# OpenGL ES Quick Start

**April 12, 2012**

**STB_XXX**

## Revision History

| ***Doc Number*** | ***Revision Date*** | ***Description/Author*** |
|---|---|---|
| STB_XXX | 4/08/11 | Initial version. |
|  | 3/12/12 | Revised. |

# Section 1: Introduction

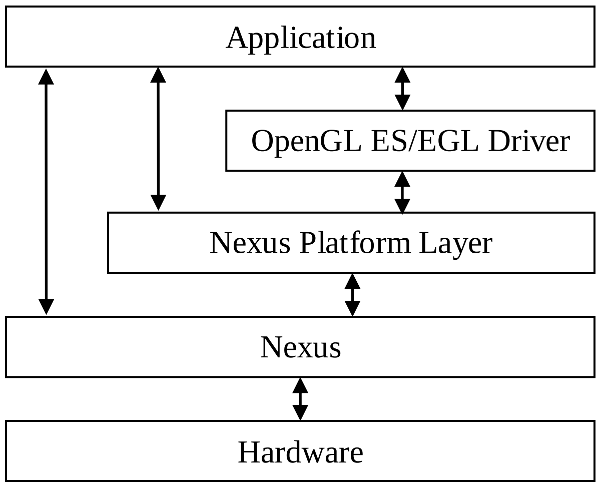
## Overview

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. 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 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.



## 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.

## 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. For this example, we will be using single-process mode, so ensure that the environment variable NEXUS_MODE is not set. Multi-process is 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
export B_REFSW_DEBUG=n
make
```

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 use the “nexus” script in *nexus/bin* thus:

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

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
```

To build a debug version of the cube application:

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

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

# Section 2: Basic Usage

## The Examples

The examples folder contains the following basic applications:

- cube: which was described in the previous section.
- earth_es2: is an animation of the Earth, Moon and a star-field.
- poly_rate and poly_rate2: are simple benchmarks for measuring polygon rates in ES 1.1 and 2.0.

These examples can be built in the same way as for "cube".

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,

- v3ddriver,
- nxpl

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 exclusive mode 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
- 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;
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 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. However, for those cases where it is impractical to migrate to full 3D, the examples folder contains two applications that demonstrate how to mix 2D graphics with 3D graphics:

- cube_composited: demonstrates the recommended approach for mixing 2D and 3D which allows 3D rendering to be overlapped with 2D.
- cube_pixmap: shows an alternative method which is simpler, but much less efficient.

These examples can be built in the same way as for the cube example.

You can also mix 2D and 3D content generated by separate threads or processes using the Nexus surface compositor. See the multi-process section below for more details.

## Cube Composited

The cube composited application mixes 2D and 3D rendering in an efficient way by allowing the 2D and 3D parts of the application to run asynchronously. The application has two threads. The main thread is responsible for rendering 2D and for the main flow of the program. The secondary thread manages the 3D rendering. Because the 3D thread is asynchronous, it can “run-ahead” and be preparing frames in advance whilst the 2D thread only needs to capture 3D data when it is available.

The default platform layer provides the tools to help with building a compositing application. The normal display registration process found in cube is replaced by:

```
CompBufferOnDisplayFunc bufferOnDisplayFunc = 0;

NXPL_CompositingData    compData;
memset(&compData, 0, sizeof(NXPL_CompositingData));

compData.bufferBlit = DisplayBuffer;

NXPL_RegisterNexusCompositedDisplayPlatform(&nxpl_handle, &compData);
bufferOnDisplayFunc = compData.bufferOnDisplay;
```

The `DisplayBuffer()` function will be called when the 3D driver is ready to display a new buffer. It is the responsibility of the application to handle these notifications appropriately. The application is also responsible for using the `bufferOnDisplayFunc` to notify the 3D driver that the buffer has been displayed and can be reused.

In this example, the DisplayBuffer() function simply records the buffer and context information. The 2D thread blits the current 3D buffer onto the display and then during the vsync-callback it signals the 3D driver via `bufferOnDisplayFunc` that the buffer is ready for reuse.

## Cube Pixmap

Pixmaps are off-screen surfaces which can be rendered into, independently of the display, and are also 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 that to synchronize the 3D and 2D drawing, it is necessary to execute a `glFinish()`, which unlike `eglSwapBuffers()`, is a blocking API. The GL will wait until all rendering has completed before returning from the finish. Using pixmaps will inevitably result in under-utilization of the 3D hardware.

Note that if an application wishes to render to an off-screen buffer and use the results exclusively within OpenGL ES, as a texture for example, then it should use “framebuffer objects” rather than pixmaps.

The cube pixmap application is similar to the basic cube application except that it draws the results into a “pixmap” surface and then copies this into a 2D Nexus display. The application must first obtain a config which supports pixmap surfaces. A pixmap surface can then be created thus:

```
void                *nxpl_pixmap_handle;
NEXUS_SurfaceHandle nx_gl_surface;
BEGL_PixmapInfo     pix_info;

memset(&pix_info, 0, sizeof(BEGL_PixmapInfo));
pix_info.width  = w;
pix_info.height = h;
pix_info.format = BEGL_BufferFormat_eA8B8G8R8;

NXPL_CreateCompatiblePixmap(nxpl_handle, &nxpl_pixmap_handle, &nx_gl_surface, &pix_info);

pixmap_surface = eglCreatePixmapSurface(display, config, nxpl_pixmap_handle, NULL);
```

The surface can be drawn into using `eglMakeCurrent()`, as for a normal render surface and can be accessed by Nexus functions using the returned Nexus surface handle.

This application also uses the “discard framebuffer” extension which can be used to prevent the driver from preserving the multisample, depth and stencil buffers after a `glFinish()`.

# Multi-process

OpenGL ES applications can be run in multiple processes. The Nexus surface compositor is used to manage the display. A separate server application is responsible for marshalling the results from the graphics processes and composing them for display. The server process is akin to a window manager for a desktop environment. In the examples below, we use the “blit_server” example application which is an example implementation using the Nexus surface compositor.

Multi-process 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.

To use multi-process mode, compile the blit-server application and the example client applications e.g. the basic examples cube and earth_es2 as describe below. Run the server, and then run the clients specifying different client IDs for each application. Each client is allocated a different region of the display by the blit-server (depending on the client ID).

You will find it helpful to have several telnet sessions in order to run the client applications separately from the server.

## User Mode

To build the client applications:

```
export B_REFSW_DEBUG=n
export surface_compositor=y
export NEXUS_MODE=client
export CLIENT=y
cd rockford/applications/khronos/v3d/nexus/cube
make clean
make
cd rockford/applications/khronos/v3d/nexus/earth_es2
make
```

To build the server application:

```
export B_REFSW_DEBUG=n
unset NEXUS_MODE
unset CLIENT
export surface_compositor=y
cd nexus/build
make clean
make
cd nexus/examples/multiprocess
make blit_server
```

To run the multi-process examples, in one telnet or console:

```
cd nexus/bin
nexus blit_server
```

And in second telnet:

```
./cube client=0 &
./earth client=1 &
```

## Kernel Mode

To build the client applications:

```
export B_REFSW_DEBUG=n
export surface_compositor=y
export NEXUS_MODE=proxy
export CLIENT=y
cd rockford/applications/khronos/v3d/nexus/cube
make clean
make
cd rockford/applications/khronos/v3d/nexus/earth_es2
make
```

To build the server application:

```
export B_REFSW_DEBUG=n
export surface_compositor=y
export NEXUS_MODE=proxy
unset CLIENT
cd nexus/build
make clean
make
cd nexus/examples/multiprocess
make blit_server
```

To run the server application in one telnet session:

```
cd nexus/bin
nexus blit_server &
```

To run the client applications in a second telnet session:

```
cd nexus/bin
./cube client=0 &
./earth client=1 &
```