- Release the OpenGL ES context and finalize EGL.
- Destroy the native window.
- Unregister the Nexus display with 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 (non-exclusive display options are described later in the document). 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;          /* X position of window – unused in exclusive
mode */
win_info.y        = 0;          /* Y position of window – unused in exclusive
mode */
```

```
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 a surface using `eglCreateWindowSurface()` passing the native window created above. Each native window can have at most one surface. 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 some 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 maximum CPU usage and frame tearing and is highly undesirable for production code.

Section 3: Advanced Usage

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

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. You could also write a custom platform layer that does composition of the 3D rendered buffers.

CPU access to GPU data

To access what you have just rendered, the temptation for a developer to call glReadPixels() can be great. 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. In most vendors case it will imply a conversion from an internal GPU format to a CPU raster format.

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. A pixmap removes all the disadvantages of reading via the GL API.

For an example of pixmap rendering see

rockford/applications/khronos/v3d/nexus/cube_pixmap

Multi-process

In single-process, exclusive display mode, only one OpenGL ES application can be running at any time. That application has complete control of the display. Many real-world use-cases require that multiple applications can run simultaneously, each using a portion of a composited display.

Nexus surface compositor

Nexus has an in-built surface compositor which can be used to virtualize the display in this way. The low-level V3D Magnum module will virtualize the 3D hardware so that multiple OpenGL ES applications will run simultaneously, with the available compute and resources being fairly shared by the applications.

A separate server application is responsible for marshaling the results from the graphics processes and composing them for display. The server process is akin to a window manager for a desktop environment. You can create your own server application, or use / modify the provided NxClient framework which provides an 'nxserver' application.

NxClient

NxClient is an add-on to core Nexus that provides a client-server framework suitable for implementing a display compositor for multiple applications. Video, audio, 2D and 3D applications can each create windows on the display and will all run simultaneously. Refer to the Nexus documentation for more information about NxClient.

The V3D examples can be compiled to target multi-process use with an NxClient based server.

You should first build the NxClient examples:

```
cd vobs/nexus/nxclient
make
```

Then build the 3D examples – we'll use cube as an example:

```
cd rockford/applications/khronos/v3d/nexus/cube
export NXCLIENT_SUPPORT=y
make
```

Open a telnet session and run the server application:

```
cd nexus/bin
nexus nxserver
```

Open a second telnet session to run the client applications:

```
cd nexus/bin
./nexus.client cube &
```

The example applications take command line arguments to control their size and position on the display:

```
./nexus.client cube d=360x240 o=100x100 &
```

will make a cube window 360x240 pixels at 100,100.

Kernel / User Mode

Nexus based applications can be compiled and run in either Nexus kernel-mode or user-mode. In kernel-mode, most of the Nexus code is run in kernel-space, whereas in user-mode there is only a small component in Nexus which is in kernel-space. For more details of how Nexus kernel and user modes differ, consult the Nexus documentation.

Regardless of what mode Nexus is running in, the platform layer, OpenGL ES driver and user applications are all run in user-space.

User Mode

To build the client applications for NxClient user-mode:

```
unset NEXUS_MODE
unset NXCLIENT_SUPPORT
make clean

cd rockford/applications/khronos/v3d/nexus/cube
make
cd rockford/applications/khronos/v3d/nexus/earth_es2
make
```

Kernel Mode

To build the client applications for NxClient kernel-mode:

```
export NEXUS_MODE=proxy
unset NXCLIENT_SUPPORT
make clean

cd rockford/applications/khronos/v3d/nexus/cube
make
cd rockford/applications/khronos/v3d/nexus/earth_es2
make
```

Section 4: Runtime Flags

Export symbols

In the Linux environment, these can be added to either debug your app or customize the runtime behavior. Example usage:

```
export V3D_GL_ERROR_ASSIST=1
```

V3D_GL_ERROR_ASSIST

The OpenGL ES driver will print errors to the console when a GL or EGL error has occurred. You can avoid using glGetError() in many places by setting this. Note: The 'GPU Monitor' tool can also provide this information – see separate documentation.

V3D_DOUBLE_BUFFER

The 3d core normally runs triple buffered, this forces it to use double buffering. Double-buffering uses less memory and has slightly lower frame latency, but will not smooth out variations in frame-rate as well as triple-buffering.

V3D_FORCE_DITHER_OFF

16-bit 565 output modes enable dithering. This forces it off. This will make color banding more visible, but may produce better results if the final image is scaled up onto the display as the dither pattern will not be present.

V3D_RELAX_HEAP_ERRORS

OpenGL ES driver will check the heap that it is assigned. If the heap is invalid for some reason the driver will normally abort. Setting this will relax some of those requirements so that the driver can be tested. This should not be permanently set.

V3D_BIN_MEM_MEGS

The amount of memory that the Magnum V3D module will reserve for the hardware binner to use. Note: this is the total for all simultaneous applications, not per-application. Defaults to 16MB if not set.

V3D_BIN_MEM_CHUNK_POW

Determines the size of each bin memory chunk, and by implication the total number of chunks. A value of 0 will make 256KB chunks, 1=512KB, 2=1024KB, 4=2048KB etc.

Section 5: Performance

Introduction

This section describes some general guidelines for maximizing performance from Broadcom's VC4 GPU based 3D graphics solutions when using OpenGL ES. We assume some working knowledge of 3D graphics and the OpenGL ES family of APIs.

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.

Programming Guidelines

The following hints and guidelines should help you to maximize the performance of your OpenGL ES application on the VC4 GPU. Not all of them are orthogonal, and to maximize performance, it may be necessary to make some trade-offs.

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.

Vertex Buffer Objects

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 whereas vertex buffer objects are copied once at submission time and retained in device memory for re-use in subsequent draw commands.

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 VC4 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
	…

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 less 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 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.

Minimizing API Calls

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

- 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. The triangle strip trick with zero area triangles described above can help.

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 switching.

Applications should set up immutable or infrequently changed state once at the beginning 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 can be useful to query the error status once per frame to detect e.g. out of memory conditions. Also take a look at the V3D_GL_ERROR_ASSIST environment variable described in the quick start guide.

Avoiding Stalls

The VC4 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 this 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 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.

Buffer Clear

On the VC4, 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.

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 565 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 565 mode).

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

Compressed texture formats help with both memory bandwidth and footprint.

Make the most of your texture channels. Analysing your needs may suggest a sensible choice of texture format:

- No alpha – use an ETC1 or RGB565 format
- Two channels – use Luminance-Alpha
- One channel – use Luminance or Alpha
- Two two-channel textures could be held in a single RGBA texture

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 downsampled 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. An example might be in applying a video as a texture.

Discarding Buffers

When rendering to the default (displayed) framebuffer, the GL never preserves the depth and stencil 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 behavior).

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 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.

Judicious use of this extension can save considerable memory bandwidth.

Conditional Code

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

For this reason, uniform conditionals in shader programs should be avoided. It is usually better to use several variants of a shader program 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.

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.

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. When starting new projects, we recommend using OpenGL ES 2.0. It is more flexible and more efficient.

The VC4 architecture is designed to implement OpenGL ES 2.0. 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.

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.

Section 6: Power Management

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 (keypresses/animation timers/etc.) rather than fixed time redraw based off vsync.

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 v3d is a frame renderer, and as such cannot be put into an S3 state without 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.

Section 7: Video Texturing

Introduction

This section describes the current recommendations for video texturing using VC4. These guidelines are valid only for drivers supporting the EGL_BRCM_image_update_control extension (those released during or after May 2012). Earlier drivers have very different performance characteristics.

Video texturing can be implemented with standard OpenGL ES texturing using glTexImage2D, but the CPU load will be heavy and the performance will be poor for large images. We recommend using the EGL Image based solution presented here.

EGLImage & glEGLImageTargetTexture2DOES

These specifications describe the standard EGLImage APIs we use to support video texturing:

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image.txt
http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_base.txt
http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_pixmap.txt
http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_gl_image.txt

The VC4 driver allows the creation of an EGLImage from a native pixmap in RGBX8888 format. Note: YUV444 can be thought of a being in RGBX8888 format – with an appropriate shader to convert back to RGB.

The native pixmap is made in the same way as it would be when using it as a render target. Native pixmaps are usually created by calling a function in the 'platform layer' which routes via the driver to determine the size and alignment constraints for the allocated surface.

The EGLImage is created from the pixmap like this:

```
m_eglImage = eglCreateImageKHR(display, EGL_NO_CONTEXT, EGL_NATIVE_PIXMAP_KHR,

                               (EGLClientBuffer)eglPixmap, attr_list);
```

To use the EGLImage as a texture, we have to create a texture id as usual with `glGenTextures` and bind it with glBindTexture. We then tell the driver that the texture data comes from an EGLImage using:

```
glEGLImageTargetTexture2DOES(GL_TEXTURE_2D, m_eglImage);
```

Whenever the texture is now bound, the VC4 will behave as if it is texturing directly from the memory wrapped by the EGLImage (i.e. the underlying pixmap). Under-the-hood, VC4 actually performs a hardware conversion of the EGLImage to an efficient internal format prior to rendering each frame. A shadow copy of the texture is essentially maintained internally in this process. Note: the EGL_BRCM_image_update_control extension (described later) can be used to optimize the conversion process by specifying when to update, and what needs updating.

Note: unlike earlier driver versions, RGBX8888 EGL images have no power-of-two width restriction. Behaviour of YUV422 EGL images is unchanged, and these therefore do still have power-of-two width limitations, and lower performance.

The EGL_BRCM_image_update_control extension

Performance

If you are using RGBX8888 EGLimage based texturing, you can use this extension to optimize the conversion from raster data to internal texture format.

Without the extension, the conversion process takes place for each EGL image prior to each frame being rendered. This allows the renderer to always see the latest copy of the data, and to use the efficient internal texture format for rendering.

Video data, and potentially other texture sources, are unlikely to change every time a new 3D frame is rendered. For 24fps video, and 60fps rendering, less than half the rendered frames actually have new video texture data.

Using the EGL_BRCM_image_update_control extension, you can inform the 3D driver when the video texture data has changed, and therefore how often to update its shadow copy. You can also provide update region information so that smaller updates can be performed. Both of these will result in performance improvements.

Here's an example of how you might use the extension in a rendering loop:

```
// Generate a texture ID
glGenTextures(1, &texID);

// Bind the texture
glBindTexture(GL_TEXTURE_2D, texID);

// Create our EGLImage
eglImage = eglCreateImageKHR(display, EGL_NO_CONTEXT, EGL_NATIVE_PIXMAP_KHR,
                             (EGLClientBuffer)pixmap, attrList);

// Register this EGL image for explicit updates.
// It won't automatically update now.
eglImageUpdateParameteriBRCM(display, eglImage,
                             EGL_IMAGE_UPDATE_CONTROL_SET_MODE_BRCM,
                             EGL_IMAGE_UPDATE_CONTROL_EXPLICIT_BRCM);

// Use the EGL image as the current texture
glEGLImageTargetTexture2DOES(GL_TEXTURE_2D, eglImage);

while (1)
{
   // We may not have a new video frame ready to use yet.
   if (HaveNewVideoFrame())
   {
      // Get the latest video frame
      stripedSurface = GetLatestVideoFrame();

      // Now destripe the surface into our destination buffer
      DestripeIntoPixmap(pixmap, stripedSurface);

      // We must wait for the destripe to complete now
      WaitForDestripeToComplete();

      // Tell the 3D core what region of the image has changed (all of it in this
      // case). Without this the texture will never change as we are in explicit
      // update mode.
      EGLint rect[4] = { 0, 0, width, height };
      eglImageUpdateParameterivBRCM(display, eglImage,
                                    EGL_IMAGE_UPDATE_CONTROL_CHANGED_REGION_BRCM,
                                    rect);
   }
   else
   {
      // Texture is unchanged from last frame
   }

   Render3DScene();
}
```

Tearing and update artifacts

3D rendering cores will often have a pipeline of multiple frames being processed simultaneously for performance reasons. V3D is no exception – up to three frames may be in-flight at any time.

This means that you can never know exactly when V3D will be texturing from your EGL image. If you only have one EGL image buffer you will very likely be updating it with a new video frame at the same time as V3D is reading it. This will result in 'tearing' being visible in the video texture.

The traditional way to fix this kind of tearing is to have three separate EGL images into which you destripe the video frames in turn. Providing you are using eglSwapBuffers() and swap interval is not zero, this guarantees that you cannot be writing and reading simultaneously. The drawback with this approach is that three EGL images (and their shadow copies) can amount to a large amount of memory.

If you cannot afford the memory required, the EGL_BRCM_image_update_control provides a lock/unlock mechanism to avoid tearing without requiring three EGL images.

Here's how you might modify the rendering loop above to include locking and unlocking:

```
while (1)
{
   // We may not have a new video frame ready to use yet.
   if (HaveNewVideoFrame())
   {
      // Lock the EGL image for writing. This might take some time if the 3D
      // core is using it right now.
      eglImageUpdateParameteriBRCM(display, eglImage,
                                   EGL_IMAGE_UPDATE_CONTROL_SET_LOCK_STATE_BRCM,
                                   EGL_IMAGE_UPDATE_CONTROL_LOCK_BRCM);

      // We must get the latest video frame now, not before the lock. The lock
      // might stall, so if we got the frame prior to locking it may be out of
      // date or worse, part of a circular buffer chain and being overwritten
      // by the video decoder.
      stripedSurface = GetLatestVideoFrame();

      // Now destripe the surface into our destination buffer
      DestripeIntoPixmap(pixmap, stripedSurface);

      // We must wait for the destripe to complete now
      WaitForDestripeToComplete();

      // Tell the 3D core what region of the image has changed (all of it in this
      // case). Without this the texture will never change as we are in explicit
      // update mode.
      EGLint rect[4] = { 0, 0, width, height };
      eglImageUpdateParameterivBRCM(display, eglImage,
                                    EGL_IMAGE_UPDATE_CONTROL_CHANGED_REGION_BRCM,
                                    rect);

      // Unlock the eglImage, so the 3D core can use it.
```

```
      eglImageUpdateParameteriBRCM(display, eglImage,
                                   EGL_IMAGE_UPDATE_CONTROL_SET_LOCK_STATE_BRCM,
                                   EGL_IMAGE_UPDATE_CONTROL_UNLOCK_BRCM);
   }
   else
   {
      // Texture is unchanged from last frame
   }

   Render3DScene();
}
```

Using this lock/unlock mechanism, your application simply locks the EGL image prior to updating it and unlocks it when done. V3D will take appropriate actions to avoid reading the buffer while you have it locked. The lock call may stall until V3D finishes using the buffer.

You can use a single buffer with locking and unlocking, but for very large images the visual results can sometimes be a little 'choppy' unless you use at least two EGL images in turn. With just one EGL image the application and V3D are both contending for use of a single resource. Both components can stall while waiting on the other. If you use two buffers, there are two resources shared by the two components. Contention will be lower, and there will be fewer stalls.

Be aware that if you use multiple buffers along with small update regions, you need to ensure that all the buffers get all of the changes and all have their regions updated appropriately. You cannot update a small region of image 1, then a different region of image 2 and expect GL to see both changes in both textures.

Performance

The following table list various performance metrics gathered on three different platforms.

The recommended, accelerated configurations are shown in **bold**.

					7435 - 3 slice, 4 QPU			7425 - 2 slice, 4 QPU			7231 - 1 slice, 4 QPU		
Display Resolution	**Texture Mode**	**Format**	**Source Video**	**Destripe Size**	**FPS**	**CPU %**	**Bandwidth MB/frame**	**FPS**	**CPU %**	**Bandwidth MB/frame**	**FPS**	**CPU %**	**Bandwidth MB/frame**
1280x720	EGLImage	YUV422	1920x872	512x480	189.3	62.0	6.5	133.7	40.2	6.9	95.0	63.6	6.7
1280x720	TexImage2D	YUV422	1920x872	512x480	181.8	71.7	4.7	149.6	53.6	4.7	74.7	71.7	4.6
1280x720	**EGLImage**	**RGBX8888**	**1920x872**	**512x480**	**208.6**	**73.7**	**5.0**	**171.3**	**53.0**	**5.1**	**107.0**	**78.0**	**5.0**
1280x720	TexImage2D	RGBX8888	1920x872	512x480	172.2	73.4	4.7	151.4	54.4	4.8	76.5	74.2	4.7
1280x720	**EGLImage**	**YUVX444**	**1920x872**	**512x480**	**208.0**	**73.5**	**4.9**	**170.8**	**52.7**	**5.0**	**104.5**	**76.9**	**5.0**
1280x720	TexImage2D	YUVX444	1920x872	512x480	171.4	73.0	4.7	151.0	53.3	4.7	74.1	71.4	4.6
1280x720	EGLImage	YUV422	1920x872	1024x720	100.9	34.9	16.8	71.3	22.8	19.1	69.1	47.6	10.1
1280x720	TexImage2D	YUV422	1920x872	1024x720	120.5	66.7	6.7	92.2	50.9	7.0	44.5	71.6	7.0
1280x720	**EGLImage**	**RGBX8888**	**1920x872**	**1024x720**	**160.8**	**56.8**	**7.5**	**115.6**	**38.2**	**8.1**	**79.2**	**61.0**	**8.3**
1280x720	TexImage2D	RGBX8888	1920x872	1024x720	96.8	73.1	6.7	95.6	47.9	7.1	42.1	73.2	7.0
1280x720	**EGLImage**	**YUVX444**	**1920x872**	**1024x720**	**160.3**	**58.8**	**7.4**	**115.8**	**37.7**	**8.0**	**79.0**	**61.0**	**8.3**
1280x720	TexImage2D	YUVX444	1920x872	1024x720	97.9	73.3	6.7	95.9	48.2	7.1	42.0	72.5	7.0
1920x1080	EGLImage	YUV422	1920x872	512x480	124.5	52.3	6.5	85.7	31.8	9.6	52.4	44.5	9.4
1920x1080	TexImage2D	YUV422	1920x872	512x480	100.3	55.2	6.1	86.5	41.9	6.2	43.7	52.1	6.0

1920x1080	**EGLImage**	**RGBX8888**	**1920x872**	**512x480**	**122.5**	**56.2**	**6.8**	**101.8**	**40.0**	**6.9**	**57.5**	**54.7**	**6.9**
1920x1080	TexImage2D	RGBX8888	1920x872	512x480	96.6	57.3	6.2	88.5	41.1	6.2	43.8	53.6	6.1
1920x1080	**EGLImage**	**YUVX444**	**1920x872**	**512x480**	**120.8**	**54.6**	**6.6**	**100.8**	**39.9**	**6.8**	**55.5**	**53.3**	**6.8**
1920x1080	TexImage2D	YUVX444	1920x872	512x480	64.7	63.5	8.2	87.3	40.5	6.1	43.3	52.2	6.0
1920x1080	EGLImage	YUV422	1920x872	1024x720	69.0	31.3	21.1	56.4	22.7	18.5	45.3	39.4	12.8
1920x1080	TexImage2D	YUV422	1920x872	1024x720	74.1	59.4	8.2	63.1	42.7	8.3	33.3	58.9	8.2
1920x1080	**EGLImage**	**RGBX8888**	**1920x872**	**1024x720**	**99.9**	**48.0**	**9.6**	**76.7**	**31.7**	**10.0**	**49.3**	**48.5**	**10.1**
1920x1080	TexImage2D	RGBX8888	1920x872	1024x720	66.6	64.3	8.2	64.5	41.3	8.4	33.7	60.4	8.3
1920x1080	**EGLImage**	**YUVX444**	**1920x872**	**1024x720**	**99.3**	**47.2**	**9.5**	**76.5**	**31.8**	**9.9**	**48.6**	**48.0**	**10.1**
1920x1080	TexImage2D	YUVX444	1920x872	1024x720	64.7	63.5	8.2	64.5	41.4	8.3	32.9	59.9	8.3
1920x1080	EGLImage	YUV422	1920x872	2048x2048	13.7	10.0	146.9	12.4	7.5	163.8	13.4	15.3	120.2
1920x1080	TexImage2D	YUV422	1920x872	2048x2048	26.5	66.5	24.6	24.1	52.6	25.1	15.4	64.0	24.7
1920x1080	**EGLImage**	**RGBX8888**	**1920x872**	**2048x2048**	**39.0**	**22.7**	**34.0**	**33.4**	**16.2**	**33.3**	**26.4**	**27.9**	**31.3**
1920x1080	TexImage2D	RGBX8888	1920x872	2048x2048	14.1	80.8	24.3	23.2	54.2	25.1	12.8	70.6	24.8
1920x1080	**EGLImage**	**YUVX444**	**1920x872**	**2048x2048**	**40.4**	**23.5**	**33.1**	**33.5**	**16.6**	**33.2**	**26.6**	**28.3**	**31.1**
1920x1080	TexImage2D	YUVX444	1920x872	2048x2048	13.8	80.1	24.2	23.4	53.6	25.1	12.1	72.1	24.7

Notes:

1) CPU is average CPU usage as a percentage during rendering

2) Bandwidth figures only include read bandwidth from memory into V3D. They do not include writes into memory from V3D, or any CPU or video decode bandwidths.

Important Performance Notes

Use RGBX8888 EGL images, not YUV422

YUV422 images cannot be accelerated in the same way as we have for RGBX8888. These will not use the new video texturing mechanism, and are therefore no longer recommended.

Size matching

In order to obtain peak rendering performance and bandwidth utilization, it is important to match the video decode size as closely as possible with its displayed size.

This is important for three reasons:

1) The video textures are not mip-mapped, so displaying a large video texture in a small screen area will result in a lot of sparse texture lookups. Sparse lookups use more bandwidth than spatially local ones.
2) Submitting textures larger than their display size wastes resource during conversion to internal texture format.
3) Sparsely sampled textures will show aliasing artifacts, reducing apparent quality.

In general, it is preferable to use a video texture that is smaller than its displayed size than one which is larger.

Submit at video frame rate

Use the EGL_BRCM_image_update_control extension to explicitly mark changed video frames. There's no point updating the texture 60 times per second, when the video is only decoding at 24 frames-per-second.

Memory Footprint

The following table show the memory footprint requirements of an RGBX8888 EGL image chain for video texturing. The old driver column relates to previous versions of the driver, where tearing could only be eliminated by using three EGL images, and images had power of two width restrictions. Using the EGL_BRCM_image_update_control extension, you can use 1, 2 or 3 EGL images in the new driver. Using more buffers results in less contention and better visual appearance.

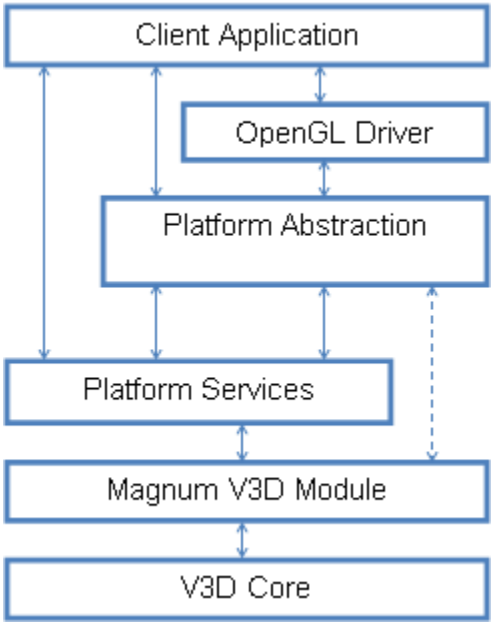
RGBX8888		Memory Footprint (in Mbytes) (All without tearing)			
Width	Height	Old driver (triple buffered)	Single buffered	Double buffered	Triple buffered
512	512	3.00	2.00	4.00	6.00
1024	768	9.00	6.00	12.00	18.00
1280	720	16.88	7.03	14.06	21.09
1024	1024	12.00	8.00	16.00	24.00
1920	1080	25.31	15.82	31.64	47.46
2048	2048	48.00	32.00	64.00	96.00

Section 8: New Platforms

Introduction

Broadcom's V3D graphics core is available in many different SoC solutions for several different markets. As such, it will need to interface to several different underlying platforms. Platform in this context refers to the underlying software layer responsible for managing resources such as memory and renderable surfaces. To avoid having to support multiple platforms in the 3D driver code directly, we have implemented an abstract platform support layer. To support a new platform, a new platform layer implementation is needed, but driver changes are not required.

Several platform layer implementations are provided with the driver stack and can be used as is. You may also want to customize or write one of your own to target a new underlying system.



- Client application is the application code written by the developer. It has access to the OpenGL driver via the standard API. It also must initialize the platform layer to use either the standard layer or a bespoke platform layer. This code will be standard for most applications.
- The OpenGL driver is supplied as a shared object file and is used via the standard EGL, OpenGL ES and OpenVG APIs. Developers can treat this as a black box.
- The platform abstraction layer provides the services required by the driver. These fall into three specific areas of functionality:

    1. Device memory allocation
    2. Communications with the hardware control module
    3. Display buffer management

- When implementing a platform layer, these services will be mapped onto the underlying platform services e.g. Nexus, DirectFB, CDI etc.
- The V3D Module is a standard Magnum component which abstracts the V3D hardware. The platform services or platform abstraction layer can communicate with the V3D core via this layer. It must never access the hardware directly.

The platform layer is linked as part of the application, not the driver, so it is possible to replace it when specialized behavior is required (e.g. dual display support for example). Broadcom supplies default implementations (in vobs\rockford\middleware\platform) for various platforms (e.g. Nexus exclusive display (single process), Nexus surface compositor (multi-process), Android application, Android compositor and DirectFB) which can be used as a starting point.

Registering the platform

In order to use the V3D EGL/GLES driver, the application must register its platform interface with the driver. This is done via an extra API exposed from the driver library.

```
typedef struct
{
   BEGL_MemoryInterface  *memInterface;
   BEGL_HWInterface      *hwInterface;
   BEGL_DisplayInterface *displayInterface;

   BEGL_DisplayCallbacks displayCallbacks;/* For client to callback into driver */

} BEGL_DriverInterfaces;

void BEGL_RegisterDriverInterfaces(BEGL_DriverInterfaces *iface);
```

The driver interface structure points at the three specific interfaces that are being hooked out and provides some callback pointers for the display and hardware layers. This is normally done via an API provided in the platform layer itself. For example, the default Nexus platform implements a function `NXPL_RegisterNexusDisplayPlatform()`.

Each interface has an associated “context” pointer. This pointer is opaque to the 3D driver code. The platform layer interfaces can use this pointer to store its own state information. The context is passed to all the platform layer functions.

Device Memory Allocation

The V3D driver needs to allocate and manage blocks of device memory for everything that is accessed by the V3D core. This includes render buffers, textures and control lists for example. The driver uses this interface to allocate all device memory that it accesses directly. Bin memory blocks are allocated using the hardware interface described in the next section.

```
typedef struct
{
   void *context;

   /* Allocate aligned device memory, and return the cached address */
   void *(*Alloc)(void *context, size_t numBytes, uint32_t alignment);

   /* Free a previously allocated block of device memory. Pass a cached address.*/
   void (*Free)(void *context, void *pCached);

   /* Return a physical device memory offset given a cached pointer. */
   uint32_t (*ConvertCachedToPhysical)(void *context, void *pCached);

   /* Return a cached memory pointer given a physical device memory offset. */
   void *(*ConvertPhysicalToCached)(void *context, uint32_t offset);

   /* Flush the cache for the given address range.*/
   void (*FlushCache)(void *context, void *pCached, size_t numBytes);

   /* Retrieve information about the memory system
      Type can be one of :
       BEGL_MemHeapStartPhys, BEGL_MemHeapEndPhys, BEGL_MemCacheLineSize,
       BEGL_MemLargestBlock, BEGL_MemFree
   */
   uint32_t (*GetInfo)(void *context, BEGL_MemInfoType type);

   /* Copy data as fast as possible. Might use DMA where available (if both
      pointers are in contiguous device memory) */
   void (*MemCopy)(void *context, void *pCachedDest, const void *pCachedSrc,
                   uint32_t bytes);

} BEGL_MemoryInterface;
```

The platform layer creator (or application developer) is free to use whatever mechanism is appropriate on their platform to implement these functions. Nexus platforms, for example, use the Nexus heap memory manager to fulfill these requests.

Hardware Interfacing

The hardware interface is a high-level abstraction of the V3D involving the submission of discrete “jobs” or “tasks”. The Magnum V3D module implements all of the required functionality for these interfaces. The platform layer simply has to route these calls to Magnum via any intermediate layers (Nexus for example).

```
typedef struct
{
   void *context;

   /* Fills in information about the version and features of the V3D core */
   bool (*GetInfo)(void *context, BEGL_HWInfo *info);

   /* Send a job to the V3D job queue */
   bool (*SendJob)(void *context, BEGL_HWJob *job);

   /* Retrieves and removes a notification for this client */
   bool (*GetNotification)(void *context, BEGL_HWNotification *notification);

   /* Acknowledge the last synchronous notification */
   void (*SendSync)(void *context, bool abandon);

   /* Request bin memory */
   bool (*GetBinMemory)(void *context, const BEGL_HWBinMemorySettings *settings,
                                       BEGL_HWBinMemory *memory);

   /* Setup or change performance monitoring */
   void (*SetPerformanceMonitor)(void *context,
                                 const BEGL_HWPerfMonitorSettings *settings);

   /* Get performance data */
   void (*GetPerformanceData)(void *context, BEGL_HWPerfMonitorData *data);

} BEGL_HWInterface;
```

The driver allocates bin memory using GetBinMemory(), it then uses this in a “job” which is submitted via SendJob(). Each job can contain a number of callbacks and the data associated therewith is accessed using GetNotification().

Tile format (TF) conversion jobs require synchronization with the driver. This is handled using the SendSync() function which is called when the conversion task is allowed to continue.

The remaining methods support querying information from the core. GetInfo() is for interrogating the version and capabilities of the core and Get/SetPerformanceData() functions are used to monitor the behavior of the core as it runs (for example by the GPUMonitor tool).

Most of these methods are simply implemented by converting the BEGL datatypes into those appropriate for the back-end of the driver and attaching a client ID. For example, one might convert BEGL types to Nexus types and the Nexus module would convert these to Magnum types.

Display Buffer Management

This is the most complex of the platform interfaces. It manages the creation and destruction of render buffers, along with the presentation of rendered buffers when they become available.

The V3D hardware has certain constraints on its render buffers in terms of memory alignment and padding. Because of this, the driver must either be responsible for creating them, or the driver must be queried for the particular constraints the platform layer or application must respect when making them. The driver doesn't care about how the buffer memory is wrapped – so you can create a native surface type when asked for a buffer, provided that the alignment and padding constraints are met for the underlying memory.

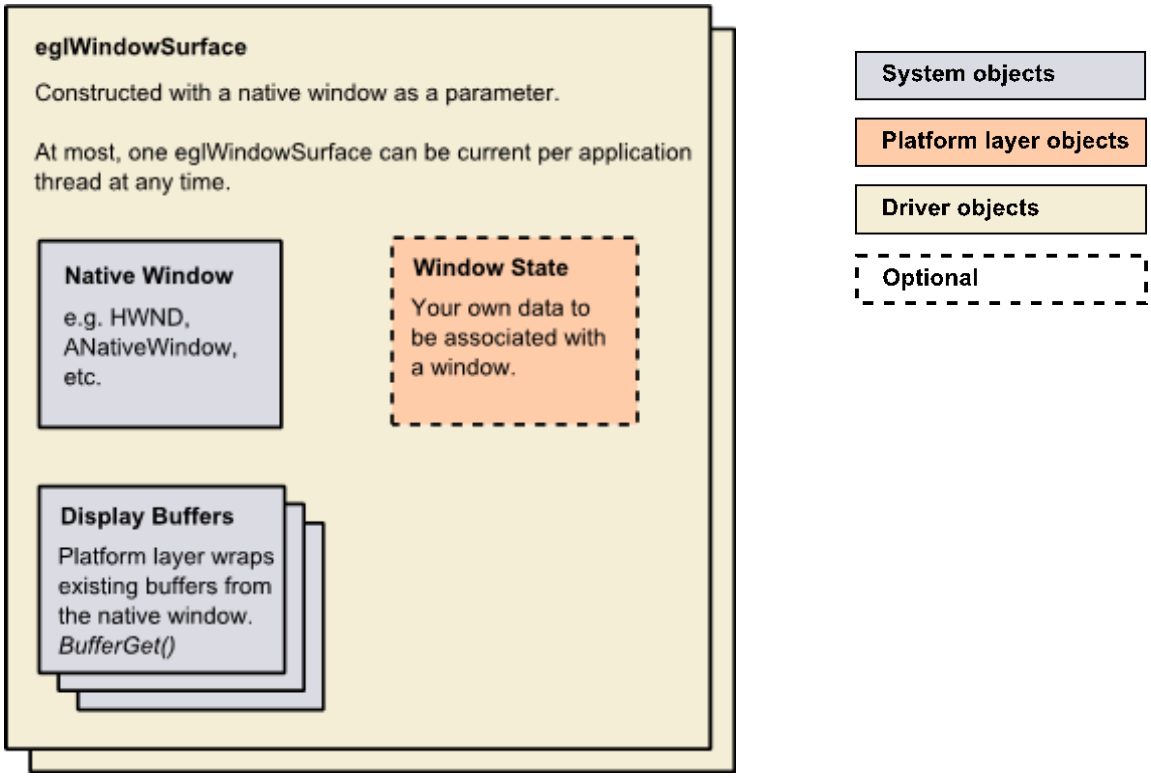
The display interface is as follows. The functions will be described in detail below, so are not commented here for brevity.

```
typedef struct
{
   void *context;

   BEGL_Error (*BufferDisplay)(void *context,
                               BEGL_BufferDisplayState *state);

   BEGL_Error (*WindowUndisplay)(void *context,
                                 BEGL_WindowState *windowState);

   BEGL_BufferHandle (*BufferCreate)(void *context,
                                     BEGL_BufferSettings *settings);

   BEGL_BufferHandle (*BufferGet)(void *context,
                                  BEGL_BufferSettings *settings);

   void * (*WindowPlatformStateCreate)(void *context,
                                       BEGL_WindowHandle window);

   BEGL_Error (*WindowPlatformStateDestroy)(void *platformState);

   BEGL_Error (*BufferDestroy)(void *context,
                               BEGL_BufferDisplayState *bufferState);

   BEGL_Error (*BufferGetCreateSettings)(void *context,
                                         BEGL_BufferHandle buffer,
                                         BEGL_BufferSettings *outSettings);

   BEGL_Error (*WindowGetInfo)(void *context,
                               BEGL_WindowHandle window,
                               BEGL_WindowInfoFlags flags,
                               BEGL_WindowInfo *outInfo);

   BEGL_Error (*BufferAccess)(void *context,
                              BEGL_BufferAccessState *bufferAccess);
```

```
   BEGL_Error (*IsSupported)(void *context,
                             BEGL_BufferFormat bufferFormat);

   BEGL_Error (*GetNativeFormat)(void *context,
                                 BEGL_BufferFormat bufferFormat,
                                 unsigned int *outNativeFormat);

   BEGL_Error (*SetDisplayID)(void *context,
                              unsigned int displayID,
                              unsigned int *outEglDisplayID);

} BEGL_DisplayInterface;
```

Swap-chain overview

The EGL driver needs to obtain a number of display buffers to use as a swap-chain for its rendering. This process is triggered when eglCreateWindowSurface is called by the application.



Each EGL window surface is connected to a single native window. Attempting to connect multiple EGL window surfaces to the same native window will result in an error.

Rarely, applications will render into multiple native windows. This typically implies that a surface compositing platform is in play. Your platform layer should account for this if you want to support multi-windowed applications. You may have multiple eglWindowSurfaces, each with its own set of swap-chain buffers. Multiple windows can also be written simultaneously from multiple threads, so the platform layer must be thread-safe. An application might also do multi-threaded rendering, where all but one thread is rendering into FBOs or Pbuffers.

Your platform layer is responsible for creating swap-chains. In many cases, the swap –chains will be creating by the underlying platform (e.g. Android or DirectFB), but where the platform does not support swap-chains (e.g. Nexus), the platform layer will need to do this work.

In addition to the swap-chain buffers, you might find it useful to be able to hold your own arbitrary data associated with the window/swap-chain (labeled Window State in the diagram above). Use `WindowPlatformStateCreate()` and `WindowPlatformStateDestroy()` functions to manage the creation and deletion of this state.

Swap-chain management

When the driver calls `WindowGetInfo()`, it inspects the `swapchain_count` in the returned `BEGL_WindowInfo` structure to determine how many buffers should be present in the swap-chain for the window. Usually this will be 2 or 3 for i.e. for double or triple buffering.

Buffers in the swap-chain can be created either by an external library, such as DirectFB or internally to the platform layer itself. In the latter case, it may be convenient to lazily create the buffers in `DispBufferGet()` as they are requested (see later).

Note that externally created buffers must still respect the size and alignment constraints required by the V3D driver. These constraints should be queried from the driver using the `BufferGetRequirements()` function pointer of the display callbacks. The requirements may change from release to release, so should always be queried dynamically.

`DispBufferGet()` will be called for each buffer in the swap-chain. You should return the appropriate externally created buffer as requested, or create and return the buffers if they do not exist.

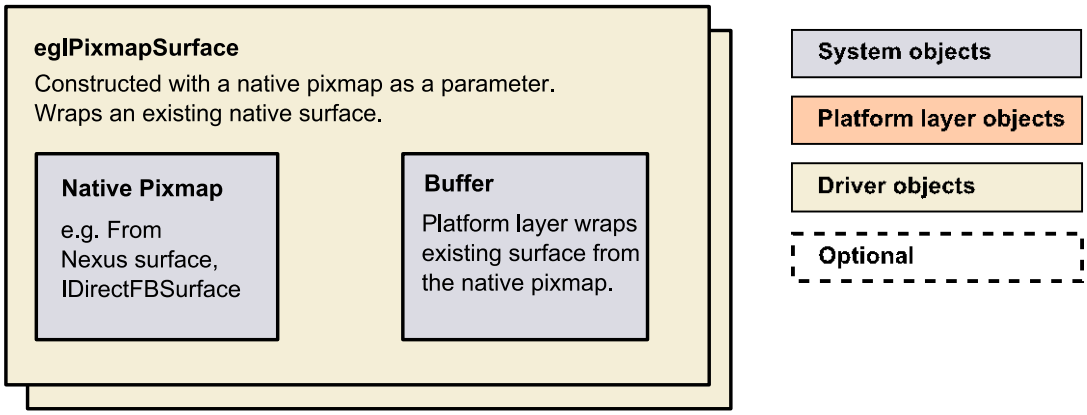
The driver will make calls to your `BufferAccess()` function in order to lock buffers for rendering. When the driver requests that a buffer is locked, you must return the physical device address and cached address of that buffer.

When rendering is complete, the `BufferDisplay()` function will be called. The buffer is now available for display and must be unlocked. The V3D driver will not write to a buffer unless it has the lock.

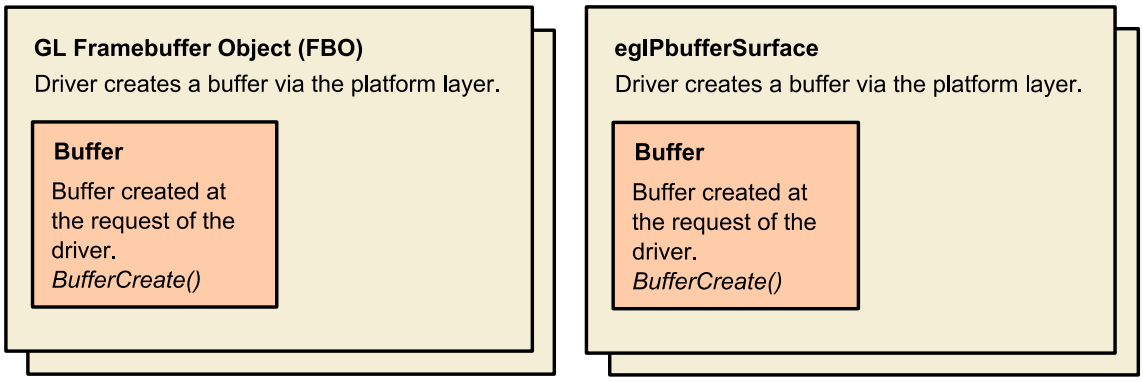
`DispBufferCreate()` is used create buffers such as FBOs and pixmaps. If the platform layer is responsible for creating swap chains, then it is recommended to use `DispBufferCreate()` internally also to create the buffers.

Other render targets

Pixmap surfaces wrap existing native surface formats and can be rendered in a single-buffered fashion. Before using the surface in a native API you must call `glFinish()` to ensure that the rendering is complete. Since this call blocks until done, it is not very efficient; the 3D core cannot render ahead. For this reason, we discourage the use of pixmap rendering.



FBO and Pbuffers are surfaces that are created and used only by GL. The buffers used for this rendering are created via the `BufferCreate()` function.



Required Functions

BEGL_BufferHandle BufferCreate(void *context, BEGL_BufferSettings *settings)

Request creation of an appropriate displayable buffer, which must be created using the constraints in the settings parameter.

The created “BEGL_BufferHandle” is an opaque type, so the platform layer is free to create any kind of object that it requires. For example, in a Nexus platform layer, this function might be implemented to create a NEXUS_Surface (with the correct memory constraints of course), and when is passed to BufferDisplay, the platform layer can simply use it as a NEXUS_Surface. It doesn't have to wrap the memory each time, or perform any lookups.

This function will be used to create buffers for pixmaps, framebuffer objects and pbuffers. If you are not using an externally created swap-chain, it is recommended that it be used to create the swap-chain buffers for each window surface. See BufferGet().

BEGL_BufferHandle BufferGet(void *context, BEGL_BufferSettings *settings)

The driver uses this function to request a buffer from a swap chain. The platform code can create the buffers on demand, or return entries from a pre-existing external swap chain e.g. one created by another software layer such as DirectFB.

If the platform layer is creating its own buffers, it is recommended to use BufferCreate() to do so.

BEGL_Error BufferDestroy(void *context, BEGL_BufferDisplayState *bufferState)

Destroy a buffer previously created with BufferCreate().

The buffer will not be used by the driver after this call has been made, so you can safely delete the resources associated with the buffer.

BEGL_Error BufferAccess(void *context, BEGL_BufferAccessState *bufferAccess)
The driver calls this method to signal that it plans to render to the buffer. The platform layer should do whatever it needs to obtain a pointer to the memory for the surface. This will almost certainly involve some kind of locking mechanism. If the buffer cannot be locked immediately (because it is currently on the display for example) this function should block until the lock can be granted.

BEGL_Error BufferDisplay(void *context, BEGL_BufferDisplayState *state)
The driver calls this method when it has finished rendering to the surface. The platform layer can now unlock the buffer and should present it to the display. If `state->waitVSync` is set, then the call should wait for the next vertical sync before returning.

BEGL_Error BufferGetCreateSettings(void *context, BEGL_BufferHandle buffer, BEGL_BufferSettings *outSettings)
Get information such as width, height, memory pointers etc. about a created window buffer.

BEGL_Error WindowUndisplay(void *context, BEGL_WindowState *windowState)
Flush all pending display of all of the swap-chain buffers in this window until they are all done, then remove the window from display. Should block until complete. Buffers and windows might be destroyed immediately following this call, so ensure the buffers are no longer in use when you return.

Optional Functions

BEGL_Error IsSupported(void *context, BEGL_BufferFormat bufferFormat)
Check if the platform can support the buffer format passed in. A little-endian platform is required to support `BEGL_BufferFormat_eA8B8G8R8`, `BEGL_BufferFormat_eX8B8G8R8` and `BEGL_BufferFormat_eR5G6B5`, so these are not checked. A big-endian platform is required to support `BEGL_BufferFormat_eR8G8B8A8`, `BEGL_BufferFormat_eR8G8B8X8` and `BEGL_BufferFormat_eR5G6B5`. If your platform only supports these formats, you do not need to provide this function.

BEGL_Error GetNativeFormat(void *context, BEGL_BufferFormat bufferFormat, unsigned int *outNativeFormat)
Provide support for the EGL API eglGetConfigAttrib(), which translates an EGL config ID into a platform specific render type. The typical use-case would be to select a config ID and create a matching native window of that type, providing the native window to eglCreateWindowSurface().

BEGL_Error SetDisplayID(void *context, unsigned int displayID, unsigned int *outEglDisplayID)
Called when eglGetDisplay() is called. Used in cases such as X11, so the X display can be passed into the driver.

```
BEGL_Error WindowGetInfo(void *context, BEGL_WindowHandle window,
                         BEGL_WindowInfoFlags flags, BEGL_WindowInfo *outInfo)
```

Called to determine current size of the window referenced by the opaque window handle. This is needed by EGL in order to know the size of a native 'window'.

The following functions are optional, but if you implement one, then the other must also exist.

```
void *WindowPlatformStateCreate(void *context, BEGL_WindowHandle window)
```

Called by the driver prior to any swap-chain buffers being created for the given window.

If you need to store any state per window (swap-chain) then you can allocate your own structure and return it.

The `platformState` of the `BEGL_WindowState` argument passed to most platform layer functions contains the pointer for the associated window that you return from here.

```
BEGL_Error WindowPlatformStateDestroy(void *platformState)
```

Called when your custom window state structure needs to be destroyed. The structure will not be used after this.