Qualcomm Technologies, Inc.

Qualcomm® Adreno™ OpenGL ES

Developer Guide

80-NU141-1 B

May 1, 2015

Questions or comments: https://support.cdmatech.com/

The cover image is taken from the “Palazzo” demo developed by the Advanced Content Group at Qualcomm Technologies, Inc. The demo is running on a Qualcomm Snapdragon 810 using OpenGL ES 3.1. and depicts an immersive 3D environment, with near photo-realistic rendering including dynamic lighting, reflections, and hardware tessellated objects.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	Dec 2014	Initial release
B	May 2015	Updated for Adreno 4xx to Chapter 1 and Chapter 3; added additional compression information to Section 9.3

Contents

Figures

Tables

Equations

1 Overview

This document is a guide for developing and optimizing OpenGL ES applications for Android on platforms containing Qualcomm® Adreno™ GPUs. OpenGL ES is a subset of the OpenGL API for developing 2D and 3D graphics, designed for use on embedded systems, which typically are constrained by processor capability, memory limitation, and power consumption limits.

The document introduces the basics of OpenGL ES development with Adreno, detailing how to set up a development environment, and provides walkthroughs of sample applications. It also describes how to use debugging and profiling tools.

Detailed descriptions of the Adreno-specific implementations of OpenGL ES APIs are provided, along with an introduction to the Android Extension Pack (AEP), and the developer tools provided by the Adreno SDK.

Good design practice with OpenGL ES is discussed, followed by advice on how to optimize applications specifically for the Adreno hardware.

This document is intended for application developers, with a good working knowledge of a modern 3D graphics API such as OpenGL ES, OpenGL, or Microsoft Direct3D. It is not intended as a primer for 3D graphics.

1.1 Adreno GPU

The Adreno GPU is built in as part of the all-in-one design of the Qualcomm® Snapdragon™ processors. Accelerating the rendering of complex geometries allows the processors to meet the level of performance required by the games, user interfaces, and web technologies present in mobile devices today.

The Adreno GPU is built purposely for mobile APIs and mobile device constraints, with an emphasis on performance and efficient power use.

The original Adreno 130 variant provides support only for OpenGL ES 1.1. The Adreno 2xx series and onward supports OpenGL ES 2.0. The Adreno 3xx series adds support for OpenGL ES 3.0 and OpenCL. Adreno 4xx adds support for OpenGL ES 3.1[1] and the AEP.

This section outlines the various technologies and subsystems provided by the Adreno GPU to support the graphics developer. Best practice for using these is discussed in later chapters.

[1] Product is based on a published Khronos specification and is expected to pass the Khronos Conformance Testing Process when available. Current conformance status can be found at www.khronos.org/conformance.

1.1.1 Texture features

Multiple textures

Multiple texturing or *multitexturing* is the use of more than one texture at a time on a polygon. Adreno 4xx supports up to 32 total textures in a single render pass, meaning up to 16 textures in the fragment shader and up to 16 textures at a time for the vertex shader. Effective use of multiple textures reduces overdraw significantly, saves Algorithmic Logic Unit (ALU) cost for fragment shaders, and avoids unnecessary vertex transforms.

To use multiple textures in applications, refer to the multitexture sample in the Adreno SDK OpenGL ES tutorials.

Video textures

More games and graphics applications today require video textures, which consist of moving images that are streamed in real time from a video file. Adreno GPUs support video textures.

Video textures are a standard API feature in Android today (Honeycomb or later versions). See the Android documentation for further details on surface textures at http://developer.android.com/reference/android/graphics/SurfaceTexture.html.

Apart from using the standard Android API as suggested, if an application requires video textures, the standard OpenGL ES extension can also be used. See http://www.khronos.org/registry/gles/ extensions/OES/OES_EGL_image.txt.

Figure 1-1 Video texture example

Texture compression

Texture compression can significantly improve the performance and load time of graphics applications since it reduces texture memory and bus bandwidth use. Compressed textures can be created using the Adreno Texture Compression and Visualization Tool and subsequently used by an OpenGL ES application.

Important compression texture formats supported by Adreno 3xx are:

- ATC – Proprietary Adreno texture compression format (for RGB and RGBA)
- ETC – Standard OpenGL ES 2.0 texture compression format (for RGB)
- ETC2 – Texture compression format that is supported in the OpenGL ES 3.0 API (for R, RG, RGB, and RGBA)

Adreno 4xx adds support for ASTC LDR compression, which is made available through the Android Extension Pack.

To learn more about the use of texture compression, see the *Compressed Texture* tutorial in the Adreno SDK.

Floating point textures

Adreno 2xx, 3xx, and onward support the same texturing features, including:

- Texturing and linear filtering of FP16 textures via the GL_OES_texture_half_float and GL_OES_texture_half_float_linear extension
- Texturing from FP32 textures via GL_OES_texture_float

Through the OpenGL ES 3.0 API, Adreno 3xx and onward also includes rendering support for FP16 (full support) and FP32 (no blending).

Cube mapping with seamless edges

Cube mapping is a fast and inexpensive way of creating advanced graphic effects, like environment mapping. Cube mapping takes a three-dimensional texture coordinate and returns a texel from a given cube map (similar to a sky box).

Adreno 3xx and onward supports seamless-edge support for cube map texture sampling.

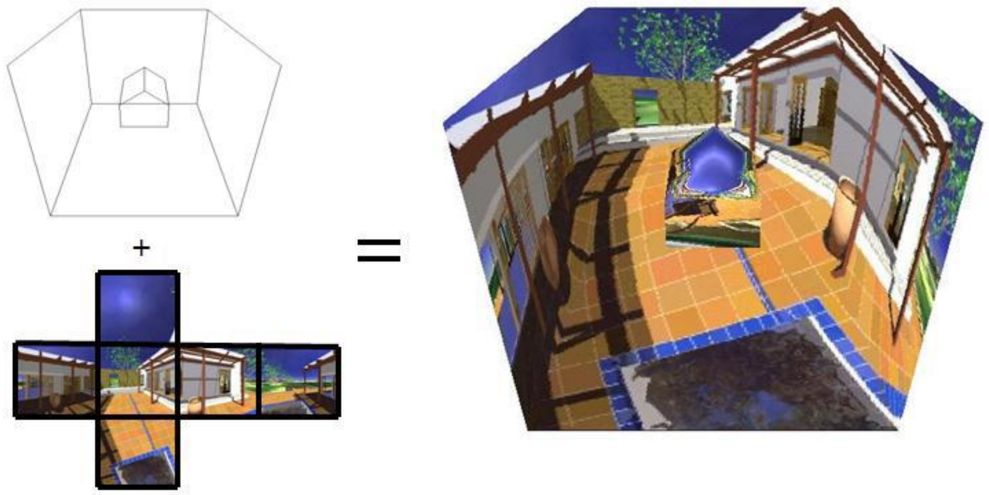

Figure 1-2 Cube mapping

3D textures

In addition to 2D textures and cube maps, there is a ratified OpenGL ES 2.0 extension for 3D textures called GL_OES_texture_3D. This extension allows 3D texture initialization and use for volumetric rendering purposes. This is a core functionality starting with OpenGL ES 3.0.

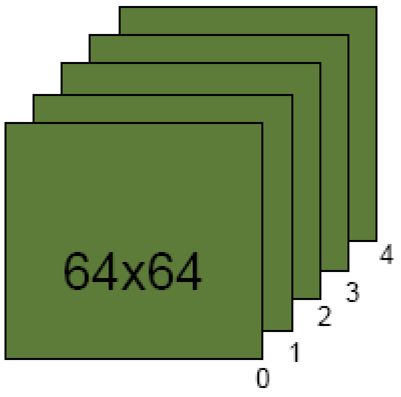

Figure 1-3 3D texture

Large texture size

Adreno 330 supports texture sizes up to 8192x8192x8192. Depending on memory availability, Adreno 420 can address textures of resolution up to 16384x16384x16384.

sRGB textures and render targets

sRGB is a standard RGB color space created cooperatively by Hewlett-Packard and Microsoft in 1996 for use on monitors, printers, and the Internet. Smartphone and tablet displays today also assume sRGB (nonlinear) color space. To get the best viewing experience with correct colors, it is important that the color space for render targets and textures matches the color space for the display, which is sRGB. Unfortunately, OpenGL ES assumes linear or RGB color space by default. As Adreno 3xx and 4x support sRGB color space for render targets as well as textures, it is possible to ensure a correct color viewing experience.

PCF for depth textures

Adreno 3xx and 4x have hardware support for the OpenGL ES 3.0 feature of Percentage Closer Filtering (PCF). A hardware bilinear sample is fetched into the shadow map texture, which alleviates the aliasing problems that can be seen with shadow mapping in real time applications.

1.1.2 Visibility processing

Early Z rejection

Early Z rejection provides a fast occlusion method with the rejection of unwanted render passes for objects that are not visible (hidden) from the view position. Adreno 3xx and 4xx can reject occluded pixels at up to 4x the drawn pixel fill rate.

Figure 1-4 shows a color buffer represented as a grid, and each block represented as a pixel. The rendered pixel area on this grid is colored black. The Z-buffer value for these rendered black pixels is 1. If trying to render a new primitive onto the same pixels of the existing color buffer that has the Z-buffer value of 2 (as shown in the second grid with green blocks), the conflicting pixels in this new primitive will be rejected as shown in the third grid representing the final color buffer. Adreno 3xx and 4x can reject occluded pixels at up to four times the drawn pixel fill rate.

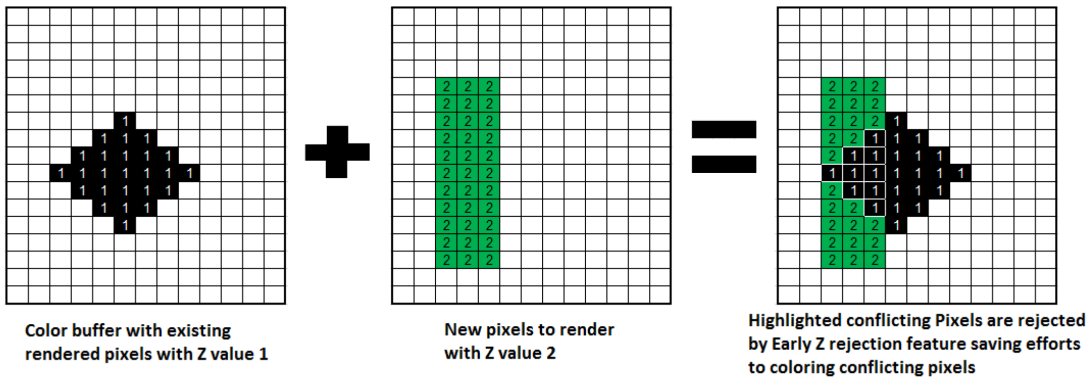
Color buffer with existing rendered pixels with Z value 1
New pixels to render with Z value 2
Highlighted conflicting Pixels are rejected by Early Z rejection feature saving efforts to coloring conflicting pixels

Figure 1-4 Early Z rejection

To get maximum benefit from this feature, QTI recommends drawing a scene with primitives sorted out from front-to-back; i.e., near-to-far. This ensures that the Z-reject rate is higher for the far primitives, which is useful for applications that have high-depth complexity.

FlexRender™ (hybrid deferred and direct rendering mode)

QTI introduced its new FlexRender solution as part of Adreno 3x and 4x. FlexRender refers to the ability of the GPU to switch between indirect rendering (binning or deferred rendering) and direct rendering to the frame buffer.

There are advantages to both the direct and deferred rendering modes. The Adreno 3x and 4x GPUs were designed to maximize performance by switching between the two modes in a dynamic fashion. This works by the GPU analyzing the rendering for a given render target and selecting the mode automatically.

The deferred mode rendering mechanism of the Adreno GPU uses tile-based rendering and implements a binning approach is used to create bins of primitives are processed in each tile. The first pass associates each primitive with a set of BinIDs and back-facing information. This pass is done once per frame. In the second pass, these BinIDs are used to trivially reject the primitives that fall outside the current bin being rendered and perform early back-face culling.

The second pass runs once per bin. Each bin is rendered to the GMEM. Then, each bin is resolved to the render surface in memory. The deferred mode rendering mechanism is shown in further detail in Figure 1-5.

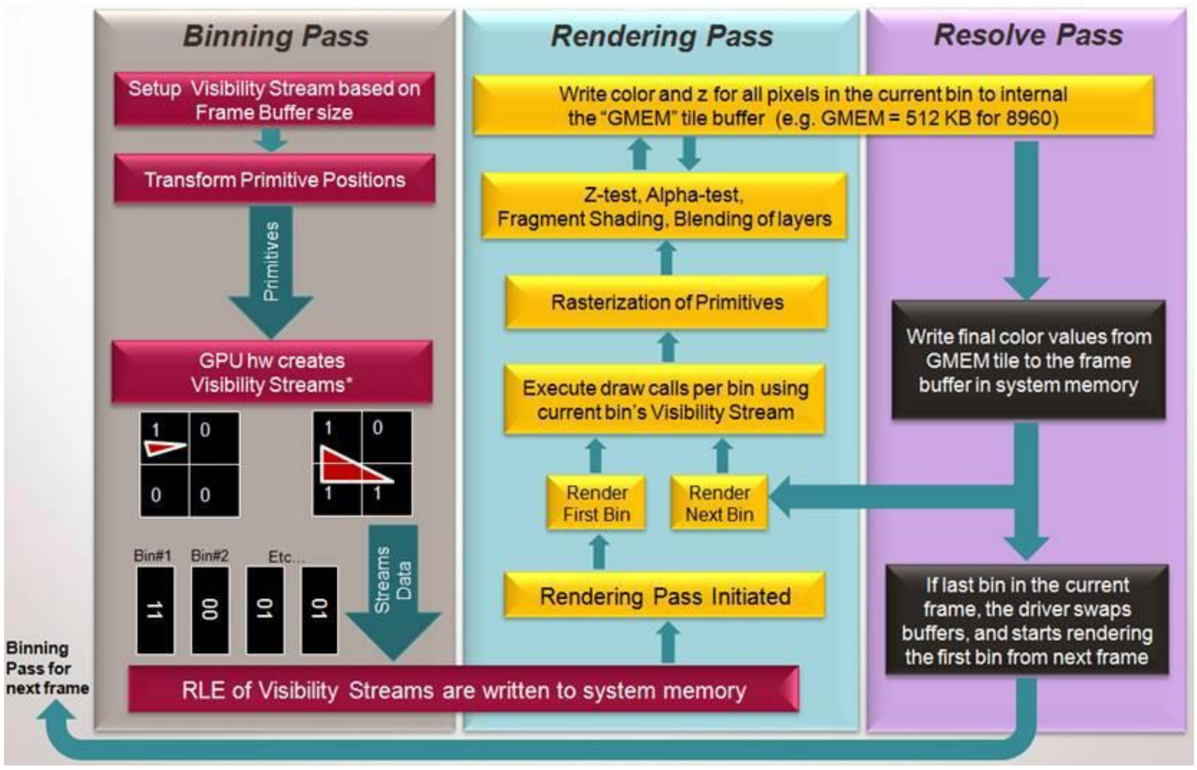
Binning Pass
Setup Visibility Stream based on Frame Buffer size
Transform Primitive Positions
Primitives
GPU hw creates Visibility Streams*
1 0 0 0
1 0 1 1
Bin#1 Bin#2 Etc...
11 00 01 01
Streams Data
Binning Pass for next frame
RLE of Visibility Streams are written to system memory
Rendering Pass
Write color and z for all pixels in the current bin to internal the "GMEM" tile buffer (e.g. GMEM = 512 KB for 8960)
Z-test, Alpha-test, Fragment Shading, Blending of layers
Rasterization of Primitives
Execute draw calls per bin using current bin's Visibility Stream
Render First Bin
Render Next Bin
Rendering Pass Initiated
Resolve Pass
Write final color values from GMEM tile to the frame buffer in system memory
If last bin in the current frame, the driver swaps buffers, and starts rendering the first bin from next frame

Figure 1-5 Deferred style of rendering with Adreno 3xx and 4x

1.1.3 Shader support

Unified shader architecture

All Adreno GPUs support the Unified Shader Model, which allows for use of a consistent instruction set across all shader types (vertex and fragment shaders). In hardware terms, Adreno GPUs have computational units, e.g., ALUs, that support both fragment and vertex shaders.

Adreno 4xx uses a shared resource architecture that allows the same ALU and fetch resources to be shared by the vertex shaders, pixel or fragment shaders, and general purpose processing. The shader processing is done within the unified shader architecture, as shown in Figure 1-6.

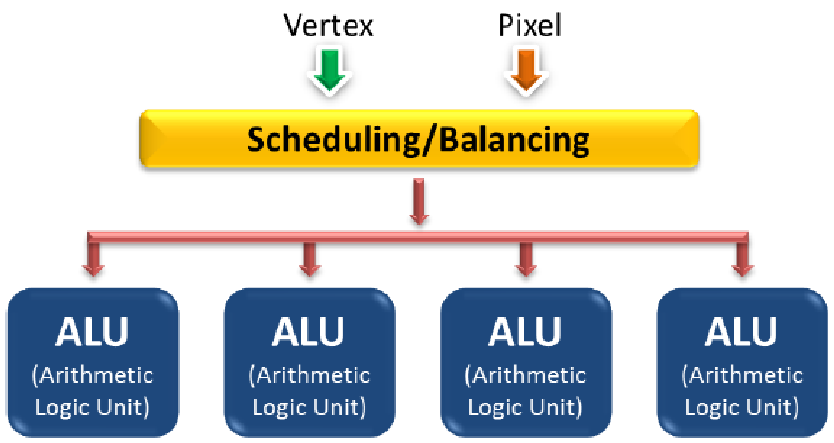
Vertex
Pixel
Scheduling/Balancing
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)

Figure 1-6 Unified shader architecture

Figure 1-6 shows that vertices and pixels are processed in groups of four as a vector, or a thread. When a thread stalls, the shader ALUs can be reassigned.

In unified shader architecture, there is no separate hardware for the vertex and fragment shaders, as shown in Figure 1-7. This allows for greater flexibility of pixel and vertex load balances.

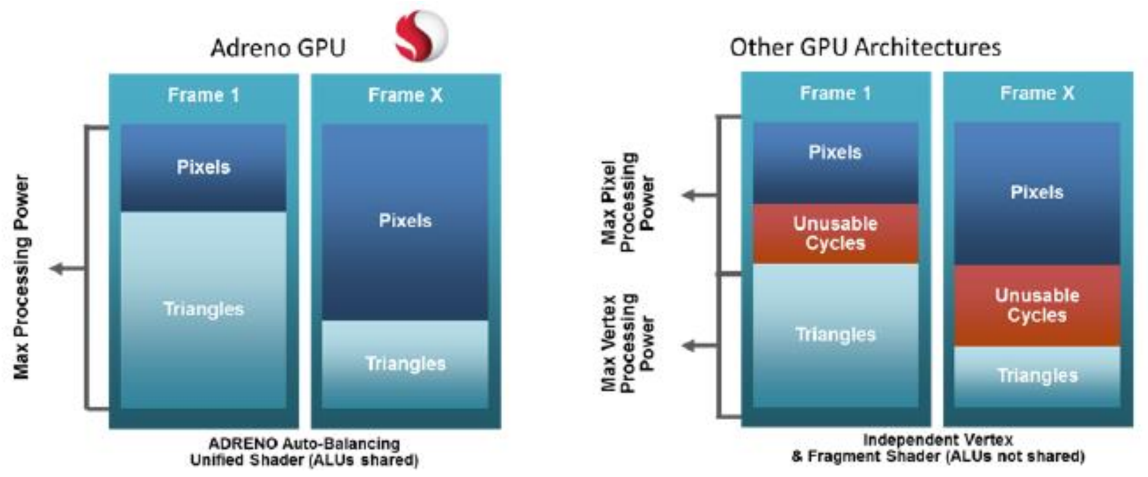

Figure 1-7 Flexibility in shader resources – Unified shader architecture

The Adreno 4xx shader architecture is also multithreaded, e.g., if a fragment shader execution stalls due to a texture fetch, the execution is given to another shader. Multiple shaders are accumulated as long as there is room in the hardware.

No special steps are required to use the unified shader architecture. The Adreno GPU intelligently makes the most efficient use of the shader resources depending on scene composition.

Scalar architecture

Adreno 4xx has a scalar component architecture. The smallest component Adreno 4xx can support natively is a scalar component. This results in more efficient hardware resource use for processing scalar components, and it does not waste a full vector component to process the scalar.

The scalar architecture of Adreno 4xx can be twice as power-efficient and deliver twice the performance for processing a fragment shader that uses medium-precision floating point (mediump), compared to other mobile GPUs today, which use vector architecture. For Adreno 4x, mediump is a 16-bit floating point and highp is a 32-bit floating point.

1.1.4 Other supported features

Index types

A geometry mesh can be represented by two separate arrays, one array holding the vertices, and another holding sets of three indices into that array, which together define a triangle.

Adreno 4xx natively supports 8-bit, 16-bit, and 32-bit index types. Most mobile applications use 16-bit indices.

Multisample anti-aliasing (MSAA)

Anti-aliasing is an important technique for improving the quality of generated images. It reduces the visual artifacts of rendering into discrete pixels.

Among the various techniques for reducing aliasing effects, multisampling is efficiently supported by Adreno 4x. Multisampling divides every pixel into a set of samples, each of which is treated like a "mini-pixel" during rasterization. Each sample has its own color, depth, and stencil value. And those values are preserved until the image is ready for display. When it is time to compose the final image, the samples are resolved into the final pixel color. Adreno 4xx supports the use of two or four samples per pixel.

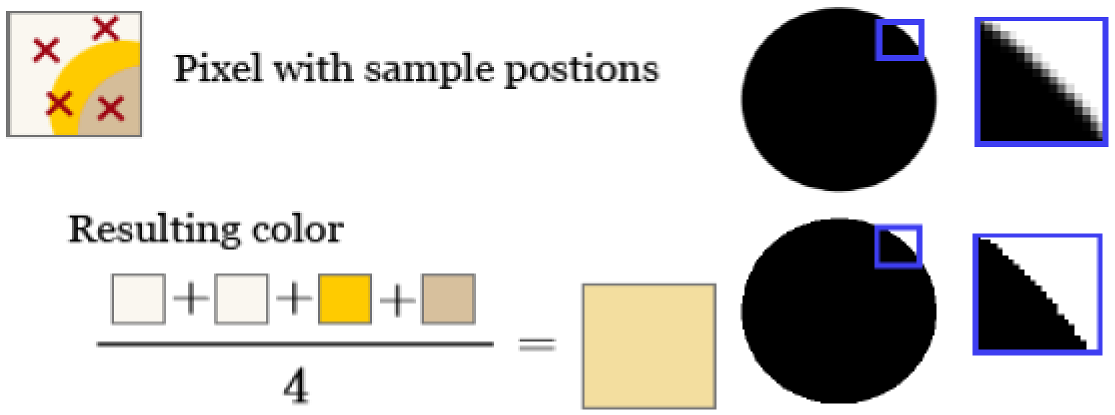

Figure 1-8 MSAA

Vertex texture access or vertex texture fetch

With the advantage of having shared resources to process vertex and fragment shaders, in the Adreno GPUs the vertex shader has direct access to the texture cache. It is simple to implement vertex texture algorithms for function definitions, displacement maps, or lighting level-of-detail (LoD) systems on Adreno GPUs. Vertex texture displacement is an advanced technique that is used to render realistic water in games on a desktop and for consoles. The same can now be implemented in applications running on Adreno GPUs.

The following is an example of how to do a texture fetch in the vertex shader:

```
/////vertex shader
attribute vec4 position;
attribute vec2 texCoord;
uniform sampler2D tex;
void main() {
float offset = texture2D(tex, texCoord).x;
…..
gl_Position = vec4(….);
}
```

1.1.5 Adreno APIs

Adreno 4xx supports Khronos standard APIs including:

- OpenGL ES 1.x (fixed function pipeline)
- OpenGL ES 2.0 (programmable shader pipeline)
- OpenGL ES 3.0
- EGL
- OpenCL 1.1e

Adreno 4xx additionally supports:

- OpenGL ES 3.1 (most recent version of this API)
- AEP
- OpenCL 1.2full

Along with the OpenGL ES APIs, the extensions to these APIs are also supported.

In addition to the Khronos standard APIs, Adreno 4xx supports Microsoft Direct3D 11 API with Feature Level 9_3. Adreno 4xx supports Direct3D 11 with Feature Level 11_2. Discussion of these APIs is outside the scope of this document.

1.2 OpenGL ES

OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems. It consists of well-defined desktop OpenGL subsets, creating a flexible and powerful low-level interface between software and graphics acceleration.

1.2.1 Open GL ES versions

The following sections outline the different versions of OpenGL ES that are available and how they relate to OpenGL. They also detail the main functional differences between them and optimizations that can be utilized by developers.

OpenGL ES 1.x

Defined relative to the OpenGL 1.5 specification, OpenGL ES 1.x is designed for fixed function hardware, and emphasizes hardware acceleration of the API. It provides enhanced functionality, improved image quality, and optimizations to increase performance while reducing memory bandwidth usage to save power.

For more about the APIs and specifications, see https://www.khronos.org/opengles/1_X/.

OpenGL ES 2.x

Defined relative to the OpenGL 2.0 specification, OpenGL ES 2.x is designed for programmable hardware and focuses on a programmable 3D graphics pipeline, providing the ability to create shader and program objects and to write vertex and fragment shaders. It does not support the fixed function transformation and fragment pipeline of OpenGL ES 1.x.

For more information, see https://www.khronos.org/api/opengles/2_X.

OpenGL ES 3.x

OpenGL ES 3.x is backwards compatible with 2.x and 1.x. It provides enhancements to the rendering pipeline to enable acceleration of advanced visual effects, a new version of the OpenGL ES Shading Language, and an enhanced texturing functionality, among other things.

For a complete description of the API, see https://www.khronos.org/api/opengles/3_X.

1.3 About Android

Android is a mobile operating system based on the Linux kernel and is currently developed by Google.

1.3.1 OpenGL ES support on Android

Android supports several versions of the OpenGL ES API.

- OpenGL ES 1.0 and 1.1 – This API specification is supported by Android 1.0 and higher.
- OpenGL ES 2.0 – This API specification is supported by Android 2.2 (API level 8) and higher.
- OpenGL ES 3.0 – This API specification is supported by Android 4.3 (API level 18) and higher.
- OpenGL ES 3.1 – This API specification is supported by Android 5.0 (Lollipop).

1.3.2 Android and OpenGL ES on Adreno

The Adreno GPU versions support the use of different levels of the OpenGL ES specification. For each level of the OpenGL ES specification, there is also a minimum version of the Android OS required, e.g., to use OpenGL ES 3.0, it requires at least Adreno 3xx and at least Android 4.3 (Jelly Bean). Table 1-1 lists these requirements.

Table 1-1 Adreno, OpenGL ES, and Android versions

Adreno ver	OpenGL ES ver supported	Android ver required
Adreno 1xx	1.1	Android 1.0 (Alpha)
Adreno 2xx	2.0	Android 2.2 (Froyo)
Adreno 3xx	3.0	Android 4.3 (Jelly Bean)
Adreno 4xx	3.1	Android 5.0 (Lollipop)

2 OpenGL ES 2.0 with Adreno

2.1 Development environment

Before developing OpenGL ES applications, it is necessary to set up a suitable development environment. A development system is needed, which can be based on Windows, Linux, or OSX. There must also be a target system for testing the application. For the purposes of this document, that means an Android device or emulator.

2.1.1 Development system

There are a number of software pieces that are required to create the development toolchain. The required software packages are as follows.

Adreno SDK

The Adreno SDK offers an OpenGL ES emulator, demos and tutorials, and an SDK browser that allows the running, building, and deploying of these samples to the device with a single click.

NOTE: The OpenGL ES emulator relies on the implementation of desktop OpenGL on the host system, as shown in Table 2-1.

Table 2-1 OpenGL versions required by the emulator

OpenGL ES ver	Desktop OpenGL ver needed
OpenGL ES 2.0	OpenGL 2.0
OpenGL ES 3.0	OpenGL 3.3
OpenGL ES 3.1	OpenGL 4.3

Tip

Be sure to update to the latest graphics driver versions on the development system.

Android developer tools

When developing using Eclipse, download and install Android Developer Tools (ADT), which is a plug-in for Eclipse that provides a full suite of Android-specific tools, including XML editors, GUI builders, and debugging and profiling tools for both the emulator and the device.

NOTE: ADT comes in a bundle that includes the core Android SDK. The Android SDK provides the API libraries and developer tools necessary to build, test, and debug applications for Android. If ADT is not installed, the standalone version of the Android SDK is necessary instead.

Android NDK

The Android NDK is a toolset that allows code implementation using native languages, such as C and C++. Some of the Adreno SDK samples are developed in this manner, so it is necessary to install the NDK to take advantage of these samples.

Apache Ant

Apache Ant is a toolset used to aid in the building of software packages, most usually Java-based. It is used in Android to create APK packages, and is needed as part of any general Android development environment

Java development kit

The Java Development Kit (JDK) is a prerequisite for running Ant and many other development tools.

2.1.2 Target system

When deploying and testing an application, there are three possibilities as discussed here.

Adreno SDK emulator

If there is no available hardware, or if it is necessary to test against a pure implementation of the OpenGL ES API, then the Adreno SDK emulator, along with the associated profiler tool, allows for debugging and optimization of applications quickly and without large hardware outlay costs.

Existing Adreno-based consumer devices

The Adreno GPU is used in a broad range of tablet and mobile phone devices from major manufacturers. There are too many devices to list here, but it is not difficult to find a modern Android device running any given version of the Adreno GPU.

QTI Snapdragon MDP

To gain access to the very latest Adreno GPUs and the most advanced multimedia technologies, it is necessary to go beyond the normal mobile phone and tablet devices available in the consumer marketplace.

QTI provides a mobile development board designed specifically to aid development of OpenGL ES applications on Adreno. The Snapdragon Mobile Development Platform (MDP) offers both mobile and tablet options, and provides early access to a high-performance Android platforms for development, testing, and software optimization. The development devices contain preinstalled development and optimization software, e.g., the Snapdragon performance visualizer, which allows performance monitoring, profiling, and debugging. This helps to easily locate and resolve performance bottlenecks.

For more information about the currently available MDP devices, see https://developer.qualcomm.com/mobile-development/development-devices/snapdragon-mobile-development-platform-mdp.

2.2 Setup instructions

This section lists the steps needed to install the software toolchain for building OpenGL ES applications for Android. This material also provides information and instructions on how to set up a working toolchain, which can lead to the building, installation, and running of any sample application in the Adreno SDK.

2.2.1 Android development on Windows

These instructions cover the setup of a development environment on a Windows x64 development system with an Android target device. If a different platform is necessary for the development system, the details of the installation may differ.

Install the Adreno SDK together with a number of Android development packages. These instructions aim to get a development project up and running in the shortest time possible.

2.2.1.1 Set up the Adreno SDK

1. Download the Adreno SDK from the Qualcomm Developer Network website at https://developer.qualcomm.com/download/.
2. Extract all files to a folder, e.g., C:\AdrenoSDK_Windows.
3. Follow the instructions in the extracted README.txt file. Make sure the packages specified are installed before continuing.
4. Execute AdrenoSDK_Installer.exe. Choose a folder name such as AdrenoSDK in which to install the SDK.
5. Run AdrenoSDK\Bin\Browser.exe to launch the SDK browser. Use this to navigate to the SDK documentation and sample applications.

2.2.1.2 ADT

1. Download the Windows 64-bit ADT bundle from the Android Developers site at https://developer.android.com/sdk/index.html.
2. Extract all files into the Android directory, e.g., C:\Android\adt-bundle-windows-x86_64-20140702.

NOTE: Choose the latest bundle when downloading. The dated version number will likely be different.

3. Add the following to the Windows PATH environment variable:
   a. C:\Android\adt-bundle-windows-x86_64-20140702\sdk\tools
   b. C:\Android\adt-bundle-windows-x86_64-20140702\sdk\platform-tools

2.2.1.3 Android NDK

1. Download the Windows 64-bit NDK from the Android Developers site at http://developer.android.com/tools/sdk/ndk/index.html.
2. Extract all files into the Android directory, e.g., C:\Android\android-ndk-r9b.

3. Add the environment variable ANDROID_NDK_ROOT=C:\Android\android-ndk-r9b to the Windows system.

NOTE: This path may change depending on the latest version number of the NDK.

4. Add this to the Windows PATH environment variable %ANDROID_NDK_ROOT%.

2.2.1.4 JDK

1. Download the Windows x64 Oracle JDK from the Oracle Technology Network site at http://www.oracle.com/technetwork/java/javase/downloads/.
2. Install the JDK to C:\Program Files\Java.
3. Add this environment variable to the Windows system JAVA_HOME=C:\Program Files\Java\jdk1.8.0_05.

NOTE: This path may change depending on the latest version number of the JDK.

4. Add this entry to the PATH variable %JAVA_HOME%\bin.

2.2.1.5 Ant

1. Download Ant from the Apache Ant website at http://ant.apache.org/bindownload.cgi.
2. Extract all files to C:\Ant\apache-ant-1.9.4.
3. Add the environment variable ANT_HOME=C:\Ant\apache-ant-1.9.4.

NOTE: This path may change depending on the latest version number of Ant.

4. Add this entry to the PATH variable %ANT_HOME%\bin.

2.2.1.6 Build and run sample application

1. Open a command prompt.
2. Navigate to the Android\jni directory in one of the Adreno SDK sample applications, e.g., cd\AdrenoSDK\Development\Samples\OpenGLES30\DepthOfField\Android\jni.
3. Run the ndk-build command.
4. Navigate up one level to the Android directory (cd ..).
5. Run the InstallAssets.bat script to copy the sample assets.
6. Remaining in the Android directory, use the Android command to update the project for Ant Android update project -p -t android-XX, where android-XX is the Android SDK version installed, e.g., android-20.
7. Use Ant to create the APK for the sample application ant debug or ant release.
8. Install the sample application on the connected device, i.e., ant installed or ant install.
9. Run the application on the device.

2.3 Walkthrough of sample applications

This section discusses some key tasks that may be faced in the development of an OpenGL ES application. In particular:

- How to set up an OpenGL ES context under Android
- How to detect the Adreno GPU and determine its version
- How to detect available OpenGL ES extensions
- How to implement basic Phong-Blinn lighting
- How to retrieve the values of OpenGL ES constants for the rendering context

This section illustrates these points by referencing the source code of one of the example applications included with the Adreno SDK called *Lighting*. This section also illustrates the other points with code snippets.

2.3.1 Create an ES 2.0 context on Android

When developing for the Android platform, an OpenGL ES context must be created using the EGL API. In the case of the sample applications provided in the Adreno SDK, this is done by common framework code, which is shared across the various sample applications. This is handled by the Android implementation of the framework as discussed in this section. Some basic information on EGL for understanding the code is also provided. For more detailed information about EGL, see:

- EGL Reference Pages at http://www.khronos.org/registry/egl/sdk/docs/man/
- EGL Specification at http://www.khronos.org/registry/egl/specs/eglspec.1.5.pdf

The initialization process is handled by the CFrmAppContainer::InitializeEgl method, implemented in the file SDK\Development\Samples\Framework\Android\FrmApplication_Platform.cpp. If the method returns TRUE, the initialization was performed successfully and an OpenGL ES rendering context has been bound to the calling thread.

This method first initializes an EGLDisplay instance to represent the default display. EGLDisplay is an abstraction of a display on which graphics may be drawn. Each EGLDisplay instance usually corresponds to a single physical screen. All other EGL objects are children of an EGLDisplay instance. For this, the OpenGL ES context is represented by an EGL object of type EGLContext, so it is necessary to initialize an EGLDisplay before proceeding.

```
EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(display,
NULL,    /* major */
              NULL);   /* minor */
```

NOTE: The last two arguments of eglInitialize are optional. If pointers are provided to EGLint variables, they are filled in with the major and minor version numbers of the EGL implementation provided by the driver.

Before creating an OpenGL ES context, answer a few questions:

- What capabilities should be provided by the default frame buffer?
- How many bits should its color buffer use per component?

- Should it include a depth buffer? If so, how many bits should it use per pixel?
- Should it include a stencil buffer? If so, how many bits should it use per pixel?
- Which OpenGL ES version is needed?
- To where should the output from the rendering process be directed? Should it be to a window, or to some kind of off-screen render target?

The underlying hardware usually supports rendering to a number of different frame buffer configurations. To query which of these match the needs of the application, the EGL implementation must provide a list of EGLConfig instances that represent supported configurations.

The code snippet below defines an "attribute list", which is a key/value array terminated by a single EGL_NONE entry. The attribute list specifies requirements for the frame buffer configuration. Here, only a handful of attributes are specified. There are many other properties that could be included in the list. Since they are left out, EGL assumes that they take on the default values defined in the EGL specification.

The attribute list is passed as one of the parameters to the EGL API function eglChooseConfig. This function returns a list of EGLConfig instances that meet the requirements. These are sorted on best match to the requested attributes. If the list size is limited to a single entry, it is guaranteed to retrieve the best matching configuration.

```
EGLConfig config;
EGLint    configAttribs[] =
{
    EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
    EGL_RED_SIZE,        5,
    EGL_GREEN_SIZE,      6,
    EGL_BLUE_SIZE,       5,
    EGL_DEPTH_SIZE,      16,
    EGL_STENCIL_SIZE,    8,
#ifdef _OGLES3
    // this bit opens access to ES3 functions on
    // QCOM hardware pre-Android support for ES3
    EGL_RENDERABLE_TYPE, EGL_OPENGL_ES3_BIT_KHR,
#else
    EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
#endif
    EGL_NONE /* terminator */
};
eglChooseConfig(display,
configAttribs,                &config,
                1,          /* config_size */
               &numConfigs);
```

Also make sure that the dimensions and the pixel format of the native window buffers matches the image data that is rendered. Since it is working with native code, do this by calling ANativeWindow_setBuffersGeometry. The native visual ID is retrieved from the selected configuration and defines the pixel format in a way that is guaranteed to be understood by ANativeWindow_setBuffersGeometry.

```
eglGetConfigAttrib(display,
config,
EGL_NATIVE_VISUAL_ID,
                  &format);
ANativeWindow_setBuffersGeometry(
m_pAndroidApp->window,     ANativeWindow_getWidth
(m_pAndroidApp->window),
     ANativeWindow_getHeight(m_pAndroidApp->window),
format);
```

To create an OpenGL ES rendering context, it is necessary to supply draw and read surfaces. Given that it is an OpenGL ES context that EGL is being asked to create, this means:

- Draw surface corresponds to the surface that all draw calls operating on the default frame buffer will rasterize to
- Read surface corresponds to the surface that all read calls operating on the default frame buffer will take their data from

In this case, a window surface is being created since the application must render to an Android window. In EGL, window surfaces are always double-buffered.

To create a surface, provide the EGLDisplay handle, as well as the EGLConfig instance, which tells EGL the requirements for the object to be created. It is possible to pass an attribute list to further customize the surface behavior, but in this case it is not necessary.

```
EGLSurface surface = eglCreateWindowSurface(
display,     config,
    m_pAndroidApp->window,
NULL); /* attrib_list */
```

All necessary EGL objects are now available to create an OpenGL ES context. Call the eglCreateContext function passing the EGLDisplay and EGLConfig created earlier. Since the context does not need to share its name space with any other rendering context, set the third argument to NULL.

NOTE: In the attribute list passed to eglCreateContext, the EGL_CONTEXT_CLIENT_VERSION property is set to 2. The interpretation of this is that EGL creates a context for the highest OpenGL ES version that is backwards compatible with the requested version and is supported by the driver.

The eglCreateContext call returns an OpenGL ES context instance but does not bind it to the current thread; that is done by the eglMakeCurrent call in the next line. As shown in the following code snippet, the call uses the EGLSurface instance created earlier as both draw and read surfaces. After the eglMakeCurrent call completes successfully, the application can start issuing OpenGL ES calls from the thread that called eglMakeCurrent.

```
EGLint contextAttribs[] =
{
    EGL_CONTEXT_CLIENT_VERSION, 2,
    EGL_NONE  /* terminator */
};
EGLContext context = eglCreateContext(
display,     config,     NULL,
/* share_context */     contextAttribs);
   if (eglMakeCurrent(display,
surface, /* draw */
surface, /* read */
context) == EGL_FALSE) {     return
FALSE; }
```

2.3.2 Adreno GPU detection

If the application needs to check whether an Adreno GPU is present, it can:

1. Call glGetString(GL_RENDERER) within an active rendering context to retrieve a string containing platform-specific information about the active renderer.
2. Check if the retrieved string contains the substring Adreno.

The Adreno GPU version is also found in the GL_RENDERER string, following the “Adreno” keyword.

The following code from the PlatformDetect sample application shows how this can be done. It parses the GL_RENDERER string and uses a text output function to display the results. This code is found in the file scene.cpp, method CSample::DetectAdreno: const GLubyte* renderer = glGetString(GL_RENDERER);.

```
const char* pos = strstr((char *) renderer, "Adreno");
if (pos) {   ShowText("Adreno GPU detected\n");   pos
+= strlen("Adreno");   if (*pos == ' ') ++pos;   if
(!strncmp(pos, "(TM)", strlen("(TM)")))   {     pos +=
strlen("(TM)"); // Ignore TM marker   }
  if (*pos == ' ') ++pos;
ShowText("Adreno version: %s\n", pos); }
else {
  ShowText("Adreno GPU not detected\n");
}
```

2.3.3 Detect supported ES extensions

Depending on the running version of OpenGL ES, there are two ways of retrieving a list of extensions supported by the driver.

- Under OpenGL ES 2.0 (and later), retrieve the list of extensions by calling:

```
const GLubyte* extensions = glGetString(GL_EXTENSIONS);
```

Bind the thread to a rendering context before making this call. It returns a NULL-terminated string containing the list of extensions supported by the active OpenGL ES implementation. The extensions are delimited by a single space character. The number of extensions present is also available through the OpenGL ES constant value GL_NUM_EXTENSIONS, which can be queried with a glGetIntegerv call.

NOTE: Be aware that the extensions string can be large. The application must never assert that the string be a certain maximum size, or limit the number of extensions to a certain number.

The Platform Detect sample application in the Adreno SDK was written for OpenGL ES 2.0, and it uses the above method to list all the extensions supported. The code to retrieve and parse the GL_EXTENSIONS string, from the file scene.cpp, method CSample::ListExtensions is as follows.

```
const char* extensions = (const char *)
glGetString(GL_EXTENSIONS); for (int posStart = 0, posCurrent = 0;
true; ++posCurrent) {   char c = extensions[posCurrent];   if (c
== ' ' || c == 0)   {     if (posCurrent > posStart)     {
ShowText("Extension: %.*s\n", posCurrent - posStart,
extensions + posStart);     }     if (c == 0)     {       break;
// reached the terminating EOS character
    }
  posStart = posCurrent + 1; // next extension will start
                         // after the space character
  }
}
```

- Things are simpler when using OpenGL ES 3.0 or later, where a new function called glGetStringi is introduced. This function allows requests for each individual extension name by index number, meaning that it is no longer necessary to write string-parsing code.

In this case, the snippet above could be replaced by the following code:

```
glGetIntegerv(GL_NUM_EXTENSIONS, &n_extensions);
for (int n_extension = 0;
n_extension < n_extensions;
++n_extension) {   const GLubyte*
extension;
  extension = glGetStringi(GL_EXTENSIONS, n_extension);
  ShowText("Extension: %s\n", extension);
}
```

2.3.4 Implementation of Blinn-Phong lighting

The simulation of light transfer is a broad subject that cannot be covered in depth in this developer guide; instead, it focuses on the use case implemented by the Lighting sample in the Adreno SDK. The sample makes the following assumptions:

- The Blinn-Phong lighting model should be used.
- All meshes should be lit by light from a single direction. In other words, the rendering model must assume that the light source is located far away, so that all light rays are traveling in the same direction.

It is encouraged to become familiar with the subject matter to get a better understanding of a variety of lighting models used in modern graphical applications.

2.3.4.1 Theoretical introduction

The standard lighting equation is one way to compute the contribution made by scene lighting to the meshes rendered by GLSL applications. Back in OpenGL ES 1.1, it was exposed to applications as part of the fixed-function rendering pipeline. From OpenGL ES 2.0 onward, the fixed-function rendering pipeline is no longer supported, but the same technique can still easily be mapped to OpenGL ES shaders.

The standard lighting equation is as follows:

$$result = (diffuse\ contribution) + (specular\ contribution) + (ambient\ contribution) \tag{2-1}$$

The equation defines a local lighting model—it focuses only on the fragment being processed and ignores the existence of any other geometry in the scene. This means that it will not produce any shadows, reflections, or refractions. These effects must be simulated by separate techniques that are not covered in this developer guide.

In the equation, diffuse and specular factors shade the mesh, taking into account the properties of the point light (location and color) and the details of the shaded point (normal vector). The ambient contribution is a special component to account for indirect light effects.

The geometry is also assigned a material. The material is a set of multipliers that are used in the computations to give the geometry a distinctive look in the scene.

Diffuse contribution

The diffuse component is one of the two factors used in the equation that model direct light, i.e., light that strikes the object directly. It represents the amount of the incoming light that scatters off the diffuse surface of the mesh and reaches the eye. The contribution is not affected by the location of the viewer, since the reflected rays are scattered randomly and it assumes a statistically uniform distribution. However, the position of the light source relative to the surface is important, because a surface that is perpendicular to the rays receives more light than a surface oriented at a steeper angle.

The diffuse component therefore follows Lambert’s law, which says that the intensity of the reflected light is proportional to the cosine of the angle between the rays of light and the normal surface. For this reason, the diffuse component is sometimes referred to as Lambertian lighting.

This component can be calculated using the diffuse contribution equation.

$$(diffuse\ contribution) = \left(dot(n,l) * diffuse_{light}\right) * diffuse_{material} \tag{2-2}$$

Where:

- n is the unit normal vector for the point being shaded.
- l is the unit light vector that points toward the light source from the shaded point.
- $diffuse_{material}$ is the diffuse color of the material.
- $diffuse_{light}$ is the diffuse color of the light.
- $dot(x, y)$ is a dot vector operation applied against the vectors x and y.

It is important to remember to clamp the result of the dot vector operation to zero to prevent the point from being lit from behind. In order to make sure that the dot operation returns a cosine of the angle between vectors *n* and *l*, the two vectors must be of unit length.

Specular contribution

The specular contribution is the second factor used in the equation that models direct light. It represents the amount of incoming light that is reflected from the surface of the mesh and reaches the eye.

Unlike the diffuse contribution, the intensity of this factor is highly dependent on the camera location. It is the specular contribution that gives the shiny appearance to rendered geometry.

The Blinn-Phong model description of the specular contribution is as follows:

$$(specular\ contribution) = dot(n,h)\ {}_{glossiness_{material}} * specular_{light}) * specular_{material} \tag{2-3}$$

Where:

- n is the unit normal vector for the point being shaded.
- v is the view vector, i.e., a vector that points toward the eye from the shaded point.
- h is a special halfway vector between v and a light vector (as defined for the diffuse component) l, defined by the halfway vector calculation:
  $$h = (v + l) / length(v + l) \tag{2-4}$$
- $glossiness_{material}$ defines the glossiness of the material. Smaller values give a broader and more gentle fall off from the hotspot. Large values give a sharp fall off.
- $specular_{light}$ is the specular color of the light.
- $specular_{material}$ is the specular color of the material.
- dot(x, y) is a dot vector operation applied against the vectors x and y.

As with the diffuse contribution, it is important to clamp the dot vector operation result to prevent it from going negative.

Ambient contribution

In the real world, the light rays emitted by light sources usually bounce off the walls many times before they reach the viewer. This is referred to as indirect light. The standard lighting model does not track rays, so the contribution of indirect lighting must be faked. The simplest way to achieve this is by adding a constant value to each fragment to make up the missing energy contribution. This is exactly how the ambient contribution works.

Mathematically, the ambient contribution can be described as follows:

$$(ambient\ contribution) = ambient * ambient_{material} \tag{2-5}$$

Where:

- *ambient* is the global ambient light value for the scene.
- $ambient_{material}$ is the ambient color of the material.

2.3.4.2 GLSL implementation in Lighting demo

The following sections describe how the lighting theory can be put to use in GLSL shader code.

The Lighting sample application can run in two modes. Each mode uses a slightly different pair of fragment and vertex shaders.

- Per-Vertex mode, where the lighting calculations are performed in the vertex shader – The result values are then interpolated by hardware during rasterization and then saved directly to the render target in the fragment shader stage.
- Per-Pixel mode moves the actual lighting calculations to the fragment shader – The vertex shader stage is still used for some of the vector computations.

Per-Vertex mode takes less time to execute because the lighting calculations are performed on a per-vertex basis, i.e., less often. However, the visual quality of this approach is significantly lower when compared to the per-pixel approach. The difference is more obvious with lower levels of geometry tessellation. The shiny reflections introduced by the specular contribution take the biggest hit in this mode—the highlights easily blend between vertices owing to the highly nonlinear nature of their behavior.

The following sections describe how the GLSL shaders work for each of these modes.

Per-vertex rendering mode shaders

Both the fragment and the vertex shaders are stored in the SDK\Development\Assets\Samples\Shaders\PerVertexLighting.glsl file.

The vertex shader starts by declaring uniforms, attributes, and varyings.

```
struct MATERIAL {
vec4  vAmbient;
vec4  vDiffuse;
vec4  vSpecular; };
uniform   mat4     g_matModelView;
uniform   mat4     g_matModelViewProj;
uniform   mat3     g_matNormal; uniform
vec3     g_vLightPos; uniform   MATERIAL
g_Material;
attribute vec4 g_vPositionOS;
attribute vec3 g_vNormalOS;
```

```
varying   vec4     g_vColor;
```

The meaning of each of these fields is as follows:

- g_matModelView – Model-view matrix; transfers a single vertex defined in object space and positions it relative to the viewer
- g_matModelViewProj – Model-view-projection matrix; transfers a single vertex defined in object space to the clip space
- g_matNormal – Normal matrix; used to transfer the normal vector to world space
- g_vLightPos – Light position in world space
- g_Material – Stores material properties for the rendered mesh

The shader defines two input attributes:

- g_vPositionOS – Input vertex data; the vertices are defined in object space
- g_vNormalOS – Input normal data; the normals are defined in object space

Finally, the vertex shader passes a single four-component vector to the fragment shader:

- g_vColor – Shaded color value for the processed vertex; the final fragment value is defined by a weighted average of three such values, and the final outcome is directly correlated with the sample location within the triangle built of the three vertices

After the inputs and outputs are defined, continue with the main entry-point implementation.

```
void main() {  vec4 vPositionES = g_matModelView     *
g_vPositionOS;  vec4 vPositionCS = g_matModelViewProj *
g_vPositionOS;
```

Here, the input vertex position transforms into two spaces:

- vPositionES – Eye space (also known as world space)
- vPositionCS – Clip space

```
    // Output clip-space position
   gl_Position = vPositionCS;
```

A vertex shader behavior is undefined if gl_Position is not set to any value. In this step, ensure the variable is set to the clip-space vertex position.

```
 // Transform object-space normals to eye-space  vec3
vNormal = normalize(g_matNormal * g_vNormalOS);
```

The sample carries out the lighting calculations in world space. This is necessary for the specular component calculations to work correctly. After transforming to world space, normalize the vector, because the subsequent calculations require the vector to be of unit length.

```
 // Compute intermediate vectors  vec3 vLight
```

```
= normalize(g_vLightPos);  vec3 vView   =
normalize(vec3(0.0,0.0,0.0)
vPositionES.xyz);  vec3 vHalf   =
normalize(vLight + vView );
```

Here, calculate a few vectors that are necessary for the actual light calculations:

- vLight – The light vector; note that the demo allows the user to move the light during runtime, so for simplicity and tutorial consistency, use the light position here to calculate the actual light vector.
- vView – The view vector; assume the camera to be located at the origin of the world space.
- vHalf – The half vector; calculated as described in Section 2.3.4.1 in the equation for halfway vector calculation.

```
 // Compute the lighting in eye-space  float fDiffuse
=     max(0.0, dot(vNormal, vLight));  float fSpecular
= pow(max(0.0, dot(vNormal, vHalf) ),
g_Material.vSpecular.a);
```

This part calculates the diffuse and specular contributions. The computations map straightforwardly to the notes above. Note that:

- Max operations make sure the result values never go below zero.
- The alpha channel of g_Material.vSpecular stores the glossiness factor.

```
 // Combine lighting with the material properties
g_vColor.rgba  = g_Material.vAmbient.rgba;
g_vColor.rgba += g_Material.vDiffuse.rgba * fDiffuse;
g_vColor.rgb  += g_Material.vSpecular.rgb * fSpecular;
}
```

In the final part of the vertex shader, take the computed standard light equation contributions and multiply them by material-specific values. Then, sum up all values, which gives the final shaded color for the vertex. Values for each sample are interpolated between vertices.

Given that it is describing the per-vertex shading, the fragment shader is as follows:

```
varying vec4 g_vColor;
void main() {
gl_FragColor = g_vColor; }
```

The shader takes the interpolated color value and stores it in the render target at index zero. Since the demo is written for OpenGL ES 2.0, the gl_FragColor variable is used here, which always maps to draw buffer at index zero.

Per-pixel rendering mode shaders

The shaders used in per-fragment rendering mode work as follows.

Start with the vertex shader, which is expected to be rather lightweight; it should only output vectors that can be linearly interpolated across the primitive surface. It should not perform any lighting-specific computations, since those will be executed on a per-fragment level.

```
uniform   mat4 g_matModelView;
uniform   mat4 g_matModelViewProj;
uniform   mat3 g_matNormal;
attribute vec4 g_vPositionOS;
attribute vec3 g_vNormalOS;
varying   vec3 g_vNormalES;
varying   vec3 g_vViewVecES;
void main() {  vec4 vPositionES = g_matModelView     *
g_vPositionOS;  vec4 vPositionCS = g_matModelViewProj *
g_vPositionOS;
 // Transform object-space normals to eye-space
vec3 vNormalES = g_matNormal * g_vNormalOS;
 // Pass everything off to the fragment shader
gl_Position  = vPositionCS;  g_vNormalES  =
vNormalES.xyz;
 g_vViewVecES = vec3(0.0,0.0,0.0) - vPositionES.xyz;
}
```

The shader outputs the normal vector and the view vector, and also sets the gl_Position to the vertex position after transforming it to clip space.

The fragment shader for the per-pixel rendering mode appears as follows.

```
struct MATERIAL {
vec4  vAmbient;
vec4  vDiffuse;
vec4  vSpecular; };
uniform MATERIAL g_Material;
uniform vec3     g_vLightPos;
varying vec3     g_vViewVecES;
varying vec3     g_vNormalES;
void main() {  // Normalize per-pixel
vectors  vec3 vNormal =
normalize(g_vNormalES);  vec3 vLight  =
normalize(g_vLightPos);  vec3 vView   =
normalize(g_vViewVecES);  vec3 vHalf   =
normalize(vLight + vView);
 // Compute the lighting in eye-space  float fDiffuse
=     max(0.0, dot(vNormal, vLight));  float fSpecular
= pow(max(0.0, dot(vNormal, vHalf) ),
g_Material.vSpecular.a);
 // Combine lighting with the material properties
gl_FragColor.rgba  = g_Material.vAmbient.rgba;
gl_FragColor.rgba += g_Material.vDiffuse.rgba * fDiffuse;
gl_FragColor.rgb  += g_Material.vSpecular.rgb * fSpecular;
}
```

This should look familiar. The fragment shader starts by normalizing the interpolated vectors, which is important, because even though they were initially at unit length, the interpolation process may have changed their magnitude. Given that the dot vectors are used to obtain the cosine of the angle between the two vectors, ensure that the vectors are always normalized.

The remaining calculations appear as if they were taken straight from the vertex shader from the per-vertex rendering mode implementation. The only difference is that they are now executed on a per-fragment basis, instead of per-vertex. This directly translates to an improved visual experience, at the cost of significantly higher hardware utilization.

2.3.5 Retrieving ES constant values

Once a rendering context is bound to the thread and makes it active, the functions listed in Table 2-2 retrieve OpenGL ES constant values.

Table 2-2 Getter functions in OpenGL ES

Getter function	Available in ES 2.0?	Available in ES 3.0?	Available in ES 3.1?
glGetBooleanv	✓	✓	✓
glGetBooleani_v	✗	✗	✓
glGetFloatv	✓	✓	✓
glGetInteger64i_v	✗	✓	✓
glGetInteger64v	✗	✓	✓
glGetIntegeri_v	✗	✓	✓
glGetIntegerv	✓	✓	✓

The getter function name consists of the glGet prefix, a part that indicates the result format, and a suffix v or i_v. The suffix i_v indicates that the getter works for indexed states.

NOTE: Using an indexed getter for a nonindexed state is not allowed and results in an error.

Every item of the OpenGL ES state has a base format in which the corresponding value is stored. However, using a getter for a different format is permitted. In the case of a format mismatch, the base value is converted to the format corresponding to the getter used.

The PlatformDetect sample application from the Adreno SDK demonstrates how to retrieve the constant maximums and ranges defined in the OpenGL ES 2.0 core specification. The following code may be found in the file scene.cpp, method CSample::ListMaxValues:

```
GLfloat aliasedLineWidthRange[2] = {0.0f, 0.0f};
glGetFloatv(GL_ALIASED_LINE_WIDTH_RANGE, aliasedLineWidthRange);
ShowText("GL_ALIASED_LINE_WIDTH_RANGE = %f, %f\n",
aliasedLineWidthRange[0], aliasedLineWidthRange[1]);
GLfloat aliasedPointSizeRange[2] = {0.0f, 0.0f};
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, aliasedPointSizeRange);
ShowText("GL_ALIASED_POINT_SIZE_RANGE = %f, %f\n",
aliasedPointSizeRange[0], aliasedPointSizeRange[1]);
GLint maxCombinedTextureImageUnits = 0;
glGetIntegerv(GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
&maxCombinedTextureImageUnits);
ShowText("GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS = %d\n",
maxCombinedTextureImageUnits);
GLint maxCubeMapTextureSize = 0;
glGetIntegerv(GL_MAX_CUBE_MAP_TEXTURE_SIZE, &maxCubeMapTextureSize);
ShowText("GL_MAX_CUBE_MAP_TEXTURE_SIZE = %d\n",
maxCubeMapTextureSize);
GLint maxfragmentUniformVectors = 0;
glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_VECTORS,
&maxfragmentUniformVectors);
ShowText("GL_MAX_FRAGMENT_UNIFORM_VECTORS = %d\n",
maxfragmentUniformVectors); GLint
maxRenderbufferSize = 0; glGetIntegerv(GL_MAX_RENDERBUFFER_SIZE,
&maxRenderbufferSize); ShowText("GL_MAX_RENDERBUFFER_SIZE = %d\n",
maxRenderbufferSize);
GLint maxTextureImageUnits = 0;
glGetIntegerv(GL_MAX_TEXTURE_IMAGE_UNITS, &maxTextureImageUnits);
ShowText("GL_MAX_TEXTURE_IMAGE_UNITS = %d\n",
maxTextureImageUnits);
GLint maxTextureSize = 0;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTextureSize);
ShowText("GL_MAX_TEXTURE_SIZE = %d\n",
maxTextureSize);
GLint maxVaryingVectors = 0;
glGetIntegerv(GL_MAX_VARYING_VECTORS, &maxVaryingVectors);
ShowText("GL_MAX_VARYING_VECTORS = %d\n",
maxVaryingVectors);
GLint maxVertexAttribs = 0;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxVertexAttribs);
ShowText("GL_MAX_VERTEX_ATTRIBS = %d\n", maxVertexAttribs);
GLint maxVertexTextureImageUnits = 0;
glGetIntegerv(GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
&maxVertexTextureImageUnits); ShowText("GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS =
%d\n",
maxVertexTextureImageUnits);
```

```
GLint maxVertexUniformVectors = 0;
glGetIntegerv(GL_MAX_VERTEX_UNIFORM_VECTORS,
&maxVertexUniformVectors); ShowText("GL_MAX_VERTEX_UNIFORM_VECTORS = %d\n",
maxVertexUniformVectors);
GLint maxViewportDims[2] = {0, 0};
glGetIntegerv(GL_MAX_VIEWPORT_DIMS, maxViewportDims);
ShowText("GL_MAX_VIEWPORT_DIMS = %d, %d\n",
maxViewportDims[0], maxViewportDims[1]);
```

For more detail about the glGet* methods, see:

- The OpenGL ES Specification at http://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf.
- The OpenGL ES Reference Page for glGet at https://www.khronos.org/opengles/sdk/docs/man3/html/glGet.xhtml.

There is also more information about the following:

- Conversion rules
- Details of the states that are available to be queried
- States that are considered nonindexed and those that are considered indexed

2.4 About the OpenGL ES implementation

Table 2-3 lists the values of all the GL constant values as supported by Adreno in OpenGL ES 2.0 contexts.

Table 2-3 Adreno GL constant values for OpenGL ES 2.0 contexts

GL constant name	Adreno value
GL_ALIASED_LINE_WIDTH_RANGE	1.0 to 8.0
GL_ALIASED_POINT_SIZE_RANGE	1.0 to 1023.0
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS	32
GL_MAX_CUBE_MAP_TEXTURE_SIZE	16384
GL_MAX_FRAGMENT_UNIFORM_VECTORS	224
GL_MAX_RENDERBUFFER_SIZE	16384
GL_MAX_TEXTURE_IMAGE_UNITS	16
GL_MAX_TEXTURE_SIZE	16384
GL_MAX_VARYING_VECTORS	32
GL_MAX_VERTEX_ATTRIBS	32
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS	16
GL_MAX_VERTEX_UNIFORM_VECTORS	256
GL_MAX_VIEWPORT_DIMS	16384

2.5 Debug and profile

Debugging an OpenGL ES application is usually much more time-consuming than debugging a typical nongraphics application. Even if OpenGL ES is asked to perform an invalid operation, it is rare that it results in a crash. The most common crash case is when asking to download client-side data from an incorrect memory location. There is no fatal exception either.

The lack of a crash or fatal exception means that there is no good starting point for an investigation. Typically, information about the OpenGL ES state configuration must be gathered, but this is not easy to do, unless using a specialized tool like the Adreno Profiler. Otherwise, manually add glGet* calls to the code to find out how OpenGL ES is configured at a particular point. However, this can be a time-consuming activity.

A problem relating to OpenGL ES usually manifests itself in one of two ways:

- Parts of the geometry may not render at at all, or may be drawn with visual glitches. Problems like these are usually caused by driver bugs, by a misunderstanding by the developer about the way context sharing or certain other OpenGL ES features work, by thread race conditions in the application, or because the application assumes that it can use more resources of a certain type, e.g., texture units, than is supported by the OpenGL ES implementation.
- The application works correctly on the development platform but malfunctions on other platforms. In addition to the causes listed above, a common cause for this problem is shader bugs. Typical errors include missing #extension declarations, omitted precision definitions, packed/shared member layout incompatibilities between platforms, or shaders attempting to use more active samplers or uniforms than is supported by the OpenGL ES implementation.

There is usually no straightforward way of diagnosing the shader issues mentioned in the second point above. While some of the shader language problems could be determined by running vendor-specific offline compilers against the shaders, this is often an arduous and impractical task, given that many OpenGL ES applications generate shaders on-the-fly. Also, these compilers rarely expose program linking functionality, which is one of the areas where many shader-specific incompatibility problems arise. As such, these issues can only be detected by manually testing an application on a number of different platforms.

Fortunately, many of the problems listed in the first bullet point can be detected at development time.

The following sections describe the three different techniques that developers can use to detect errors such as these, in the process of implementing the OpenGL ES application. They also cover some tools that can improve software performance.

2.5.1 Debug an OpenGL ES application

2.5.1.1 Using glGetError

The OpenGL ES API is a strict C API, meaning that any errors it detects are be reported via exceptions. The basic method for an OpenGL ES application to detect errors is the use of a glGetError call. This call can be issued in any thread to which a rendering context has been bound.

There is a range of different error codes that can be returned if an invalid call is made. The normal rule is that if OpenGL ES detects an error, then the offending command does not cause any modification of the OpenGL ES state, nor does it rasterize any samples. The error code GL_OUT_OF_MEMORY is a notable exception here. This error is reported when a memory allocation request fails on the driver side. If an application detects this error, it should assume that the OpenGL ES state has become undefined. The safest step it could take would be to terminate the process.

All error codes that OpenGL ES can report are:

- GL_INVALID_ENUM – One of the GLenum arguments passed to a function was invalid. A common cause for this is that the developer passes an enum value defined as part of an OpenGL ES extension, when that extension is not actually available. Another reason is that the value is simply incorrect.
- GL_INVALID_FRAMEBUFFER_OPERATION – An API function that requires the draw and/or read frame buffers to be complete was called, but this was not the case. Note that this is not required by all API functions. This error code can also be reported if the OpenGL ES implementation does not support rendering to a particular frame buffer configuration, in which case the application should use a different set of internal formats for the frame buffer concerned.
- GL_INVALID_OPERATION – The operation attempted to perform was invalid, given the current OpenGL ES state configuration. This error code is vague and the offending command must be narrowed down to find the cause of the error. For large applications especially, this may not be a trivial task.
- GL_INVALID_VALUE – A numeric argument passed to a function was invalid. This usually occurs for reasons similar to the ones described for GL_INVALID_ENUM.
- GL_OUT_OF_MEMORY – The driver ran out of memory while trying to execute the command. The application should terminate as soon as possible, as the working environment can no longer be assumed to be stable.

A generalized description of the error is given above, but a more specific interpretation is dependent upon the command causing the error.

NOTE: glGetError does not tell the application which command has reported the error, and that errors can be reported by OpenGL ES commands called implicitly by the driver; e.g., when the driver has previously decided to defer a call such as a draw call, and is now issuing it internally in the course of processing another command.

Internally, when the driver detects an error in an API call, it raises a flag corresponding to one of the codes. If further errors are detected, they do not affect the recorded error state. When glGetError is called, the cached error code is returned and the flag is lowered so that subsequent glGetError calls return GL_NO_ERROR.

Since the Adreno driver caches API calls in a command buffer, glGetError calls can be considered expensive, because they flush the pipeline and wait for the completion of all buffered commands. Unless performance is unimportant for the application, it is not recommended to insert glGetError calls after every API call made, at least not in release builds. A good compromise is to insert guard glGetError calls at strategic locations, for debug builds only. Using this approach, performance is unaffected in production builds, and in the case that a bug report arrives, it will not take long for a team to prepare a debug version that can be used to locate the rendering pipeline part causing the problem.

Using GL_KHR_debug

There are some significant limitations with the use of glGetError.

- The error codes do not convey precise information about the type of error.
- To fix a problem, the programmer must first work out which OpenGL ES API call has caused the error to be raised, and must then look at the bigger picture, e.g., OpenGL ES state, to understand why it is that the error code is being generated. This takes up development time that could instead have been spent on implementing new features.
- There is no way of getting the driver to make a callback to the application so that the programmer could insert a breakpoint to find out what went wrong and where.

The Adreno driver supports a special extension called GL_KHR_debug, which aims to address these needs and includes a number of features to enhance the debugging experience for OpenGL ES developers.

The following section focuses on the features that are most relevant to debugging OpenGL ES applications and will provide an overview of the API. For further details, see https://www.opengl.org/registry/specs/KHR/debug.txt.

When the GL_KHR_debug extension is supported, applications can sign up for driver callbacks by calling glDebugMessageCallbackKHR. This function allows the registering of a callback function pointer and an optional user-specified argument. OpenGL ES uses that callback to provide feedback to the application in the form of human-readable, NULL-terminated strings. The feedback can contain detailed information about why an error code was generated, as well as information of the following types:

- Warnings about the use of deprecated functionality, or of something that is marked as undefined in the specification
- Implementation-dependent performance warnings
- Warnings about using extensions or shaders in a vendor-specific way
- User-injected messages
- Debug group stack notifications (covered below)

Each feedback message is accompanied by a set of enums that provide information about the:

- Origin of the message (driver, shader compiler, window system, etc.)
- ID of the message

- Type of the message (error report, performance warning, portability hint, etc.)
- Severity level of the message (high, medium, low, notification)

The callbacks will only occur if GL_DEBUG_OUTPUT_STATE_KHR is enabled by calling glEnable. To disable the functionality again, call glDisable for the same state enum.

The application can inject its own messages into the debug stream by calling glDebugMessageInsertKHR. This is especially useful for middleware which can use the mechanism to provide hints to the developer or notification of any error situations detected.

All messages either generated by the driver or inserted into the command stream using glDebugMessageInsertKHR are written to the active debug group, which is the top of the debug group stack. A new debug group (identified by a user-specified message) can be pushed onto the stack by calling gl-PushDebugGroupKHR. Existing debug groups can be popped off the stack by calling glPopDebugGroupKHR. Whenever a debug group is pushed onto or popped off of the debug group stack, the message that has been associated with the group will be inserted into the stream. An example use case would be to use debug groups to mark the start and end of each rendering pass.

The application can filter out unwanted messages by calling glDebugMessageControlKHR. Any of the properties of the message can be used as a filtering key. This is referred to as volume control. The volume control setting applies to the active debug group, and will be inherited if a new debug group is pushed onto the stack.

If the application does not register a callback function but does enable GL_DEBUG_OUTPUT_STATE_KHR, then the messages will be stored in a message log. The log can hold up to GL_MAX_DEBUG_LOGGED_MESSAGES messages. Once the storage fills up, any subsequently generated messages are discarded until such time as the application frees up some space by fetching one or more messages. The messages can be fetched by calling glGetDebugMessageLogKHR, which returns both the message string and the associated properties of each message.

Another useful feature offered by the GL_KHR_debug extension is the ability to provide the callback information in two different modes:

- Asynchronous mode (active by default) – The OpenGL ES implementation can call the debug callback routine concurrently from multiple threads, including threads that the context that generated the message is not currently bound to (examples include but are not limited to, threads to which other contexts are bound, or threads that are internally used by the driver). It can also issue the callback asynchronously after the OpenGL ES command that generated the message has already returned. While Asynchronous mode is active, it is the responsibility of the application to ensure thread safety.
- Synchronous mode – The driver is not allowed to issue more than one callback at a time, per rendering context. The callback will be made before the OpenGL ES command that generated the debug message is allowed to return. Synchronous mode causes all calls to be implicitly flushing, so performance is greatly reduced. However, given the fact that the callback occurs at the time of the OpenGL ES API call, this mode greatly simplifies the debugging process for developers.

  Synchronous mode can be explicitly enabled by calling glEnable for GL_DEBUG_OUTPUT_SYNCHRONOUS_KHR mode. For the application to go back into Asynchronous debugging mode, glDisable can be called for the same enum.

The GL_KHR_debug extension also allows the developer to label OpenGL ES objects with NULL-terminated strings using the functions glObjectLabelKHR and glObjectPtrLabelKHR. The label can later be retrieved using glGetObjectLabelKHR or glGetObjectPtrLabelKHR. This could help easily identify OpenGL ES objects during the debugging process, without the need to traverse the application's internal data model.

Shader debugging

Situations may arise where a rendering glitch is caused not by incorrect OpenGL ES API usage, but rather by a bug hidden somewhere in one of the shaders that make up the program used by the draw calls. Here are a few tips on how to approach such situations.

- Start by making sure that all the input attributes defined for the vertex shader are passed correct values. For instance, if the lighting is not working as expected, start by verifying that the normal data that is being used for the calculations is valid, e.g., check this by passing unmodified normal data to the fragment shader to verify by visual inspection that each vertex is assigned correct vector values.

NOTE: Be careful when modifying shaders. E.g., if the developer commented out the existing implementation and replaced it with code to pass the normal vector to the fragment shader, then it could cause many of the existing input attributes and uniforms to become inactive. Depending on how the application and shaders are written, this could make the bug even more difficult to track down. Instead of removing the whole body, arrange for all other variables that might contribute to the result value to be multiplied by a very small value, e.g., 1.0/256.0 for an internal format that uses 8 bits per component, so that they do not hide the result value that is being visually inspected.

- Do the same for all the uniforms used by the shader. Pay special attention to uniform block members. Make sure the shaders define the same uniform block layout that the application is assuming.
- Use transform feedback to transfer the data out of the rendering pipeline back to the process, if it is suspected that some of the calculations may be executing incorrectly. Transform feedback allows for checking that the data is passed correctly through the whole rendering pipeline, except for the fragment shader stage. This becomes especially important when starting to use geometry and/or tessellation control/evaluation shaders.
- If the platform supports geometry shaders, use them to emit helper geometry, e.g., normal vectors.
- If some textures are not being sampled correctly, it is likely that they are considered incomplete. The easiest way to check this is by selecting GL_NEAREST minification filtering, and setting GL_TEXTURE_BASE_LEVEL and GL_TEXTURE_MAX_LEVEL to the index of a mipmap that has been uploaded.
- If textures are still inaccessible, ensure no sampler object is overriding the texture parameters of a texture unit to which that texture has been bound. Verify that the sampler uniforms are set to use the corresponding texture units.
- Use the Adreno Profiler tool to edit the shaders and investigate the results in real time.

2.5.2 Profile an OpenGL ES application

If the application is underperforming, the first step is to identify the rendering pipeline parts that are taking too long. Unfortunately, given that OpenGL ES works asynchronously, it is not enough just to record the time before and after each rendering pass and then check the time difference. The values given would usually only show how long it took the driver to store the request in the command buffer.

Here are a few approaches to measure the GPU time taken by a specific part of the rendering pipeline:

- Starting with OpenGL ES 2.0, if wanting to measure time taken for a given set of API calls, delimit that region with glFinish calls. Record the start time just after the first glFinish call, and the end time just after the second glFinish call. The time difference will tell how much time it took to execute that block. This method is less accurate than the one below because of the longer round trip that needs to be made before the execution flow is returned to the caller. It also causes more degradation to overall rendering performance because the first glFinish call must wait until all previously enqueued commands finish executing on the GPU.
- If using OpenGL ES 3.0, a more efficient way is to use sync objects as a lightweight alternative to the heavyweight glFinish calls. To do so, follow the same pattern used in the first solution, but replace each glFinish call with the following code:

```
GLsync sync;
sync = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
glClientWaitSync(sync);
```

NOTE: Do not forget to release the sync objects using glDeleteSync when finished with the measurements.

- It is possible to use the Scrubber and Grapher modes of the Adreno Profiler to collect more detailed information about rendering performance. For more detail on the topic, see Section 8.1.

3 Using OpenGL ES 3.0 with Adreno

3.1 New features in OpenGL ES 3.0

The arrival of OpenGL ES 3.0 in August 2012 substantially expanded the OpenGL ES feature set available to embedded application developers. Up to that time, many of the more complex features had only been available to desktop developers using OpenGL 3.x.

To take a few examples:

- Sampler objects and vertex attribute array divisors became a core feature in desktop OpenGL 3.3
- Seamless cube map filtering and fence sync objects became core in desktop OpenGL 3.2
- Instanced draw calls and uniform buffer objects were introduced in desktop OpenGL 3.1
- Frame buffer objects (with multiple render target support) and transform feedback were introduced in desktop OpenGL 3.0

All of the above are included in core OpenGL ES 3.0. It is true that partial support for some of these features was available in OpenGL ES 2.0 via the extensions mechanism. But their availability could not have been assumed across the whole OpenGL ES 2.0 ecosystem. If a developer wanted to use a feature that was not part of the ES2.0 core specification, they would have needed to implement a fall-back code-path to cater for devices not supporting the extension. This greatly increased the complexity of the implementation and the amount of testing needed. As a result, developers were usually discouraged from experimenting with the new features.

This section presents a conceptual view of many features introduced in OpenGL ES 3.0. Section 3.2 covers a subset of these features at the API level.

For further information about all of the features, see the OpenGL ES 3.0 specification at https://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

3.1.1 Two-dimensional array textures

Two-dimensional array textures (2D array textures) builds upon the concept of the two-dimensional texture (2D texture).

In a 2D texture, a mipmap level consists of a single image. In a 2D array texture, a single mipmap level holds a number of images. Each image held within a single mipmap is called a layer. All layers at a given mipmap level have the same resolution.

Layer data is in the internal format requested for the 2D array texture object at the time it was created. The width and height of all the layers at a given mipmap level is also defined at creation time and cannot be changed during the lifetime of the texture object. The number of layers a two-dimensional array texture object holds for each mipmap level is known as the texture object depth. This also needs to be defined at creation time.

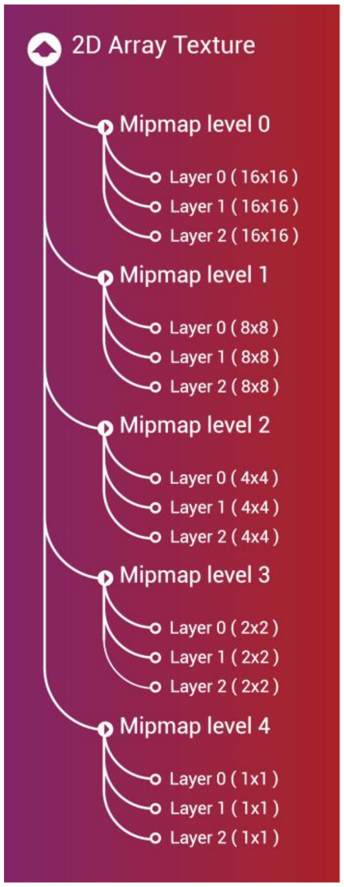

Figure 3-1 2D array texture

A mipmap chain can be allocated for a 2D array texture. As a minimum, it is possible to define just one base mipmap level for each layer, and texture sampling will still work. This can be done if mipmap-based texture filtering is not necessary, or if available memory is low.

All layers at the mipmap level n+1 must be half the size of the layers at mipmap level n, e.g., if using a layer size of 4x8 at mipmap level 0, each layer at mipmap level 1 would need to have a size of 2x4. Mipmap level 2 would be 1x2, and for the last mipmap level each layer would take a single pixel. Each dimension is clamped at 1.

Once mipmap storage is defined as either mutable or immutable, 2D array textures can be:

- Used as render targets; layers can be rendered using frame buffer objects
- Sampled from any shader stage using new GLSL texture sampling functions

Nearly all texture sampling functions available for 2D textures in ES Shading Language 3.0 can be used to sample 2D array textures. The only functions that do not support this are projective texture lookups and their derivatives. Should the requested location exceed the defined range in any axis, the behavior of the sampling functions is controlled by S/T/R wrap modes.

Texture sampling functions that operate on two-dimensional array texture targets take an additional parameter, which defines the layer index, from which the data should be sampled.

Important

Values returned by 2D array texture sampling functions always operate within the layer boundaries being sampled from. It is only the set of mipmaps defined for sampled layer that is used for calculating the result value. Hence, it is guaranteed that none of the taps used during the sampling process will take data from more than one layer. This is a fundamental concept of 2D array textures that distinguishes them from 3D textures.

2D array textures are often used to enhance existing texture atlas techniques. They are also useful for multitexturing or video frame storage purposes. Using 2D array textures, is it possible to reduce the number of texture bindings that are configured for each draw call, because shaders can be written that access multiple layers of a single 2D array texture via a single texture sampler.

3.1.2 Three-dimensional textures

From a conceptual point of view, three-dimensional textures (3D textures) are very similar to 2D array textures. The key differences between the two lie in how the data sampling process is performed and how the mipmap chain is built.

3D textures consist of a set of two-dimensional images called slices. All slices combined together form a single mipmap level. Subsequent mipmap levels must be half the size of preceding mipmap levels. A mipmap chain for a 3D texture object consists of a set of cuboids, where each subsequent cuboid is, in general, half the size of the one that precedes it.

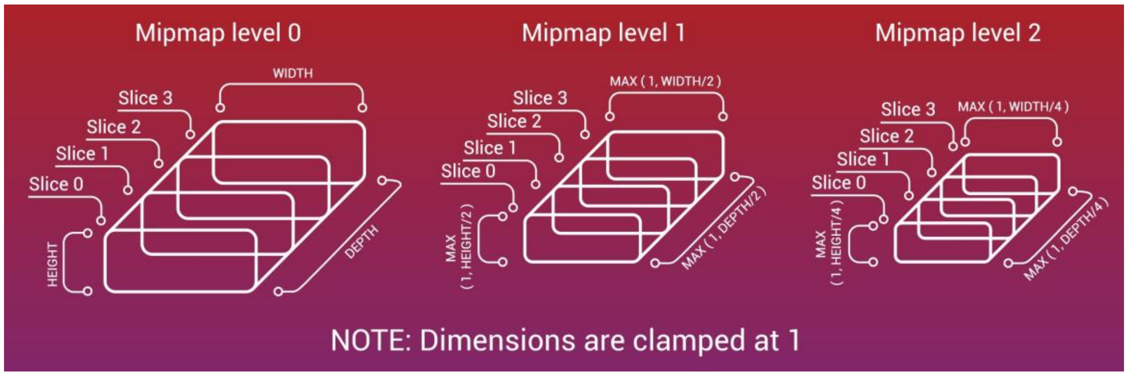

Figure 3-2 3D array texture

NOTE: This is different from 2D array textures, where each layer must have a separate mipmap chain allocated.

For each slice, the base mipmap level must be defined. Beyond this, declaration of subsequent mipmaps is optional, if nearest or linear minification filtering is used. Using mipmap-based minification filtering for a 3D texture object, for which the mipmap chain has not been defined, will render the texture incomplete, causing any sampling operations to return vec4(0, 0, 0, 1).

For 3D textures, it is assumed that all slices are uniformly distributed across a unit cube which starts at (0, 0, 0) and ends at (1, 1, 1), e.g., if a 3D texture object was defined with a depth of 4:

- Slice at index 0 would be at Z = 0
- Slice at index 1 would be at Z = 0.33
- Slice at index 2 would be at Z = 0.67
- Slice at index 3 would be at Z = 1

All 3D texture sampling functions in GLSL require the caller to provide a 3D location within the unit cube that the data should be sampled from. The result value is calculated from the slice data using a number of linear interpolation operations.

NOTE: This is a key difference between 2D array textures and 3D textures. When sampling from a 2D array texture, OpenGL ES never uses data outside the sampled layer.

The data returned by the 3D texture sampling functions when trying to sample locations defined outside the cube depends on the wrap mode configuration.

A common use case for 3D textures is the storage of volumetric data, since the functionality provides a hardware-accelerated means of calculating an interpolated value anywhere within the defined dataset.

3.1.3 Immutable textures

The only type of texture objects recognized by the core specification of OpenGL ES 2.0 were mutable texture objects. This meant that an OpenGL ES application was allowed, at any time during its execution, to completely redefine the texture mipmap configuration. Not only could it add or remove mipmaps on-the-fly, but it was also allowed to change the internal format or properties such as the width or height of any mipmap of any texture object. This freedom greatly reduced the optimization possibilities for driver implementations, which were forced to keep track of the completeness of all textures used by the application. Texture completeness verification is a significant overhead, especially if it has to be executed for every draw call the application makes.

The solution for OpenGL ES is in immutable texture objects. Initially introduced in the GL_EXT_texture_storage extension, they became a core feature of OpenGL ES 3.0. Immutable textures work just like mutable textures, except that it is no longer possible to apply any of the following API functions to texture objects that have been made immutable:

- glCompressedTexImage*
- glCopyTexImage*
- glTexImage*
- glTexStorage* – Can be used to initialize an immutable texture object

The new glTexStorage* entry-points make a texture immutable. They initialize a mipmap chain for a user-specified texture target, but do not fill the mipmaps with any contents. It is the responsibility of the application to fill the immutable texture object with actual contents by using glTexSubImage* entry-points.

Immutable textures do not have a specific use case. Instead, they should be considered a means of reducing the load on the driver, which often translates to a better rendering performance.

NOTE: On Adreno platforms, the use of immutable textures has a major performance advantage. Always use immutable textures and avoid all mutable textures.

3.1.4 Per-texture object LoD clamp

The OpenGL ES 3.0 API specification includes a technique for organizing textures at multiple resolutions that enables the display of low-resolution textures and slowly bringing in more detailed textures over multiple frames as the camera (position of the viewer) in the scene approaches the textured object. This kind of technique is typically used in mapping or navigation applications. When zooming in to the map, the larger resolution texture must be loaded, but streaming that texture takes time. Lower resolution textures are displayed in the meantime. This provides a better and more immediate viewing experience and also helps manage memory bandwidth more efficiently without compromising performance.

As an example, consider a texture that is 1024x1024 texels in size with 32 bits per texel. The Mip at LoD = 0 for this texture is 4MB and Mips 4-10 are about 5KB in size as seen in Figure 3-3. As a starting point for the LoD effect, download Mips for levels 4-10 by setting the base level to 4 and minimum LoD to 4. Once the application starts, download the Mip 3,2,1,0. Then by setting the base level to 0, slowly, over multiple frames, change min LoD to 0. Thus, the texture gradually phases to the highest resolution.

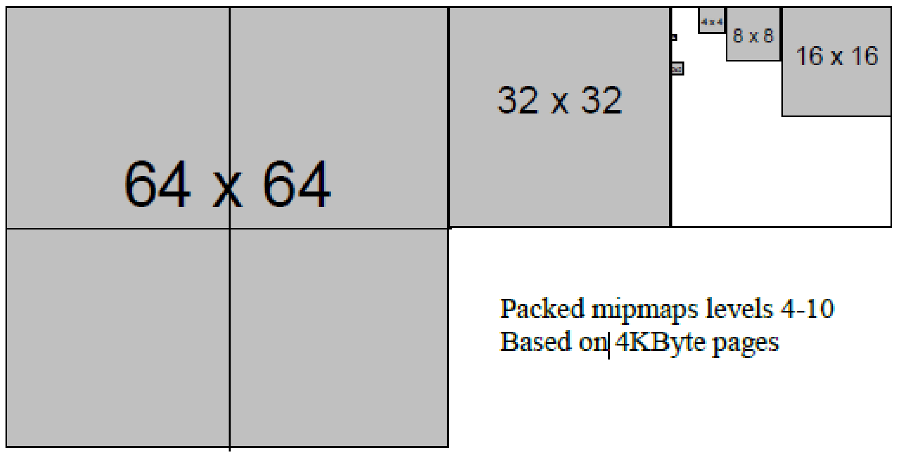

Figure 3-3 Texture LoD

Use the following parameters in glTexParameter() function to control different LoDs.

- GL_TEXTURE_BASE_LEVEL – Specifies the index of the lowest defined mipmap level; this is a non negative integer value, where the initial value is 0
- GL_TEXTURE_MAX_LEVEL – Sets the index of the highest defined mipmap level; this is a non negative integer value, where the initial value is 1000
- GL_TEXTURE_MAX_LOD – Sets the maximum level-of-detail parameter; this floating-point value limits the selection of the lowest resolution mipmap (highest mipmap level), where the initial value is 1000
- GL_TEXTURE_MIN_LOD – Sets the minimum level-of-detail parameter; this floating-point value limits the selection of highest resolution mipmap (lowest mipmap level), where the initial value is -1000.

3.1.5 PCF for depth textures

Shadow mapping creates shadows in high-end rendering for motion pictures and television. However, it has been problematic to use shadow mapping in real time applications like video games due to aliasing problems in the form of magnified *jaggies*. Shadow mapping involves projecting a shadow map on geometry and comparing the shadow map values with the light-view depth at each pixel. If the projection magnifies the shadow map, aliasing in the form of large, unsightly jaggies will appear at shadow borders. Aliasing can usually be reduced by using higher resolution shadow maps and increasing the shadow map resolution with techniques, e.g., perspective shadow maps.

Using perspective shadow-mapping techniques and increasing shadow map resolution does not work when the light is traveling nearly parallel to the shadowed surface because the magnification approaches infinity. High-end rendering software solves the aliasing problem by using a technique called percentage-closer filtering.

Unlike normal textures, shadow map textures cannot be prefiltered to remove aliasing. Instead, multiple shadow map comparisons are made per pixel and averaged together.

This technique is called percentage-closer filtering (PCF) because it calculates the percentage of surface that is closer to the light and, therefore, not in shadow. Consider the example PCF algorithm as described in Reeves et al. 1987, which called for mapping the region to be shaded into shadow map space and sampling that region stochastically; i.e., randomly. The algorithm was first implemented using the REYES rendering engine, so the region to be shaded meant a four-sided micropolygon.

Figure 3-4 is an example of that implementation.

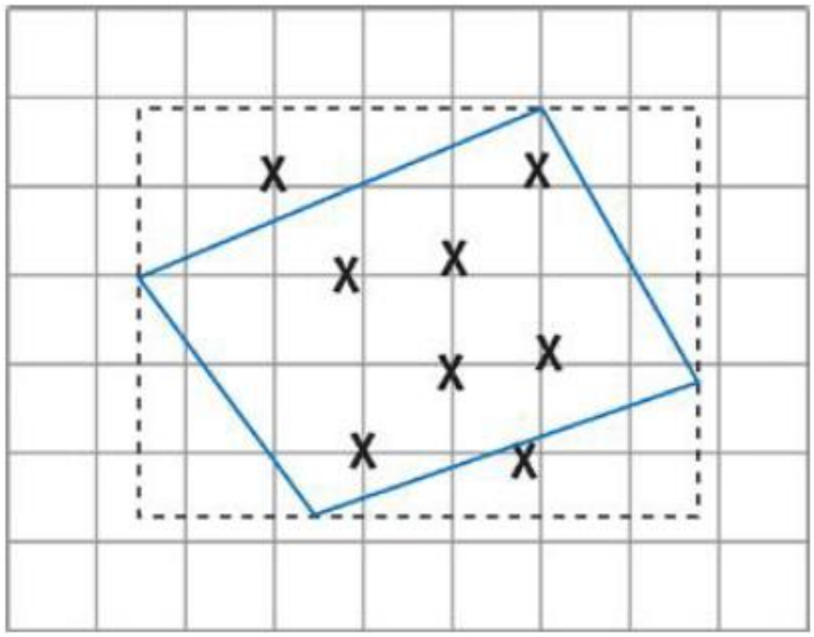

Figure 3-4 Percentage-closer filtering algorithm

Adreno 4xx has hardware support for the OpenGL ES 3.0 feature of percentage closer filtering where a hardware bilinear sample is fetched into the shadow map texture, thereby alleviating the aliasing problem with shadow mapping, as shown in Figure 3-5.

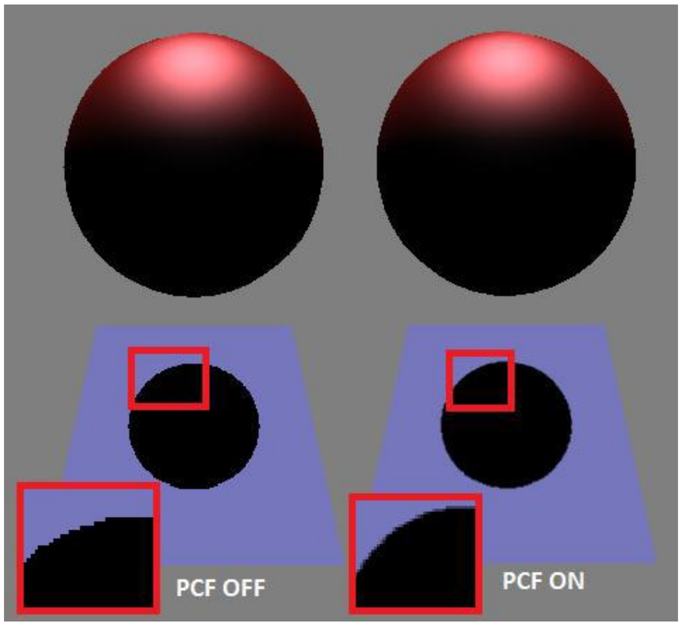

Figure 3-5 Percentage-closer filtering from the Adreno SDK

To understand how to use this feature, refer to the OpenGL ES 3.0 PCF sample from the Adreno SDK.

3.1.6 New internal texture formats

OpenGL ES 3.0 introduces sized internal formats that can be used to define texture data contents. Texture contents can now be expressed using floating-point, signed and unsigned integer internal formats, as well as in a number of different size-optimized formats. Two new internal formats can be used to store color information expressed in the sRGB color space.

Table 3-1 Internal texture formats supported in ES 3.0

Type	Internal formats
Depth	▪ GL_DEPTH_COMPONENT16 ▪ GL_DEPTH_COMPONENT24 ▪ GL_DEPTH_COMPONENT32F
Depth+Stencil	▪ GL_DEPTH24_STENCIL8 ▪ GL_DEPTH32F_STENCIL8

Type	Internal formats
Floating-point	▪ GL_R11F_G11F_B10F ▪ GL_R16F ▪ GL_R32F ▪ GL_RG16F ▪ GL_RG32F ▪ GL_RGB16F ▪ GL_RGB32F ▪ GL_RGB9_E5 ▪ GL_RGBA16F ▪ GL_RGBA32F
Signed integer	▪ GL_R16I ▪ GL_R32I ▪ GL_R8I ▪ GL_RG16I ▪ GL_RG32I ▪ GL_RG8I ▪ GL_RGB16I ▪ GL_RGB32I ▪ GL_RGB8I ▪ GL_RGBA16I ▪ GL_RGBA32I ▪ GL_RGBA8I
Signed normalized	▪ GL_R8_SNORM ▪ GL_RG8_SNORM ▪ GL_RGB8_SNORM ▪ GL_RGBA8_SNORM
sRGB	▪ GL_SRGB8 ▪ GL_SRGB8_ALPHA8
Unsigned integer	▪ GL_R16UI ▪ GL_R32UI ▪ GL_R8UI ▪ GL_RG16UI ▪ GL_RG32UI ▪ GL_RG8UI ▪ GL_RGB10_A2UI ▪ GL_RGB16UI ▪ GL_RGB32UI ▪ GL_RGB8UI ▪ GL_RGBA16UI ▪ GL_RGBA32UI ▪ GL_RGBA8UI
Unsigned normalized	▪ GL_R8 ▪ GL_RG8 ▪ GL_RGB5_A1 ▪ GL_RGB565 ▪ GL_RGB10_A2 ▪ GL_RGB8 ▪ GL_RGBA4 ▪ GL_RGBA8

For sized internal formats, it is guaranteed that the actual resolution of the internal texture data storage matches the size defined by the format.

Tip

For backward compatibility, unsized internal formats continue to be supported. However, using them may cause undesirable interactions between different OpenGL ES extensions and other corner cases. Using the new sized internal formats instead is recommended.

3.1.7 Transform feedback

It is becoming more popular to perform computations on the GPU and reuse the result data in subsequent draw calls.

Unfortunately, in the core version of OpenGL ES 2.0, there was only one feasible way of using the GPU for general-purpose computing. All calculations had to be done in the fragment shader stage, storing the results in a color attachment of the currently bound draw frame buffer. There were many limitations to this approach, including:

- Core ES 2.0 only supported rendering to renderbuffers and textures using GL_RGBA4, GL_RGBA5_A1 and GL_RGB565 internal formats, which were very limited in terms of supported precision.
- Core ES 2.0 did not support color-renderable floating-point internal format.
- Core ES 2.0 only supported one color attachment per frame buffer.

OpenGL ES 3.0 introduces support for transform feedback, which allows the capturing of output variable values, leaving the vertex shader stage. Once captured, the values can be transferred to one or more buffer object regions in two different ways:

- A single buffer object region can be used to store values of the varyings in the order specified by the application.
- Multiple buffer object regions can be used. In this case, each varying is assigned to a different buffer object region. The multiple buffer object regions may be part of the same buffer object, but this is not necessarily the case.

Using a single buffer object region, the maximum number of components that can be captured from a single vertex shader is guaranteed to be at least 64. Using multiple buffer object regions, the maximum number of components that can be captured from a single vertex shader is guaranteed to be at least 16.

Given the range of new vertex data types supported in ES 3.0, transform feedback offers possibilities comparable to compute shaders, which is a feature not available in OpenGL ES until OpenGL ES 3.1.

Example use cases include:

- Debugging – Investigate the data that the vertex shader operates on or check the values that are passed to the fragment shader stage
- GPU – Accelerated data processing
- Physics – Boids and particle systems

3.1.8 Instanced draw calls

The OpenGL ES 2.0 API supports two types of draw calls:

- Using vertex array data with glDrawArrays
- Using index data with glDrawElements

OpenGL ES 3.0 introduces a new type of draw calls called instanced draw calls. The names of the new API entry points are formed by suffixing the above function names with Instanced:

- glDrawArraysInstanced
- glDrawElementsInstanced

The key feature of the instanced draw call is that it executes repeated draw operations with a user-specified repeat count. Each repeat of the draw operation is called an instance. The vertex shader can use the new ES Shading Language constant gl_VertexID. This constant holds the index value of the draw call instance for this shader invocation.

With the new entry points, a supporting feature called vertex attribute divisor was introduced. For attributes backed by enabled vertex attribute arrays, the divisor allows specification of a rate at which the values exposed in the vertex shader via attributes should advance:

- If the rate is set to 0, the affected attribute advances once per vertex.
- Otherwise, the rate defines the number of instances that need to be drawn before the attribute advances.

Vertex attribute divisors are useful for specifying vector properties that take different values per instance (or number of instances), e.g., color, material ID, and model matrices.

A question that is often brought up in the context of instanced draw calls is: “Why? What do instanced draw calls provide, compared to a series of draw calls executed one after another?” The answer is two-fold:

- With instanced draw calls, the driver needs to perform state validation only once per call, not once per instance. Using a sequence of noninstanced draw calls, the driver would have to perform the validation many times, once for each call. This includes vertex array object validation, also frame buffer and texture completeness checks. These tasks all take a certain amount of time to execute and the cumulative effect can be significant.
- Any memory transfer operations need to be performed only for the first instance, the same memory being reused for subsequent instances.

For best performance it is crucial to find the optimal balance between the complexity of the geometry drawn by each instance and the number of instances used. It might be faster to use a smaller number of instances, each instance drawing a larger number of primitives. Always profile application rendering performance to ensure the maximum performance is given from the hardware.

Figure 3-6 shows a use case for this feature. The birds were drawn with a single instanced draw call instead of dozens of separate noninstanced draw calls, resulting in significantly improved performance:

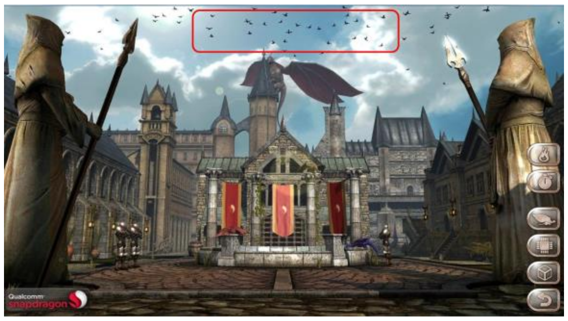

Figure 3-6 Single instanced draw call

3.1.9 Query objects

For optimal performance of any 3D application, ensure that the rendering pipeline operates asynchronously as far as possible. For an explanation, read the topic OpenGL as a Graphics Pipeline.

OpenGL ES 3.0 provides a new feature called query objects which allows a query of a number of different properties relating to the rendering process, in an asynchronous manner. These queries do not stall the rendering pipeline, unless explicitly requested to do so. This means that the application is able to keep the CPU busy with other tasks until the GPU is able to deliver the required information.

OpenGL ES 3.0 allows applications to query the following rendering pipeline properties with query objects:

- Have any of the subsequent draw calls generated fragments that passed the depth test?
- How many vertices were written into the bound transform feedback buffer(s) as a result of executing the subsequent draw calls?

A notable example of a use case for query objects is occlusion queries. This technique aims at improving rendering performance by dividing the process of drawing a complex mesh into two steps:

1. A simplified representation of the mesh is drawn. E.g., instead of drawing a teapot, a bounding box that encapsulates the object is drawn. Configure a query object to maintain a counter keeping track of fragments passing the depth test.
2. The application checks if the counter is set at a nonzero value. If so, it means the teapot is visible and it must be drawn. If not, there must be a different object located in front of it, which fully covers the teapot. In this case, it would make no sense to issue the expensive draw call.

Combined with other culling techniques, occlusion queries can significantly reduce frame rendering times. But if used wrongly they can have a serious negative impact on rendering performance. It is therefore a good idea to profile the rendering pipeline from time to time with occlusion queries first enabled and then disabled, to make sure they are actually helping performance.

The following recommendations can help make effective use of occlusion queries:

- Be careful if trying to use occlusion queries with a single-pass algorithm. It is easy to stall the driver while it waits for the query result to become available. The Adreno driver batches as many rendering commands as possible before dispatching them to the hardware. At a time when all draw calls for the rendering pass have been issued by the application and a glGetQueryObject* call has been made to retrieve the query result, it is possible that the query command is still held in a driver queue and has not yet been sent to the GPU. The safest way to use occlusion queries in single-pass algorithms is by using query data from the previous frame.
- It is easier to make performance gains using occlusion queries with multipass algorithms, as long as the query data is not requested in the same pass from which it was issued.

3.1.10 New vertex data types

OpenGL ES 3.0 introduces new types that describe vertex attribute data. These are:

- GL_HALF_FLOAT – 16-bit half floating-point values
- GL_INT – 32-bit signed integer values
- GL_INT_2_10_10_10_REV – Special packed format that allocates 10 bits for X, Y and Z components, and 2 bits for the W component; bits are interpreted as representing signed integer values
- GL_UNSIGNED_INT – 32-bit unsigned integer values
- GL_UNSIGNED_INT_2_10_10_10_REV – Same as GL_INT_2_10_10_10_REV, except that the bits are interpreted as representing unsigned integer values

3.1.11 Vertex array objects

Modification of OpenGL ES state is a serious issue for anything more than the most trivial application. State modification operations are a fundamental and essential part of the API, but the cost of these operations cannot be ignored by any application that needs a high level of performance.

One approach is to group multiple state items into a single compound state object. Complex state changes can then be carried out quickly by switching between two of these compound state objects.

OpenGL ES 3.0 follows this method. Every OpenGL ES 3.0 application may be expected to set up vertex attribute arrays for its draw calls. The more input attributes a vertex shader takes, the larger the number of vertex attribute arrays that need to be configured. This directly translates to the number of API calls that need to be made before issuing a draw call.

Vertex array objects were introduced in OpenGL ES 3.0. They encapsulate several state items including the vertex attribute array configuration. If making any of the following calls while a vertex array object is bound, then the state held by the vertex array object is updated rather than the context-wide state:

- glDisableVertexAttribArray
- glEnableVertexAttribArray
- glVertexAttribDivisor

- glVertexAttribIPointer
- glVertexAttribPointer

Any draw calls made while a vertex array object is bound to the rendering context uses the properties of the object instead of the context-wide ones.

NOTE: Vertex array objects also capture the GL_ELEMENT_ARRAY_BUFFER buffer object binding.

Once a pool of vertex array objects is configured (this can be done during a loading screen) it is no longer needed to reconfigure the vertex attribute arrays every time a draw call is made. Instead, switch to a different vertex array object using a single API call. This approach can improve rendering performance.

3.1.12 Uniform buffer objects

As seen in the discussion of vertex array objects, the OpenGL ES standard is evolving toward presenting an API focused on using as few state changes as possible.Uniform buffer objects are another example of this. As with vertex array objects, one of the goals of uniform buffer objects is to reduce the number of API calls that need to be issued before every draw call.

One common bottleneck in complex rendering pipelines is the need for frequent uniform updates. The values assigned to uniforms exist as part of the state of a program object. This means that when making the program object active, the uniform values do not need to be reloaded. However, uniforms are often used to represent properties that change frequently, e.g., model matrices, light or material settings. This means that even if sorting the draw calls, the many calls to the glUniform* entry points cannot be avoided.

The OpenGL ES 3.0 Shading Language introduces the concept of uniform blocks. This is a language construct that groups together an arbitrary set of variables and structures, and arrays of these.

NOTE: Opaque object types, such as samplers, are not supported.

Uniform blocks are defined separately for vertex and fragment shader stages, and any of the fields defined within a uniform block can be freely read from within the shader.

In OpenGL ES 2.0, uniforms were always defined at a global scope. In OpenGL ES 3.0, it is still possible to define uniforms at a global scope, in which case they are considered to be a part of the default uniform block, which has an ID of 0. The default uniform block is provided for backward compatibility and cannot be used with the new API methods introduced to support uniform buffer objects.

An important aspect of uniform blocks (other than the default uniform block) is that they no longer exist as part of a program object state. Instead, the contents of the uniform block are defined by a user-specified region of a valid buffer object. This means that the cumbersome glUniform* API does not need to be used to configure the contents of every single uniform within the program object. The approach now is to fill a region of a buffer object with values for the uniforms and then associate that region with the uniform block.

If wanting to update any of the uniforms, all there is to do is to make an update to the appropriate part of the buffer object region that has been mapped to the uniform block. This can be done using a call to the glBufferSubData function, or by mapping the buffer object region into process space and making the update there.

This functionality comes with a few limitations. The maximum uniform block size for all conformant OpenGL ES 3.0 implementations must be at least 16,384 bytes. However, use of up to 12 uniform blocks simultaneously at each shader stage is allowed. This significantly boosts the amount of uniform data the shaders are able to access. After all, the total of 196,608 bytes per stage is much more than was possible with a single default uniform block for OpenGL ES 2.0.

For example, if rendering a set of barrels and each barrel was made of a different material, the uniform block could store an array of material properties. Using uniform buffer objects, combined with instanced draw calls, a whole set of barrels can be rendered with just a single draw call.

Under the Adreno architecture, the performance of the uniform buffer object can be further improved by following these recommendations:

- Consider reorganizing the uniform buffer members if some parts of the uniform buffer are frequently updated, so as to break those parts out into a separate uniform buffer
- Avoid using sparse uniform buffers; by doing so, it will not only help improve memory usage, but could also reduce the data transfer amount needed when updating regions of the uniform buffer storage

3.1.13 Buffer subrange mapping

Buffer objects have been a part of the core OpenGL ES standard since the release of ES 1.1. However, their role used to be quite limited. Their only purpose was to back up vertex attribute arrays, so that the vertex data would be taken from VRAM instead of from client-side buffers, so as to avoid the overhead of copying vertex data on every draw call.

With the introduction of transform feedback in OpenGL ES 3.0 (see Section 3.1.7), data computed by the vertex shader now directly updates the buffer object storage. It now becomes crucial to have a means of mapping the generated data back into process space for investigation.

OpenGL ES 3.0 introduces a new API to fulfill that need. An application is now able to map a valid region of any buffer object into process space to access the existing contents or to update the memory.

Buffer subrange mapping and transform feedback together form a basis for GPGPU applications. As of version 3.0, OpenGL ES becomes a viable platform for this class of application.

3.1.14 Multiple render target support

OpenGL ES 2.0 provided support for rendering to up to three render targets at the same time. In the core version of ES 2.0, applications were not able to render to more than one color attachment, one depth attachment, and one stencil attachment.

Many modern rendering techniques rely on the ability of the fragment shader to store data into multiple render targets during the execution of a single draw call. The lack of support for multiple render targets in OpenGL ES 2.0 proved to be a major limitation for these techniques. The usual workaround was to decouple the drawing process into separate iterations. Each iteration would store a different type of information into the one available color attachment.

For instance, the first iteration could render normal data, the second could render world-space position data, and the last iteration could output albedo data. However, the cost associated with this approach meant that is was rarely viable.

The problem has been addressed in OpenGL ES 3.0 by a set of frame buffer enhancements. Using OpenGL ES 3.0, it is guaranteed to be able to draw simultaneously to at least four color attachments. Each of the available color attachments, depth attachment, or stencil attachment can now be bound to any one of the following targets:

- Renderbuffer
- Selected mipmap level of a 2D texture
- Selected mipmap level of a cube-map texture face
- Selected mipmap level of a 2D array texture layer
- Selected mipmap level of a 3D texture slice

3.1.15 Other new features

These OpenGL ES 3.0 features are only a small fraction of the changes introduced in this version.

Other new features introduced include:

- 10/10/10/2 signed and unsigned normalized vertex attributes
- 10/10/10/2 unsigned normalized and unnormalized integer textures
- 11/11/11/10 floating-point rgb textures
- 16-bit (with filtering) and 32-bit (without filtering) floating-point textures
- 16-bit floating-point vertex attributes
- 24-bit depth renderbuffers and textures
- 24/8 depth/stencil renderbuffers and textures
- 32-bit depth and 32f/8 depth/stencil renderbuffers and textures
- 32-bit, 16-bit and 8-bit signed and unsigned integer renderbuffers, textures and vertex attributes
- 8-bit srgb textures and frame buffers (without mixed RGB/SRBG rendering)
- 8-bit unsigned normalized renderbuffers
- 8-bit-per-component signed normalized textures
- Ability to attach any mipmap level to a frame buffer object
- Additional pixel store state
- At least 32 texture units, at least 16 each for fragment and vertex shaders
- Buffer object to buffer object copy operations
- Depth textures and shadow comparison
- Draw command allowing specification of range of accessed elements
- ETC2/EAC texture compression formats
- Frame buffer invalidation hints

- Indexed extension string queries
- Mandatory online compiler
- Minimum/maximum blend equations
- Multi-sample renderbuffers
- Non-power-of-two textures with full wrap mode support and mipmapping
- Non-square and transposable uniform matrices
- Opengl shading language ES 3.00
- Pixel buffer objects
- Primitive restart with fixed index
- Program binaries, including querying binaries from linked GLSL programs
- R and RG textures
- Seamless cube maps
- Shared exponent RGB 9/9/9/5 textures
- Sized internal texture formats with minimum precision guarantees
- Stretch blits (with restrictions)
- Sync objects and fence sync objects
- Texture LoD clamp and mipmap level base offset and max clamp
- Texture swizzles
- Unsigned integer element indices with at least 24 usable bits

More details on these features can be found in the the following documents:

- The OpenGL ES 3.0 Specification – http://www.khronos.org/registry/gles/specs /3.0/es_spec_3.0.3.pdf
- The OpenGL ES 3.0 Shading Language Specification – http://www.khronos.org/registry/gles/specs/3.0/GLSL_ES_Specification_3.00.4.pdf

3.2 Using key features

The following sections describe how to use the API for some of these new features.

3.2.1 Using 2D array textures

OpenGL ES 3.0 introduces a new texture target called GL_TEXTURE_2D_ARRAY. Once a generated texture object has been bound to that target using glBindTexture, it becomes a 2D array texture. Any attempt to bind that texture object to any other texture target results in a GL error. This remains the case until such time as the texture object is deleted.

Although working with 2D array textures, there are a number of functions with the name suffix 3D. This might seem confusing, but these functions can operate on either of the two texture object types:

- 2D array texture – Specify target parameter as GL_TEXTURE_2D_ARRAY
- 3D texture – Specify target parameter as GL_TEXTURE_3D

2D array textures can be initialized as mutable or immutable texture objects (see Section 3.1.3, or, for a more detailed description of the differences between the two types, refer to the OpenGL ES 3.0 Specification at https://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.)

First, look at how to set up a mutable 2D array texture. Each mipmap level defines a set of layers, and can be initialized as follows:

- Use glTexImage3D to set up the mipmap using a noncompressed internal format
- Use glCompressedTexImage3D() to set up the mipmap using a compressed internal format

Each mipmap level must be configured separately. Remember to satisfy the usual texture completeness rules when configuring the mipmap chain.

To set up an immutable 2D array texture, use glTexStorage3D instead. This entry point sets up a complete mipmap chain for the texture object in a single call and supports both compressed and uncompressed internal formats.

Now, look at how to replace the contents of a 2D array texture.

To replace one complete layer at a specific mipmap level, or just a region within the layer, use glTexSubImage3D. This works for both immutable and mutable textures.

To replace all layers at a specific mipmap level, use glTexImage3D. This works for mutable textures only and can break the texture completeness property of the object.

Use glFramebufferTextureLayer to attach a single layer from a specified mipmap level of a 2D array texture to a frame buffer. Frame buffer completeness rules apply, see the OpenGL ES 3.0 Specification at https://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

To copy a region of a currently bound read buffer to a specific layer at a given mipmap level of a 2D array texture, use glCopyTexSubImage3D.

Retrieve the ID of the texture object bound to the GL_TEXTURE_2D_ARRAY texture target of the current texture unit by issuing a glGet* query with a pname parameter value of GL_TEXTURE_BINDING_2D_ARRAY.

2D array textures hold exactly the same texture parameter state as other texture types in OpenGL ES. All glGetTexParameter* and glTexParameter* functions can be used with the new GL_TEXTURE_2D_ARRAY texture target.

In the ES Shading Language, a new set of sampler types has been introduced:

- isampler2DArray – For signed integer data types)
- sampler2DArray – For noninteger data types
- usampler2DArray – For unsigned integer data types

The following ES Shading Language texture sampling functions can be used with these new types:

- texelFetch
- texelFetchOffset
- texture
- textureGrad
- textureGradOffset
- textureLod

- textureLodOffset
- textureOffset

3.2.2 Using multiple render targets

OpenGL ES 3.0 introduces support for rendering to multiple color attachments simultaneously from a single fragment shader invocation. It is guaranteed to be able to render to at least four render targets at the same time.

Use the following functions to add and remove frame buffer attachments:

- glFramebufferRenderbuffer
- glFramebufferTexture2D
- glFramebufferTextureLayer

For detailed API descriptions for these functions, see OpenGL ES 3.0 Reference Pages at https://www.khronos.org/opengles/sdk/docs/man3/.

The following color attachments points and more are available in OpenGL ES 3.0:

- GL_COLOR_ATTACHMENT0 to render to color attachment zero
- GL_COLOR_ATTACHMENT1 to render to color attachment one
- GL_COLOR_ATTACHMENT2 to render to color attachment two

The maximum number of color attachment points for use is given by the value of GL_MAX_COLOR_ATTACHMENTS.

By default, fragment shader output is directed to a single render target only. If the default frame buffer is bound as draw frame buffer, then output is to the back buffer. If a user-supplied frame buffer is bound as draw frame buffer, then output is to the render target attached as color attachment zero of that frame buffer. The output is taken from the vector value stored in the first output variable of the fragment shader, as defined by its location.

To use multiple render targets, use glDrawBuffers to set up custom mappings from fragment shader outputs to the frame buffer color attachment points.

E.g., after binding the frame buffer as draw frame buffer, use the following call:

```
GLenum bufs[3] = {GL_COLOR_ATTACHMENT0,
                  GL_NONE,
GL_COLOR_ATTACHMENT2}; glDrawBuffers(3, bufs);
```

This sets up the mappings as follows:

- The first fragment shader output variable is directed to the render target at color attachment zero.
- The second fragment shader output variable is not used.
- The third fragment shader output variable is directed to the render target at color attachment two.

NOTE: The mapping of each output variable is constrained. The first output variable may only be mapped as color attachment zero, the second as color attachment one, etc. To be more precise, the entry at (zero-based) index position i in bufs must be eitherGL_COLOR_ATTACHMENTi or GL_NONE. This is in regards to user-generated frame buffers; the rules for the default frame buffer are different.

These bindings are stored as a part of a draw buffer configuration, which is part of frame buffer object state.

For detailed API descriptions for glDrawBuffers, see the OpenGL ES 3.0 Reference Pages at https://www.khronos.org/opengles/sdk/docs/man3/, or see the definitive OpenGL ES 3.0 Specification at https://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

3.2.3 Using query objects

To submit an asynchronous query, it is necessary to set up a query object of the appropriate type. The first step is to obtain a query object ID. Do this using the glGenQueries function.

NOTE: For some of the other OpenGL ES ID types, using the glGen* API to generate an ID is an optional step. It is possible to make up identifiers, which work just as well, provided each ID used is unique.

This is not the case for glGenQueries. Use of this API is mandatory for all query objects.

Core OpenGL ES 3.0 supports two types of queries:

- Boolean occlusion queries
- Primitive query objects

Boolean occlusion queries provide a boolean result. The result is GL_TRUE if any samples created as a result of processing a set of draw calls make it through the depth test. Otherwise, the result is GL_FALSE. To create a Boolean occlusion query, bind the query object to one of the following two target types:

- GL_ANY_SAMPLES_PASSED – The result is GL_TRUE only if any of the samples have made it through the depth test.
- GL_ANY_SAMPLES_PASSED_CONSERVATIVE – The same as GL_ANY_SAMPLES_PASSED, except that the OpenGL ES implementation is allowed to use a less precise test, which can result in false positives being returned in some cases.

Primitive query objects provide an unsigned integer result. The result is a counter value that is initially set to 0. To use a primitive query object, bind the query object to the target GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN. The counter is then incremented every time a vertex is written to one or more transform feedback buffers. For this to happen, bind the transform feedback buffer(s) and enabled transform feedback mode before making the draw call.

Use the glBeginQuery function to bind a query object to a query target type, and to mark the start of a set of OpenGL ES commands for which the query result is to be determined.

Use the glEndQuery function to mark the end of the set of commands.

To retrieve the result of the query, call the function glGetQueryObjectuiv with the pname parameter set to GL_QUERY_RESULT.

NOTE: OpenGL ES implementations are asynchronous in their nature, so query object results are not available immediately after a glEndQuery call. Before attemptting to retrieve the result value, always check its availability by calling glGetQueryObjectuiv with the pname parameter set to GL_QUERY_RESULT_AVAILABLE. Otherwise, it risks a pipeline stall.

When no longer needing a query object, release it with a glDeleteQueries call.

3.2.4 Using vertex array objects

A vertex array object stores the following information for each vertex attribute array:

- Enabled/disabled state (GL_VERTEX_ATTRIB_ARRAY_ENABLED)
- Buffer object ID to be used as the vertex attribute array data source (GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING)
- Normalization setting (GL_VERTEX_ATTRIB_ARRAY_NORMALIZED)
- Pointer setting (GL_VERTEX_ATTRIB_ARRAY_POINTER)
- Size setting (GL_VERTEX_ATTRIB_ARRAY_SIZE)
- Stride setting (GL_VERTEX_ATTRIB_ARRAY_STRIDE)
- Uses unconverted integers setting (GL_VERTEX_ATTRIB_ARRAY_INTEGER)
- Vertex attribute divisor (GL_VERTEX_ATTRIB_ARRAY_DIVISOR)
- Buffer object ID to be used as the source of index data for indexed draw calls (GL_ELEMENT_ARRAY_BUFFER_BINDING)

NOTE: Vertex array objects do not store generic vertex attribute settings (static vector values that can be assigned using glVertexAttrib* functions). These are considered to be part of the program object state instead.

Before using a vertex array object, generate one or more IDs using glGenVertexArrays.

NOTE: As with query objects, use of the glGen* API is mandatory for vertex array objects.

By default, vertex array objects are configured as follows:

- None of the attributes are backed by vertex attribute arrays
- No buffer object is bound to the GL_ELEMENT_ARRAY_BUFFER buffer object binding point

Further information about vertex array object configuration defaults can be found in the OpenGL ES 3.0 Specification http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

Before modifying the vertex array object configuration or use it for draw calls, first bind it to the rendering context using glBindVertexArray.

NOTE: For backward compatibility with OpenGL ES 2.0, a vertex array object with ID 0 is bound by default. While this default vertex array object is bound, the only available source for index and vertex data is client-side data pointers.

Tip

Avoid using client-side data pointers at all costs. They are very expensive in terms of performance. Every time a draw call is made using vertex attribute arrays backed by client-side data buffers, the attribute data needs to be copied from client process memory to video memory (VRAM).

Once the vertex array object is bound, set the vertex attribute array state using the functions glVertexAttribIPointer and glVertexAttribPointer. This has not changed much since OpenGL ES 2.0. For more information, see OpenGL ES 3.0 Reference Pages at http://www.khronos.org/opengles/sdk/docs/man3/html/glVertexAttribPointer.xhtml.

The same principles that apply to setting vertex attribute array state also apply to getting vertex attribute array state. If a nondefault vertex array object is bound to the rendering context, then the glGetVertexAttrib* functions operate on that object rather than on the generic context state.

Tip

Quickly switch between different vertex array objects using glBindVertexArray. This is much faster than reconfiguring the vertex attribute arrays on-the-fly before every draw call.

When finished with them, release one or more vertex array objects by calling glDeleteVertexArrays.

3.3 Walkthrough of sample applications

This section describes three OpenGL ES 3.0 demo applications from the Adreno SDK. It explains briefly how each demo works and highlights the most interesting features of the source code.

3.3.1 2D array textures – Demo

Find this demo at the following directory: <SDK_install_dir>\Development\Tutorials\OpenGLES \20_Texture2DArrayOGLES30. The core of the implementation is found in the file main.cpp.

The application demonstrates how to sample from 2D array textures. The source code locations described in this section are:

- Initialize function – Block marked with the comment //Create a texture, loaded from an image file.
- Render function – How the demo application renders, at the API level.
- Vertex and fragment shader bodies – How a 2D array texture is sampled in ES Shading Language and how the rendering pipeline is organized in this demo application.

3.3.1.1 Initialization

Initialization is done by the Initialize function. It can be broken down into three separate parts:

1. Two textures are loaded from TGA files
2. A 2D array texture is created and initialized
3. A program object is constructed and linked, using fragment and vertex shader bodies defined within the demo application code

Loading the textures does not use OpenGL ES, so this document ignores it.

Setting up the 2D array texture is of more interest. Here is a breakdown of what happens in the code:

```
glGenTextures( 1, &g_hTextureHandle );
glBindTexture( GL_TEXTURE_2D_ARRAY, g_hTextureHandle );
```

A single texture object ID is generated and is associated with a 2D Array Texture target.

```
glTexParameteri( GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER,
GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER,
GL_LINEAR );
```

Magnification and minification filtering for the texture object is set to GL_LINEAR. This is important because the default setting for minification filtering requires mipmaps, and the demo application is not creating any. If this step was skipped, the texture object would be considered incomplete and any texture sampling functions used on this texture object would always return (0, 0, 0, 1) instead of the a texture lookup result.

```
glTexImage3D( GL_TEXTURE_2D_ARRAY, 0, internalFormat, nWidth[0],
nHeight[0], g_nImages, 0, nFormat[0],         GL_UNSIGNED_BYTE,
NULL );
```

This call allocates storage space for two layers at mipmap level zero.

NOTE: The layer contents are not uploaded in this call.

```
for( int i = 0; i<g_nImages; i++) {     glTexSubImage3D(
GL_TEXTURE_2D_ARRAY, 0, 0, 0, i, nWidth[i],
nHeight[i], 1, nFormat[i], GL_UNSIGNED_BYTE,
pImageData[i]); }
```

This is where the layer contents are uploaded, looping to upload layer after layer.

Finally, in the last part of the function, the program object is constructed and linked. There is nothing here that is new for OpenGL ES 3.0, so this document ignores it.

3.3.1.2 Rendering a frame

Here is a closer look at the Render function:

```
glClearColor( 0.0f, 0.0f, 0.5f, 1.0f );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Start by clearing both color and depth buffers. Since no frame buffer object is bound at this time, these operations are performed using the default frame buffer.

```
glUseProgram( g_hShaderProgram ); glBindTexture(
GL_TEXTURE_2D_ARRAY, g_hTextureHandle ); The demo then activates its
program object and binds the 2D array texture object to the current texture
unit.
glVertexAttribPointer( g_VertexLoc, 4, GL_FLOAT, 0, 0,
VertexPositions ); glEnableVertexAttribArray(
g_VertexLoc );
glVertexAttribPointer( g_TexcoordLoc, 2, GL_FLOAT, 0, 0,
VertexTexcoord); glEnableVertexAttribArray(
g_TexcoordLoc ); glDrawArrays( GL_TRIANGLES, 0, 6 );
glDisableVertexAttribArray( g_VertexLoc );
glDisableVertexAttribArray( g_TexcoordLoc );
```

Finally, configure and enable two vertex attribute arrays, draw a couple of triangles, and then disable both vertex attribute arrays again.

NOTE: The vertex data is organized in such a way that the result triangles form a quad located at the center of the screen. Vertices emitted by a vertex shader invocation are positioned in a clip space which can be described as a cube spanning from (-1, -1, -1) to (1, 1, 1). When a primitive is built from these vertices, any part of that primitive not falling within the cube is clipped so that each vertex fits within the region defined by the cube. Therefore, to completely fill the surface of the two-dimensional back buffer with contents computed in the fragment shader stage, make sure that the draw call or calls generate a rectangle spanning from (-1, -1) (corresponding to the bottom-left corner) to (1, 1) (corresponding to the top-right corner).

The demo centers the rectangle in the screen space but does not entirely fill the available surface. Try experimenting with the demo implementation to gain a better understanding of how screen space rendering works.

3.3.1.3 Shaders

The body of the vertex shader body is coded in the variable g_strVSProgram.

```
#version 300 es in vec4 g_vVertex; in vec4
g_vTexcoord; out   vec4 g_vVSTexcoord; void main() {
gl_Position  = vec4( g_vVertex.x, g_vVertex.y,
g_vVertex.z, g_vVertex.w );     g_vVSTexcoord =
g_vTexcoord; }
```

In the function main, the shader sets gl_Position to the value it gets from one of its input attributes, without applying any transformations. This implies that the input vertex data is expressed in the normalized device coordinate space. Texture coordinates are provided via the other input attribute and are also passed through to the fragment shader stage untransformed.

The body of the fragment shader is coded in the variable g_strFSProgram.

```
uniform sampler2DArray g_sImageTexture;
in vec4 g_vVSTexcoord;
out vec4 out_color; void main()
{
    out_color = texture(g_sImageTexture, vec3(g_vVSTexcoord.xyx));
}
```

The shader declares a 2D array texture sampler uniform called g_sImageTexture. In the function main, the uniform is used to sample the texture.

NOTE: The second argument of the texture call includes a third component. This specifies the index of the layer that the data is sampled from.

Important

The layer index of a 2D array texture is an integer value, so to know how this is derived from the supplied floating-point component, the process is as follows:

1. Value is rounded to the nearest integer, e.g., 0.4 rounds to 0, but 0.5 rounds up to 1.0.
2. Rounded value is clamped to the number of layers. For a 2D array texture with 3 layers, this would result in an integer value between 0 and 2, inclusive.
3. Rounded and clamped value leaves the layer index that can be used for the sampling process.

3.3.2 Rendering to 2D array textures – Demo

Find this demo at the following directory: <SDK_install_dir>\Development\Tutorials\OpenGLES\25_FramebufferTextureLayerOGLES30.

This application demonstrates a few OpenGL ES 3.0 features in action, including:

- Rendering to 2D array textures
- Blitting layers of a 2D array texture to different regions of the default frame buffer

The demo uses an off-screen frame buffer to render a number of frames to successive layers of a 2D array texture. Each frame shows a pyramid, rotated to a slightly different angle. Once rendered, these layers are used to composite the back buffer contents using the frame buffer blitting mechanism. Finally, the back buffer is swapped with the front buffer to present the rendered frame to the user.

To get a better understanding of how the application works, look at the following areas:

- Configuring the off-screen frame buffer
- Carrying out the off-screen rendering
- Compositing the final rendering result

The discussion of parts of the code that are similar to those already covered in the earlier demo walkthrough are skipped.

3.3.2.1 Configuring the off-screen frame buffer

The off-screen frame buffer is constructed in CreateFBO.

```
glGenTextures( 1, &g_hTextureHandle ); glBindTexture(
GL_TEXTURE_2D_ARRAY, g_hTextureHandle );
glTexParameteri( GL_TEXTURE_2D_ARRAY,
GL_TEXTURE_MAG_FILTER,         GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER,
GL_LINEAR ); glTexImage3D( GL_TEXTURE_2D_ARRAY, 0,
nInternalFormat, nWidth,         nHeight, g_nLayers, 0,
nFormat, nType, NULL );
```

Above is the code that initializes a 2D array texture. This has 9 layers, since it is the value of the constant g_nLayers.

```
glGenRenderbuffers( 1, &(*ppFBO)->m_hRenderBuffer );
glBindRenderbuffer( GL_RENDERBUFFER, (*ppFBO)->m_hRenderBuffer );
glRenderbufferStorage( GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, nWidth,
nHeight );
```

Here, it creates a separate renderbuffer to hold depth data at 24-bit precision.

```
glFramebufferTextureLayer( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
g_hTextureHandle, 0, 0);
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, (*ppFBO)->m_hRenderBuffer );
```

Now the frame buffer is configured with the following attachments:

- Layer zero of the 2D array texture is attached to color attachment point zero
- Renderbuffer is bound to the depth attachment point

3.3.2.2 Off-screen rendering

The off-screen rendering process is done by the first part of the render function:

```
for(int i=0; i<g_nLayers; i++) {
BeginFBO( g_pOffscreenFBO, i );
glClearColor( 0.0f, 0.5f, 0.0f, 1.0f );
RenderScene( fTime+i*0.1f );
    EndFBO( g_pOffscreenFBO );
}
```

The block loops over all 2D array texture layers. For each layer, BeginFBO is used to configure the frame buffer object, which it does as follows:

```
glBindFramebuffer( GL_FRAMEBUFFER, pFBO->m_hFrameBuffer );
glFramebufferTextureLayer( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
g_hTextureHandle, 0, layer);
glViewport( 0, 0, pFBO->m_nWidth, pFBO->m_nHeight );
```

Attach the current layer as color attachment zero. Also ensure that the viewport resolution stays synchronized to the texture resolution. If that is not done, then either the pyramid would not fit in the render target, or only part of the render target would be drawn to.

Once this has been done, render makes a call to RenderScene, which is responsible for sending the commands needed to render the pyramid.

```
glBindFramebuffer( GL_FRAMEBUFFER, 0 );
glViewport( 0, 0, g_nWindowWidth, g_nWindowHeight );
```

Finally, in EndFBO, bind back to the default frame buffer and reset the viewport resolution to the dimensions of the window.

After the render loop has finished executing, all 2D array texture layers are filled with pyramids rotated to different angles. At this point, none of these pyramids have made it into the back buffer yet.

3.3.2.3 Compositing the final image

The composition process is implemented by the remaining part of the render function.

```
FLOAT offsets[9][4] =
{
{ 0.0f,      0.0f,      1.0f/3.0f, 1.0f/3.0f },
{ 1.0f/3.0f, 0.0f,      2.0f/3.0f, 1.0f/3.0f },
{ 2.0f/3.0f, 0.0f,      1.0f,      1.0f/3.0f },
{ 0.0f,      1.0f/3.0f, 1.0f/3.0f, 2.0f/3.0f },
{ 1.0f/3.0f, 1.0f/3.0f, 2.0f/3.0f, 2.0f/3.0f },
{ 2.0f/3.0f, 1.0f/3.0f, 1.0f,      2.0f/3.0f },
{ 0.0f,      2.0f/3.0f, 1.0f/3.0f, 1.0f },
{ 1.0f/3.0f, 2.0f/3.0f, 2.0f/3.0f, 1.0f },
{ 2.0f/3.0f, 2.0f/3.0f, 1.0f,      1.0f },
};
```

This starts with an array defining the regions where the pyramid images are to be placed. Nine rectangles are defined. The four values given for each rectangle are as follows:

- X coordinate of the top-left corner
- Y coordinate of the top-left corner

- X coordinate of the bottom-right corner
- Y coordinate of the bottom-right corner

```
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, 0 );
glBindFramebuffer( GL_READ_FRAMEBUFFER,
g_pOffscreenFBO->m_hFrameBuffer );
```

Prepare the frame buffer bindings prior to entering the loop that will do the compositing:

- The default frame buffer is made the target of all rendering operations to blit the pyramid images to the back buffer.
- The off-screen frame buffer is set to be the source for read operations; this tells glBlitFramebuffer where the source data should be taken from.

```
for(int i=0; i<g_nLayers; i++)
{
    glFramebufferTextureLayer( GL_READ_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, g_hTextureHandle, 0, i);     glBlitFramebuffer(
0, 0, g_pOffscreenFBO->m_nWidth,             g_pOffscreenFBO-
>m_nHeight,             INT32(g_nWindowWidth*offsets[i][0]),
            INT32(g_nWindowHeight*offsets[i][1]),
INT32(g_nWindowWidth*offsets[i][2]),
            INT32(g_nWindowHeight*offsets[i][3]),
            GL_COLOR_BUFFER_BIT, GL_LINEAR );
}
```

For each layer of the 2D array texture, two things must be done to copy the image data to the back buffer:

1. Configure this layer as color attachment zero of the read frame buffer; this will be the source of the blit operation
2. Use glBlitFramebuffer to blit one pyramid image from the read frame buffer to the back buffer at the location defined by the rectangle co-ordinates from the array

When the loop has finished and the render function returns, the final image is now fully composited in the back buffer. It is now ready to swap the back buffer with the front buffer, so that the scene is made visible on the screen.

3.3.3 Interleaved vertex buffer objects – Demo

Find this demo at the following directory: <SDK_install_dir>\Development\Tutorials\OpenGLES \19_InterleavedVBOOGLES30.

It demonstrates a technique for preparing and rendering a set of primitives. The attribute data for the primitives is stored in an interleaved manner in a vertex buffer object. To render them, use the new draw call glDrawRangeElements, which was introduced in OpenGL ES 3.0.

The application uses many of the same elements already discussed in the previous demos, so this demo focuses on new and interesting aspects instead.

3.3.3.1 Construction of an interleaved vertex buffer object

The demo uses a single buffer object to store the following information:

- Screen-space vertex position
- RGBA color data for each vertex

NOTE: The index data is not a part of this buffer. Indices are never taken from a vertex attribute array. Instead, they are downloaded from a buffer object bound to GL_ELEMENT_ARRAY_BUFFER.

Color and vertex data is stored in a linear fashion, one vertex after another, in the same order as it is listed above.

In the demonstration application, the data buffer is constructed in the function InitVertexAttributesData. A closer look is as follows:

```
pVbuff = new VERTEX_ATTRIBUTES_DATA;
```

The function sets up an instance of the VERTEX_ATTRIBUTES_DATA structure. This holds important properties of the vertex buffer object, as well as the data used for the draw calls.

```
pVbuff->nTotalSizeInBytes = nVsize + nCsize; pVbuff->pVertices
= (POS *)new CHAR[pVbuff->nTotalSizeInBytes]; pVbuff-
>pPosOffset = NULL; pVbuff->pColorOffset =
        (UINT8*)((UINT8*)pVbuff->pPosOffset + sizeof(POS));
pVbuff->nStride = sizeof(POS) + sizeof(COLOR);
```

The properties are configured as follows:

- nTotalSizeInBytes – Total number of bytes the vertex buffer object should use
- pPosOffset – Start offset for vertex position data
- pColorOffset – Start offset for vertex color data
- nStride – Number of bytes that separates vertex position and vertex color data between consecutive vertices; in this example, the stride for both attribute data types is exactly the same, which is why only one field is used to hold both values (see Figure 3-7)

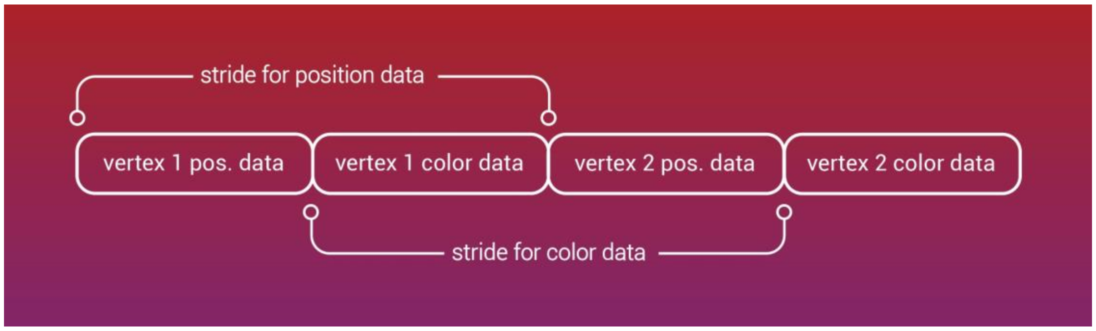

Figure 3-7 Vertex buffer stride

In the loop which follows, the vertex buffer is then populated with position and color data.

Now, look at the Render function. Here is how the properties stored in the VERTEX_ATTRIBUTES_DATA are used to configure a vertex array object for use with the vertex buffer object.

3.3.3.2 Using glDrawRangeElements

A new type of a draw call was introduced in OpenGL ES 3.0. The new function, glDrawRangeElements, builds upon the concept of glDrawElements and introduces two new parameters:

- Start – Minimum index value that is used for the purpose of the draw call
- End – Maximum index value that is used for the purpose of the draw call

If any of the index values—supplied either via the GL_ELEMENT_ARRAY_BUFFER binding or via a client-side pointer—is outside the defined range, then a mistake has been made and the resulting behavior will be undefined.

The benefit of using this type of draw call type is that if the OpenGL ES implementation knows in advance what set of index values will be used for the drawing process, then there is an opportunity to reduce the amount of vertex attribute array data that needs to be transferred in order to execute the request.

Suppose that the mesh consists of a few layers, where each layer needs to be rendered with a different shader or using a different set of textures. When rendering each layer, use glDrawRangeElements to tell the OpenGL ES implementation what range of index values the draw call will be using. In many cases, this results in improved performance.

For the demo application, the draw call is executed in the render function. This function sets up the vertex attribute arrays, configures the GL_ARRAY_BUFFER and GL_ELEMENT_ARRAY_BUFFER bindings, and then issues the draw call.

3.4 About the OpenGL ES implementation

This section gives details of the capabilities of the Adreno architecture within an OpenGL ES 3.0 context.

3.4.1 GL constant values

Table 3-2 GL_MAX constant values – OpenGL ES 3.0

Pname	Value
GL_MAX_3D_TEXTURE_SIZE	2048
GL_MAX_ARRAY_TEXTURE_LAYERS	2048
GL_MAX_COLOR_ATTACHMENTS	8
GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	197504
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS	32
GL_MAX_COMBINED_UNIFORM_BLOCKS	24
GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	197632
GL_MAX_CUBE_MAP_TEXTURE_SIZE	16384
GL_MAX_DRAW_BUFFERS	8
GL_MAX_ELEMENT_INDEX	2147483647
GL_MAX_ELEMENTS_INDICES	2147483647
GL_MAX_ELEMENTS_VERTICES	134217727
GL_MAX_FRAGMENT_INPUT_COMPONENTS	135
GL_MAX_FRAGMENT_UNIFORM_BLOCKS	12
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS	896
GL_MAX_FRAGMENT_UNIFORM_VECTORS	224
GL_MAX_PROGRAM_TEXEL_OFFSET	7
GL_MAX_RENDERBUFFER_SIZE	16384
GL_MAX_SAMPLES	4
GL_MAX_SERVER_WAIT_TIMEOUT	1000000000
GL_MAX_TEXTURE_IMAGE_UNITS	16
GL_MAX_TEXTURE_LOD_BIAS	31
GL_MAX_TEXTURE_SIZE	16384
GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	128
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	4
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	4
GL_MAX_UNIFORM_BLOCK_SIZE	65536
GL_MAX_UNIFORM_BUFFER_BINDINGS	24
GL_MAX_VARYING_COMPONENTS	128
GL_MAX_VARYING_VECTORS	32
GL_MAX_VERTEX_ATTRIBS	32
GL_MAX_VERTEX_OUTPUT_COMPONENTS	133
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS	16
GL_MAX_VERTEX_UNIFORM_BLOCKS	12
GL_MAX_VERTEX_UNIFORM_COMPONENTS	1024
GL_MAX_VERTEX_UNIFORM_VECTORS	256
GL_MAX_VIEWPORT_DIMS	16384 x 16384

Table 3-3 Other GL constant values – OpenGL ES 3.0

Pname	Value
GL_ALIASED_LINE_WIDTH_RANGE	1.0 to 8.0
GL_ALIASED_POINT_SIZE_RANGE	1.0 to 1023.0
GL_MIN_PROGRAM_TEXEL_OFFSET	-8
GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT	4

4 Using OpenGL ES 3.1 with Adreno

4.1 New features in OpenGL ES 3.1

This section provides a short introduction to some of the more important features introduced in OpenGL ES 3.1. For further details, see the following:

- OpenGL ES 3.1 Specification – http://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf
- OpenGL ES Shading Language 3.10 Specification – http://www.khronos.org/registry/gles/specs/3.1/GLSL_ES_Specification_3.10.pdf
- OpenGL ES 3.1 Reference Pages – http://www.khronos.org/opengles/sdk/docs/man31/

4.1.1 Atomic counters

OpenGL ES 3.1 provides a new class of unsigned integer variables called atomic counters. These counters can be accessed by shaders using different atomic operations. The atomic nature of these operations means that when multiple shader invocations attempt to access a single atomic counter at the same time, these accesses will be serialized so that no thread races occur.

To define an atomic counter in a shader, use the new ES Shading Language type atomic_uint. The type is opaque, so the only way of accessing or manipulating the counter value from the shader is by using one of the new accessor functions, which include:

- atomicCounter – Returns the counter value
- atomicCounterDecrement – Decrements the counter value and returns the new value
- atomicCounterIncrement – Increments the counter value and returns the value prior to the increment operation

Atomic counters must be backed by buffer object storage space. Use the new indexed buffer object binding point, GL_ATOMIC_COUNTER_BUFFER, to associate the atomic counters with regions of buffer objects. Multiple atomic counters are allowed to use a single buffer object, as long as their storage space does not intersect.

To associate an atomic counter with a unique buffer object region, there are two new layout qualifiers to use when declaring an atomic counter:

- Binding – Specifies the index for the buffer object binding point, e.g., it determines which buffer object will be used and must always be specified
- Offset – Specifies the offset in bytes of the atomic counter within the buffer object

For more details about using these qualifiers, see the OpenGL ES 3.1 Shading language Specification at http://www.khronos.org/registry/gles/specs/3.1/GLSL_ES_Specification_3.10.pdf.

For example, suppose the task is to find out which of the values passed in the input dataset are prime numbers. Solve this problem by implementing a compute shader.

A single compute shader invocation would take the unique work group and work item ID, convert this information to an unique entry index, and use that index to address the input dataset to retrieve the candidate number. Once the invocation knows which value to process, it can move on and perform necessary checks that would determine whether the value is a prime. If the input value is indeed found to be a prime, then the compute shader invocation could store it in external storage. But how do atomic counters fit in the picture here?

Use the counters as a means of obtaining index values that will be guaranteed to be unique across all compute shader invocations that have been scheduled to run. Use an OpenGL ES Shader Language instruction that increments the atomic counter, returning the value of the counter prior to the increment. Should more than one shader invocation attempt to increment the counter at the same time, it is guaranteed these requests will be serialized. Once the compute shader invocation has obtained an unique counter value, it can use it to store the prime number that it has found in a shader storage buffer or an image, at an offset that will not be overwritten by any other compute shader invocation. Finally, once all the compute shader invocations finish executing, the counter value serves as a count of the number of primes found. The application can then download the prime number values from the buffer object region or the texture mipmap that was used for result storage, back into process space.

NOTE: In core OpenGL ES 3.1, support for atomic counters is guaranteed only at the compute shader stage. They may be supported at other shader stages as well, but this is not necessarily the case.

4.1.2 Compute shaders

OpenGL ES 3.1 introduces a completely new type of shaders known as compute shaders. These do not form part of the normal rendering pipeline and can be used to execute data processing tasks using the GPU.

The isolated nature of compute shaders has a number of consequences:

- They do not support input attributes or output variables
- They are not invoked once per vertex as are vertex shaders
- No other shader stage precedes or follows their execution

However, they are like regular shaders in that they can access atomic counters, image variables, shader storage buffers, textures, uniforms, and uniform blocks.

For instance, how does a compute shader communicate with the outside world? After all, it cannot use input attributes or output variables. The answer is that it can use the object types mentioned above. Of these, only atomic counters, image variables, and shader storage buffers can be written to by the shader as it runs. When a compute shader wants to store the result of its work, it can use one of the new atomic counter functions, update the contents of an image, or write to a buffer variable.

Because compute shaders operate outside the normal rendering pipeline, they are not invoked via a draw call. Instead, the new glDispatchCompute function is used to launch the operation.

Before the computation can start, the work to be done must be split into work units.

Each unit is processed by a work group. A single work group consists of a number of invocations, which can potentially process that work unit in parallel. The number of invocations used is defined by the shader. The invocations are arranged in a 1D, 2D or 3D grid, according to the needs of the shader. The dimensions of the grid define the local work group size. There are implementation-specific constraints on the maximum size of a local work group.

Those constraints are the reason why work must usually be split into multiple units. glDispatchCompute takes a set of arguments defining work group counts in the X, Y and Z dimensions. This allows the initiation of the processing of 3-dimensional array of work units with a single API call. Recall that each of those work units can be made up of a 3-dimensional array of shader invocations.

The invocations within a single work group can synchronize their execution using the new ES SL function groupMemoryBarrier. They can communicate with each other using shared variables. They can also exchange information using writable objects such as images, provided that they synchronize access using with one of the new ES SL functions: barrier, memoryBarrier*, or groupMemoryBarrier.

Important

Work groups can be executed in any order and are not guaranteed to be executed in parallel. This means that any attempt to synchronize execution flow or resource access between different work groups can potentially lead to deadlocks. It is not possible to communicate or synchronize execution flow between multiple work groups, only between different invocations within the same work group.

Compute shaders allow for more flexibility in the way work is organized, thanks to shared variables, which are available in compute shaders, but not in other shader types. As a result, it is becoming increasingly common for them to be used for AI, physics, or post-processing effects.

Tip

Do not interleave compute and graphics shaders. Under the Adreno architecture, switching between the two pipeline types is expensive and should be avoided. Instead, batch the draw calls and dispatch calls to reduce the number of times the driver must switch.

4.1.3 ES shading language enhancements

The following enhancements have been introduced to OpenGL ES 3.1 Shading Language:

- The binding layout qualifier can now be used to specify an initial binding point associated with a uniform block or sampler uniform
- The function to split a floating-point number into significand and exponent (frexp)
- The function to build a floating-point number from significand and exponent (ldexp)
- The functions to perform 32-bit unsigned integer add and subtract operations with carry or borrow (uaddCarry, usubBorrow)
- The functions to perform 32-bit signed and unsigned multiplication, with 32-bit inputs and a 64-bit result spanning two 32-bit outputs (imulExtended, umulExtended)
- The functions to perform bit-field extraction, insertion and reversal (bitfieldExtract, bitfieldInsert, bitfieldReverse)
- Multi-dimensional arrays can now be defined in ES SL code

- The number of bits set to 1 in an integer value can now be determined with a single call (bitCount)
- The position of the most or least significant bit set to 1 can now be determined with a single function call (findLSB, findMSB)
- Texture gather functionality is now available in the ES Shading Language (textureGather, textureGatherOffset); these functions can be used to retrieve the 2x2 footprint that is used for linear filtering in a texture lookup operation
- The functions to pack four 8-bit integers to a 32-bit unsigned integer and to unpack a 32-bit unsigned integer to four 8-bit integers (packUnorm4x8, packSnorm4x8, unpackUnorm4x8, unpackSnorm4x8)
- The locations of uniforms in the default uniform block can now be preconfigured directly in ES Shading Language by using the location layout qualifier

ES Shading Language features covered in other sections of this chapter have been omitted from the above list.

4.1.4 Images and memory barriers

Earlier in the chapter, atomic counters were introduced and showed how they provide a means of communication between a shader—such as a compute shader—and the outside world. However, atomic counters are a limited tool. They can only represent unsigned integer values, and the set of functions that operate on them is restricted. If the shader needs a more powerful means of data exchange, one tool for use is images, which is another new feature introduced in OpenGL ES 3.1.

An image is an opaque uniform that points to a specific level of a texture. A number of different texture types are supported by images, for instance:

- 2D textures (defined by the ES Shader Language types image2D, iimage2D, and uimage2D)
- 2D array textures (defined by the ES Shader Language types image2DArray, iimage2DArray, and uimage2DArray)
- 3D Textures (defined by the ES Shader Language types image3D, iimage3D, and uimage3D)
- Cubemap textures (defined by the ES Shader Language types imageCube, iimageCube, and uimageCube)

Images are restricted to using a subset of the internal formats available in OpenGL ES 3.1:

- GL_R32I
- GL_R32F
- GL_R32UI
- GL_RGBA16F
- GL_RGBA16I
- GL_RGBA16UI
- GL_RGBA32I
- GL_RGBA32F
- GL_RGBA32UI
- GL_RGBA8

- GL_RGBA8I
- GL_RGBA8UI
- GL_RGBA8_SNORM

Once a mipmap level of a texture is assigned to an image, it can be read from and written to that image directly from a shader. The cost associated with that operation is usually greater than that of using atomic counters, so only use images when it is necessary to do so.

NOTE: When performing load operations on an image, the texel location must be provided using integer coordinates. No texture filtering capabilities are provided by images.

Any conforming OpenGL ES 3.1 implementation will support at least four image uniforms at the compute shader stage. If images are used at other shader stages, support is not guaranteed, so always check first, using the appropriate GL_MAX_*_IMAGE_UNIFORMS constant.

When using images, pay attention to memory consistency issues. There are a number of factors to consider, including:

- Shader invocations are executed in a largely undefined order
- The underlying memory that can be accessed through an image can also be changed by other invocations on-the-fly
- OpenGL ES may cache store operations from one shader invocation, so other invocations may not see the update
- OpenGL ES is also allowed to cache values fetched by memory reads and to return the cached value to any shader invocation accessing the same memory

It is possible to exert a degree of control over these OpenGL ES behaviors by using the following memory qualifiers on an image declaration:

- Coherent – Any write operations performed on this image must be reflected in the results of reads subsequently performed by other shader invocations
- Volatile – Any read operations performed on the image must reflect the results of updates to the underlying memory, which may have been made by another shader invocation
- Restrict – A hint to the OpenGL ES implementation and it is asserted that the underlying memory will be modified only from the current shader stage and through the image defined using this keyword
- Readonly – Restricts the image to use for load operations
- Writeonly – Restricts the image to use for store operations; memory accesses across multiple shaders are largely unsynchronized:
  - Relative order of reads and writes to a single shared memory address from multiple separate shader invocations is largely undefined
- Order of accesses to multiple memory addresses performed by a single shader invocation, as observed by other shader invocations, is also undefined

To synchronize memory transactions, shaders can use the new memoryBarrier function. This function waits for the completion of all pending memory accesses relating to the use of image variables. The results of all store operations performed on coherent variables prior to the memoryBarrier call will be visible to load operations subsequently executed in any shader invocation at any shader stage.

These are the synchronization constructs for use in shader code. A memory barrier may need to be injected at the API level (see glMemoryBarrier at http://www.khronos.org/opengles/sdk/docs/man31/html/glMemoryBarrier.xhtml in the OpenGL ES 3.1 Reference Pages). Consider a shader that accesses the same memory that is being used for shader image load/store operations but by other means, e.g., texture fetch. The memoryBarrier function in shader code is only guaranteed to synchronize those memory accesses that are made using image variables and atomic counters. Correct synchronization in such a case may require use of an API-level memory barrier.

Images are useful for all techniques that need to update arbitrary locations of a texture on-the-fly. Example use cases include dynamic scene voxelization.

4.1.5 Indirect draw calls

In OpenGL ES 3.0, when making a draw call, function parameters such as the following needed to be passed:

- Primitive mode
- Start index
- Number of indices to be rendered
- Other parameters, depending on the type of the draw call

In the context of OpenGL ES 3.0, this was quite sufficient, given the lack of tools that could have been used to generate content on-the-fly directly on the GPU. However, with the advent of compute shaders, shader storage buffer objects (SSBOs), and atomic counters, the need arose for a draw call that could source these parameter values from information stored in VRAM.

For instance, consider a case where a compute shader processes the contents of a rendered frame and detects bright locations in that frame. These locations are then stored in an image or in an SSBO. That information will later be used to blend the bright spots with small Bokeh-textured quads. The compute shader needs to use an atomic counter to keep track of how many spots have been detected. Remember that atomic counters use buffer object storage. That means that the number of bright spots is available in VRAM, and it should be able to pass that value directly into the draw call. Without indirect draw calls—introduced in OpenGL ES 3.1—that would not have been possible. It would have needed to map the buffer object region into process space, to read the counter value, and then to pass the value back to OpenGL ES as a draw call parameter.

OpenGL ES 3.1 solves this problem with indirect draw calls:

- glDrawArraysIndirect – Same functionality as glDrawArraysInstanced
- glDrawElementsIndirect – Same functionality as glDrawElementsInstanced

The indirect versions of these two draw calls read their input parameter values from a buffer object bound to the GL_DRAW_INDIRECT_BUFFER binding point, instead of taking them as formal parameters of the function call. The exception is that the mode and type arguments are still passed as formal parameters.

A previous section of this chapter discusses compute shaders. The function used to launch compute shaders, glDispatchCompute, also has an indirect version called glDispatchComputeIndirect. In this case, the binding point used for the parameter values is GL_DISPATCH_INDIRECT_BUFFER rather than GL_DRAW_INDIRECT_BUFFER.

4.1.6 Multisample textures

Rendering to multisample attachments was introduced in OpenGL ES 3.0. The support came with a few notable limitations:

- Use of renderbuffers for the process
- Renderbuffer requirement implied that multisampled data in the shaders could not be sampled and that the contents had to be "flattened" by blitting the multisample renderbuffer storage into a single-sampled regular texture, which could then be accessed using the usual texture sampling methods
- No feasible way of reading individual sample values
- Not possible to specify at the API level which samples should be modified during the execution of a draw call

These are some of the issues that multisample textures, introduced in OpenGL ES 3.1, aim to address.

Multisample textures can be created using the new glTexStorage2DMultisample entry point with the GL_TEXTURE_2D_MULTISAMPLE texture target. Mutable multisample textures are not supported.

Multisample textures do not have mipmaps. Also, since the actual physical layout of the underlying data is hardware dependent, the only way to write data to multisample textures is by rendering to them. Likewise, the only permissible way of reading the contents is by sampling the texture. Sampling can only be done using nearest filtering. If the texture is configured for linear or trilinear filtering, it will be considered incomplete.

To render to a multisample texture, attach it to one of the frame buffer object attachment points. Do this using the glFramebufferTexture2D function.

NOTE: Frame buffer completeness rules require all attachments to be multisample if any one of them is.

To sample from a multisample texture, use a new sampler type called sampler2DMS. A multisample texture sampler can use only one texture sampling function, texelFetch. In addition to the usual parameters of a sampling function, texelFetch takes an additional integer parameter specifying the sample index for sampling.

NOTE: The texelFetch function takes integer texture coordinates. This makes it clear that bilinear interpolation is not supported.

The new API function glSampleMaski provides the ability to mask the set of samples to be updated on subsequent draw calls. This will work for frame buffer attachments of both types: the new multisample textures and the renderbuffers used for multisampling in OpenGL ES 3.0.

The most significant use-case for multisample textures is in deferred renderers. These are now able to use more complex anti-aliasing mechanisms, because they now have a chance to access individual samples when sampling G-buffer contents.

Tip

Multisample textures are expensive, as the Adreno driver needs to load or store <number of samples> * <number of bytes per surface> bytes when switching render targets. Consider blitting the multi-sample textures to single-sample containers as soon as the multi-sample data is no longer needed. Then use the single-sampled representation instead of the multi-sample one, saving on both bandwidth and memory usage.

4.1.7 Separate shader objects

One of the key object types in OpenGL ES is program objects. They are needed to carry out any kind of draw call. As applications become more complex, it is not unusual for a modern OpenGL ES applications to create hundreds of program objects.

Use of a large number of program objects carries some costs:

- Longer loading times, as each program object needs to be linked (or loaded as a blob from external storage) before it can be used for drawing operations
- Increased memory usage

Separate shader objects address these issues. There is now no need for the expensive process of constructing a program object and linking a number of shader objects. Instead, each shader object can be made into a shader program. Shader programs (one per shader stage) can then be plugged into a new object called pipeline object. Once a pipeline object is bound to the rendering context, it is used for all draw calls, provided no other program object has been made active.

Before a shader program can be used in a pipeline object, it needs to be linked. This process is similar to linking a program object, but is limited to that specific shader stage. Once a shader program is linked, the binary representation can also be saved to reuse it next time the application is launched.

Using shader programs, the application can now build pipeline objects on-the-fly. This is much more efficient than the old method using program objects for the following reasons:

- Pipeline objects do not need to be linked; plug the shader programs into a pipeline object and then start issuing draw calls.
- It is common for an application to use many program objects that all have the same vertex shader in common and are differentiated by the fragment shader. With separate shader objects, the shader program for the vertex shader needs to be built only once. The same shader program can then be reused in a pipeline object as many times as needed. This saves time the linker would otherwise have to spend analyzing the same vertex shader repeatedly.

Shader programs hold uniform state information. If using the separate shader objects approach, configure uniform values separately for each shader program. This is not the case when using program objects. The uniform state is stored in a persistent fashion, so there is no worry about that information getting lost when switching to a different pipeline object. Though, if reusing the same vertex shader program between multiple pipeline objects, remember to update all uniforms that need to take different values for different pipeline configurations.

4.1.8 Shader storage buffer objects

In a previous chapter Uniform Buffers were discussed. These were one of the more important features introduced in OpenGL ES 3.0. However, they still have two unfortunate constraints:

- The maximum size, in a worst-case scenario, can be as low as 16 KB. If multiplying the number by the worst-case maximum number of uniform blocks available for use in a single shader stage (which is 12), this gives a total of 192KB. This is usually enough, but it does require that the data can be split up, which is not always practicable.
- Uniform buffers are read-only.

SSBOs, introduced in OpenGL ES 3.1, address both problems. They are guaranteed to support data blocks of size up to 227 bytes (134,217,728 bytes), and they can be used for both read and write operations. The actual maximum data block size depends on the OpenGL ES implementation and may exceed the above value. This can be checked by calling glGetInteger64v with the pname parameter set to GL_MAX_SHADER_STORAGE_BLOCK_SIZE.

All OpenGL ES 3.1 implementations must support SSBOs at the compute shader stage. Support at other shader stages is optional.

The SSBO equivalent of a uniform block is called a shader storage block, with the following key differences:

- In OpenGL ES Shading Language code, the uniform keyword is replaced by buffer.
- The last member of a shader storage block is allowed to be an unsized array. The size of the array is calculated at run time, in such a way that it makes full use of the actual size of the data store backing the shader storage block.
- On Adreno-based platforms, it is significantly faster to access a uniform block than a shader storage block.

Shader invocations may modify the contents of a shader storage block at any time. Pay attention to the synchronization of memory accesses. Section 4.1.4 discusses a range of memory qualifier keywords. These keywords can provide valuable hints to OpenGL ES implementation as to how the variables are going to be used. The same keywords may be used in the declaration of a shader storage block variable. One use case where this will be absolutely necessary is where the data is going to be reused between different shader stages executed for a single draw call. Under some circumstances, the use of memory barriers is needed to ensure no thread races occur. These barriers can operate either in OpenGL ES Shader Language code or at the level of the OpenGL ES API. Further details can be found here:

- OpenGL ES 3.1 Specification – http://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf
- OpenGL ES 3.1 Reference Pages for glMemoryBarrier – http://www.khronos.org/opengles/sdk/docs/man31/html/glMemoryBarrier.xhtml and memoryBarrier – http://www.khronos.org/opengles/sdk/docs/man31/html/memoryBarrier.xhtml
- OpenGL ES 3.1 Shader Language Specification (see Section 4.9, Memory Access Qualifiers) – http://www.khronos.org/registry/gles/specs/3.1/GLSL_ES_Specification_3.10.pdf

In some cases, it may be possible to avoid the need for using the synchronization techniques described above. As an alternative, use an atomic function. Atomic functions are a set of functions, introduced in the OpenGL ES 3.1 Shading Language, which apply a number of different atomic operations to buffer or shared variables of signed or unsigned integer types.

The following functions are provided:

- atomicAdd – Adds two values together
- atomicAnd – Performs a bit-wise AND operation on two values
- atomicCompSwap – Assigns a value to a variable, if the existing value of the variable contents differs from the value provided by the caller in another argument
- atomicExchange – Sets a variable to a new value and returns the original value
- atomicMax – Returns the maximum of two values
- atomicMin – Returns the minimum of two values
- atomicOr – Performs a bit-wise OR operation on two values
- atomicXor – Performs a bit-wise XOR operation on two values

Here are a few example use cases for shader storage buffer objects:

- Are the main way for compute shaders to exchange data with the outside world
- Can be used by shaders to access large datasets which would have been too big to fit into a texture, e.g., the vertex pulling technique

4.2 Walkthrough of sample applications

This section discusses three sample applications. Each of the samples showcases one of the new OpenGL ES 3.1 features covered in the previous section of this chapter.

4.2.1 Separate shader objects – Demo

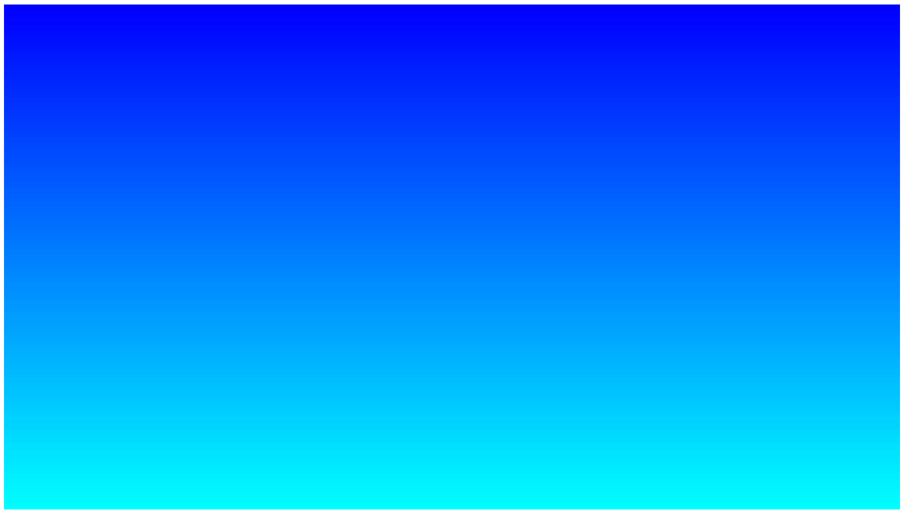

Figure 4-1 Separate shader objects demo

This application demonstrates how to use separate shader objects. It uses a single vertex shader and one of three different fragment shaders to build a pipeline object. Every five seconds, the application switches the fragment shader used by the pipeline object. Observe that the change of shader is virtually instantaneous and does not introduce any lag into the rendering process.

The shaders used are reasonably straightforward.

The vertex shader:

```
#version 310 es
out gl_PerVertex {
vec4 gl_Position; };
layout(location = 0) out vec2 uv;
void main() {   switch
(gl_VertexID)   {
case 0:
      gl_Position = vec4(-1.0, -1.0, 0.0, 1.0);
      uv          = vec2( 0.0,  1.0);
      break;
    case 1:
      gl_Position = vec4(-1.0, 1.0, 0.0, 1.0);
uv          = vec2( 0.0, 0.0);
      break;
    case 2:
      gl_Position = vec4(1.0, -1.0, 0.0, 1.0);
uv          = vec2(1.0,  1.0);
      break;
    case 3:
      gl_Position = vec4(1.0, 1.0, 0.0, 1.0);
uv          = vec2(1.0, 0.0);
      break;
}
};
```

This vertex shader outputs a full-screen quad built out of a triangle strip. It does not take any input data but configures two output variables:

- The vertex position is set to one of four predefined locations, depending on the gl_VertexID value for the running shader invocation. The vertex positions are defined so that a triangle strip using the four vertices will form a full-screen quad.
- The UV coordinates are also passed down the rendering pipeline. This vector is used to construct the gradient in the fragment shader stage.

The fragment shaders generate gradients, using the information prepared in the vertex shader stage. The first fragment shader generates a horizontal gradient, the second one creates a vertical gradient, and the third one shows the result of summing both gradients.

The first of the fragment shaders:

```
#version 310 es
layout(location = 0) in vec2 uv;
out vec4 result;
void main() {   result = vec4(uv.x,
0.0, 1.0, 1.0); };
```

Now look at how the demo:

- Initializes the shader programs
- Sets up the pipeline object
- Renders each frame

4.2.1.1 Setting up the shader programs

The function_create_separate_shader_program is used to set up a shader program. This function is called four times in all, to set up three fragment shader programs and one vertex shader program. This happens when rendering the first frame.

There are two different methods to set up a shader program in OpenGL ES:

- A shader program can provide implementations of multiple shader stages. While there might seem to be little advantage in doing this where only fragment and vertex shaders are involved, it makes more sense when geometry, tessellation control, and tessellation evaluation shader stages come into the equation.

  The first step is to set up a shader object for each shader stage to be included in the shader program. Once all of the shaders have been successfully compiled, create the shader program.

  Setting up a shader program works just like setting up a regular program object, except for one thing: before linking it, set its GL_PROGRAM_SEPARABLE property to GL_TRUE. This is done using a new API function called glProgramParameteri. Then, attach the shader objects and link the program object using glLinkProgram. If the above steps completed without error, then the shader program is ready for use.

- If the shader program is to provide the implementation of a single shader stage only, take a shortcut and use a single call to glCreateShaderProgramv. This carries out all of the steps described for the first method, and returns the ID of the shader program.

  Before attempting to use the object, verify that the shaders compiled successfully and that the shader program was correctly linked. Since there is no access to shader objects for the compilation, check for successful compilation by verifying the GL_LINK_STATUS property of the returned program object.

NOTE: The shader info log for any shader failing to compile will be appended to the program info log. This information may be helpful for diagnosing the error. It can be retrieved using the API entry point glGetProgramInfoLog.

An example from the demo application:

```
bool   result    = true;
GLuint result_id = 0;
result_id = glCreateShaderProgramv(shader_type,
1,
&shader_body); if (result_id != 0) {     GLint
link_status = GL_FALSE;
    glGetProgramiv(result_id,
GL_LINK_STATUS,
&link_status);
```

```
    if (link_status != GL_TRUE)
    {         result =
false;     } } else {
result = false; }
if (!result && result_id != 0)
{
glDeleteProgram(result_id);
    result_id = 0;
}
return result_id;
```

The example application uses the shortcut approach. The implementation calls glCreateShaderProgramv and then checks if a valid program object ID was returned. This ID is then used to check the link status to determine if the shader program has been initialized successfully.

If an error arises at any stage after the program object has been created, release it by calling glDeleteProgram.

4.2.1.2 Setting up the pipeline object

After setting up the shader programs, initialize the pipeline object. The first step is to generate a pipeline object ID using a call to glGenProgramPipelines. Associate the ID with a pipeline object instance using glBindProgramPipeline.

Here is the code from the demo application:

```
glGenProgramPipelines(1, &_pipeline_object_id);
glBindProgramPipeline(_pipeline_object_id);
```

The new pipeline object is now bound to the rendering context, but it does not yet define any shader stages. If trying to issue a draw call while an uninitialized pipeline object is bound, the results are undefined.

NOTE: The pipeline object will only be used by OpenGL ES if there is no other program object activated for this rendering context.

In the example application, the same vertex shader program is always used, but it will be switching between different fragment shader programs every five seconds. It therefore makes sense to set up the vertex shader stage during initialization, as follows:

```
glUseProgramStages(_pipeline_object_id,
GL_VERTEX_SHADER_BIT,
                   _vs_id);
```

The shader program for the vertex shader stage has ID _vs_id. The above call attaches this shader program to the pipeline object.

4.2.1.3 Using the pipeline object

At this stage the pipeline object is set up and is bound to the rendering context. However, it defines a vertex shader stage only. No fragment shader stage is present. This means that any attempt to use the pipeline object for a draw call will result in undefined behavior.

The configuration of the fragment shader stage is done in a code block that will be executed under two conditions:

- If rendering the very first frame
- If at least five seconds have passed since the last time this code block was entered

The code in this block determines which fragment shader stage should be used, based on elapsed time since the application started running. The IDs of the shader programs are held in the array fs_ids, and the variable n_fs_id_to_use is the array index of the shader program that is being used.

Configure the fragment shader stage of the pipeline object using the following call:

```
glUseProgramStages(_pipeline_object_id,
GL_FRAGMENT_SHADER_BIT,
fs_ids[n_fs_id_to_use]);
```

Now that the pipeline object has both shader stages configured, safely issue a draw call that covers whole screen space with a quad built of two triangles:

```
glDrawArrays(GL_TRIANGLE_STRIP,
0,  /* first */
             4); /* count */
```

4.2.2 Multisample textures – Demo

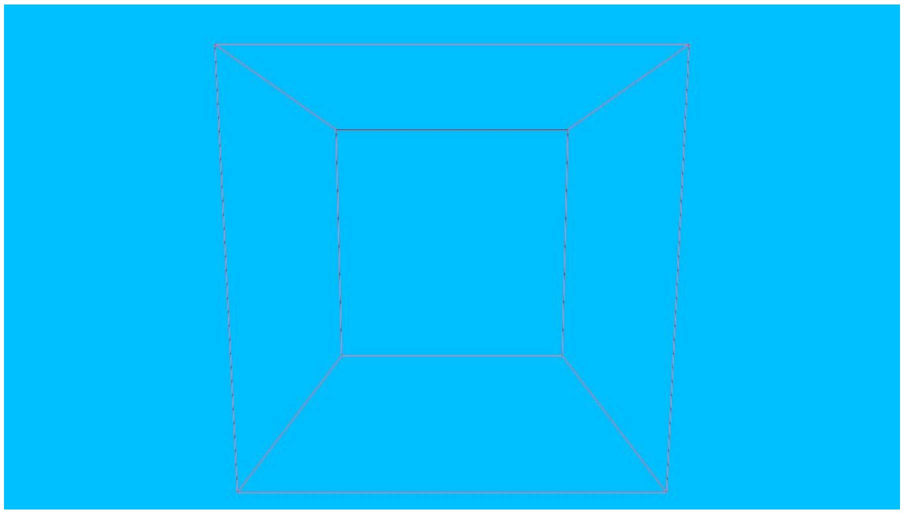

Figure 4-2 Multisample textures demo

The demo shows a rotating wireframe cube which is first rendered to a multisample texture, and then blitted into the back buffer. A number of different multisample textures are created, each holding a different number of samples per texel. The user can switch between these textures in order to see the difference in terms of visual quality.

As with the previous example, the shaders being used in this demo are simple:

- The vertex shader takes a model-view-projection matrix as input and uses it to compute the clip space coordinates for the input vertex position
- The fragment shader sets the one and only output variable to a fully opaque red color

Since this demo is working with a real mesh, much of the code is dedicated to the vertex buffer object setup process. However, it focuses solely on aspects related specifically to multisample textures:

- How are the multisample textures set up?
- How is geometry rendered into a multisample texture?
- How is multisample texture data copied to the back buffer?

4.2.2.1 Setting up the multisample textures

This demo takes the same approach to resource initialization as did the previous demo. The multisample textures are generated and assigned storage in the function responsible for rendering a single frame. This only happens when the function is called for the first time.

The following code block is responsible for setting up the textures:

```
for (n_texture = 0;      n_texture <
n_multisample_textures    ++n_texture)
{     uint32_t n_texture_samples =
n_max_color_texture_samples *
n_texture                  /
(n_multisample_textures - 1);
    if (n_texture_samples == 0)     {         /*
Requesting zero samples is not permitted */
n_texture_samples = 1;     }
    _textures[n_texture].n_samples = n_texture_samples;
    glGenTextures(1,
&_textures[n_texture].texture);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE,
_textures[n_texture].texture);
    glTexStorage2DMultisample(
GL_TEXTURE_2D_MULTISAMPLE,
n_texture_samples,         GL_RGBA8,
rendertarget_width,
rendertarget_height,         GL_FALSE); /*
fixedsamplelocations */
}
```

The demo uses a number of multisample textures, each holding a different number of samples. The total number of multisample textures used is defined by n_multisample_textures.

For each multisample texture, start by working out the number of samples it will use, n_texture_samples. Do this in such a way that the first multisample texture will use a single sample, and the last one will use the maximum number permitted by the implementation (GL_MAX_COLOR_TEXTURE_SAMPLES) , with a reasonably regular progression in between.

Then set up each multisample texture using the following steps:

1. Generate a new texture object ID using glGenTextures
2. Use glBindTexture to bind this ID to the GL_TEXTURE_2D_MULTISAMPLE texture target
3. Allocate storage for the multisample texture using glTexStorage2DMultisample

4.2.2.2 Using the multisample texture as a render target

The demo renders the wireframe cube to a multisample texture attached as a color attachment of a frame buffer object created at initialization time. The multisample texture rendered to is switched every five seconds, using the pool of multisample textures covered earlier.

Use the API function glFramebufferTexture2D to attach the new multisample texture, as follows:

```
glFramebufferTexture2D(
GL_DRAW_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D_MULTISAMPLE,
    _textures[n_texture_to_use].texture,
    0); /* level */
```

Since multisample textures do not support mipmaps, always use the base-level mipmap.

4.2.2.3 Transferring multisample texture data to the back buffer

To transfer the contents of the multisample texture to the back buffer of the default frame buffer, perform a frame buffer blit operation.This operation was introduced in OpenGL ES 3.0 and carries out a fast copy from the attachments of the read frame buffer to the corresponding attachments of the draw frame buffer. These copies bypass the fragment pipeline except that they are still subject to processing by the pixel ownership test, the scissor test and sRGB conversion.

One of the things blits allows is to flatten the contents of a multisample attachment (such as a renderbuffer or a multisample texture) to a single-sample representation. The demo leverages this functionality to merge the multisample representation of the wireframe cube into a single-sampled version, delivered directly to the back buffer. The following code snippet performs this task:

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER,
0);
glBlitFramebuffer(0, /* srcX0 */
0, /* srcY0 */
rendertarget_width,
rendertarget_height,
0, /* dstX0 */                    0, /*
dstY0 */
rendertarget_width,
rendertarget_height,
GL_COLOR_BUFFER_BIT,
GL_NEAREST); /* filter */
```

NOTE: If wondering if it might be simpler just to render a textured quad directly to the back buffer, sampling the multisample texture in a fragment shader and storing the sampled value in an output variable. The problem is that there is only one texture sampling function in OpenGL ES Shader Language that allows the sampling of multisample textures, and this function only samples from a single sample at a time, specified by one of the input parameters to the function. Of course, a shader could sample all the available samples one after another and then calculate a weighted average. However, that would be much slower than the blit operation used in the demo.

Tip

glBlitFramebuffer can also be used to perform stretch blits. For further details, see the OpenGL ES 3.1 Reference Page for the function at http://www.khronos.org/opengles/sdk/docs/man3/html/glBlitFramebuffer.xhtml.

At this point, the single-sampled representation of the wireframe cube has made it into the back buffer. The only thing left to do is to swap the back buffer and front buffer, and the cube will then be made visible.

4.2.3 Compute shaders and shader image load/store – Demo

Figure 4-3 Compute shaders and shader image load/store demo

This demo shows how to use a compute shader to populate a 2D texture and then to render it into the back buffer. The texture contents are calculated using a function of time, resulting in an animation effect.

Since the use of output variables is not available to compute shaders, the shader invocations store the results of their work using images.

In the walkthrough of this demo, the following questions are reviewed:

- How does the compute shader fill a texture mipmap?
- How is the local work group size for the compute shader calculated?
- How is an image setup?
- How is each frame rendered?

4.2.3.1 Using a compute shader to fill a texture mipmap

The demo uses the compute shader to fill a texture mipmap with contents. Each shader invocation is assigned a unique texel location for which it calculates a result value. The result value is stored using the OpenGL ES Shader Language function imageStore.

The shader begins with the following two declarations:

```
layout(local_size_x = LOCAL_SIZE_X,
local_size_y = LOCAL_SIZE_Y) in;
layout(rgba8) uniform restrict writeonly image2D image;
```

Work gets distributed to multiple shader invocations the following way. When dispatching a compute request, pass three arguments to the glDispatchCompute* entry point. These describe the size of an abstract three-dimensional grid known as a global work group. Multiply these three values together to get the total number of local work groups that must be executed before the task can be considered finished.

Depending on the capabilities of the hardware, some of these work groups may be executed in parallel, but this is not necessarily the case. It is equally possible that they may be executed sequentially, especially on lower-end architectures. Moreover, the order of work group execution is undefined. This means that communication between one work group and another is not possible.

Each local work group is in turn made up of yet another three-dimensional grid. A single cell of that grid corresponds to a single work item which is in effect a single compute shader invocation. All work items in a work group execute in parallel and they are allowed to exchange data and to synchronize execution flow with each other.

Looking at the code snippet above, the first layout declaration describes the local work group size. Rather than actual numbers, the demo uses predefined LOCAL_SIZE_X and LOCAL_SIZE_Y tokens, which are replaced with actual numbers at run time. The reason for not using hard coded values in the shader is that the maximum local work group size is implementation dependent, and it is necessary to maximize the number of work items running in parallel, for performance reasons. A detailed description of how to work out the work group size is in the nect section.

NOTE: The demo does not declare local_size_z, as it will default to 1 anyway. This means it will effectively be working with a two-dimensional array, which is what is need.

Referring back to the code snippet again, the second declaration describes an image uniform. Here is a description of the meaning of each keyword:

- Layout(rgba8) – Tells the compiler that the texture referred to by the image uses the data layout GL_RGBA8
- Restrict – Asserts that this uniform is the only means by which the memory region represented by the image will be accessed; the aim is to allow the driver to optimize access to the uniform (e.g., using caching)
- Writeonly – Indicates that the image will only be used for writing; this is an optimization hint
- Image2D – A new opaque OpenGL ES Shader Language type, representing an image unit, to which the mipmap of a two-dimensional texture object has been bound

Following these two uniform declarations is the implementation of the main function:

```
void main() {     const uvec2 image_size       =
imageSize(image);     const  vec2 image_size_float =
vec2(image_size);
    vec4 result = vec4(         abs(cos(time +
float(gl_GlobalInvocationID.x) /
image_size_float.x)),         abs(sin(time +
float(gl_GlobalInvocationID.y) /
image_size_float.y)),         abs(cos(sqrt(time))),
1.0);
    imageStore(image,
ivec2(gl_GlobalInvocationID.xy),
result); }
```

First, retrieve the resolution of the texture bound to the image unit pointed to by the image uniform. This information is needed to calculate the result value.

Then, calculate the result value.

NOTE: The time uniform is being used here for the first three components to animate the visual output.

Finally, store the result value in the image. Each shader invocation will write to a different location within the image. This is done using the uvec3 variable gl_GlobalInvocationID, which represents the coordinates of the current work item within the overall 3D array making up the global work group.

The meaning of gl_GlobalInvocationID is defined in the OpenGL ES Shading Language Specification as follows:

```
gl_GlobalInvocationID =
   gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;
```

In this way, by scheduling a sufficient number of work groups in both dimensions, it is possible to associate each texel of the texture with a work item.

NOTE: If the resolution of the texture is not an exact multiple of the local work group size in either dimension, then try to use the imageStore function to access texels outside the bounds of the texture. Fortunately, such operations will be recognized by GPU as invalid and will be ignored.

4.2.3.2 Determining local work group size

The local work group size needs to be chosen carefully, taking into account the capabilities of the platform. The size of a work group directly corresponds to the size of a region of the texture that it will process. Once that region size is known, it is easy to work out how many work groups are needed in each dimension to fully cover the texture.

The following piece of code, taken from the function _create_compute_shader, is responsible for calculating these values:

```
local_size_x = rendertarget_width;
local_size_y = rendertarget_height;
if (local_size_x > max_compute_work_group_size[0])
{     local_size_x =
max_compute_work_group_size[0]; }
if (local_size_y > max_compute_work_group_size[1])
{     local_size_y =
max_compute_work_group_size[1]; }
if (local_size_x * local_size_y >
max_compute_work_group_invocations) {
local_size_y =
max_compute_work_group_invocations /
local_size_x;
    if (local_size_x > max_compute_work_group_invocations)     {
local_size_x =
max_compute_work_group_invocations;     }
} /* Now that the local work-group size is known, determine how many global
work
_global_workgroup_size[0] =   (GLuint)
(float(rendertarget_width) /
float(local_size_x)       + 0.5f);
_global_workgroup_size[1] =     (GLuint)
(float(rendertarget_height) /
float(local_size_y)        + 0.5f);
```

Observe the following limitations affecting local and global work-group sizes:

- The enum GL_MAX_COMPUTE_WORK_GROUP_SIZE may be used to query the maximum local group size in each of the three dimensions. The maximum size is guaranteed to be at least 128 in the case of the x and y dimensions, and 64 in the case of z.

- The total number of compute shader invocations in a single local work group (i.e. the product of all three values used to describe the local work group size) must not exceed the value reported for GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS. This limit is guaranteed to be no lower than 128.
- The enum GL_MAX_COMPUTE_WORK_GROUP_COUNT may be used to query the maximum number of work groups that may be dispatched by a single command, in each of the three dimensions. It is guaranteed that the limit will not be less than 65535 in any of the three dimensions.

In the above code snippet, it starts by setting the local work group size to the resolution of the texture. Then it clamps width and height values to the maximum work group size supported by the implementation for the corresponding dimension (GL_MAX_COMPUTE_WORK_GROUP_SIZE limit).

The local work group size may still be too large at this point. Check that the GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS limit is not exceeded. Additional clamping is applied first to the Y component, and then to the X component of the local work group size, as necessary.

Now that the local work group size is known, divide each texture dimension by the corresponding local work group size to get the work group count for that dimension.

NOTE: Take a ceiling of the resulting value before converting from floating-point to integer. This is necessary to ensure that texels located very close to the border will not get truncated as a side effect of the division.

4.2.3.3 Setting up an image

So far it has been discussed how the demo uses compute shaders. This section will take a closer look at how the image unit and the image uniform are configured. Recall that these are used by the shader invocations to store their result values.

As with the previous demos, resources are initialized on the first pass through the function responsible for rendering a frame, _rendering_handler. Here is how it sets up the image uniform:

```
glGenTextures(1, &_texture);
glBindTexture (GL_TEXTURE_2D, _texture);
glTexStorage2D(GL_TEXTURE_2D,
1, /* levels */                GL_RGBA8,
rendertarget_width,
rendertarget_height);
/* Set up image bindings */
glBindImageTexture(0,        /* index */
_texture,
                   0,        /* level */
                   GL_FALSE, /* layered */
                   0,        /* layer */
                   GL_WRITE_ONLY,
                   GL_RGBA8);
/* Point the image uniform at the new binding */
```

```
glUseProgram(_program); glUniform1i
(_program_image_uniform_location,
0); /* unit */
```

Begin by setting up an immutable texture to hold the data that will be generated by the compute shader. Only the base mipmap level is needed. This will be used as a read buffer for a blit operation.

Then, call glBindImageTexture to bind the texture object to an image unit at index 0. Images work in a very similar way to texture objects. With textures, before sampling a mipmap in the shader, bind the texture object to one of the texture units and then set a sampler uniform to point to that texture unit. With images, the process is analogous and glBindImageTexture plays a role equivalent to glBindTexture. The only difference is that glBindImageTexture needs to provide a bit more detail about the mipmap—or set of mipmaps, in the case of layered texture targets such as 2D array textures—and to provide hints about the intended way to use the image.

The final part of the code snippet above activates a program object and configures the image uniform to point to texture image unit 0—the same texture unit that is used for the glBindImageTexture call.

NOTE: The call here to glUniform1i is not strictly necessary, because 0 is the default setting for all image uniforms in OpenGL ES Shader Language. However, it is good practice to include it, since forgetting to configure the texture sampler or image uniform is a frequent programming mistake.

4.2.3.4 Rendering a frame

Once all the OpenGL ES objects have been initialized, render the contents of a frame. First, assign a new time value to one of the compute shader uniforms. Then, dispatch the compute job:

```
glUseProgram(_program); glUniform1f

(_program_time_uniform_location,
float(current_time_msec) / 1000.0f);
glDispatchCompute(_global_workgroup_size[0],
_global_workgroup_size[1],
                  1); /* num_groups_z */
```

It is important to remember that OpenGL ES works in an asynchronous fashion. This is especially important in the context of images, which allow shaders to modify external entities such as texture data storage. OpenGL ES does not automatically synchronize such shader updates with other accesses that might be made to the same objects (for OpenGL ES to do this would introduce a large overhead that could significantly affect performance). Therefore, use appropriate synchronization methods when accessing any kind of storage that could have been modified from within a shader.

In the case of the demo, it is intented to use the results of the compute job in a frame buffer blit operation. The blit is the very next step in the code, after launching the compute job. Without proper synchronization, the blit operation may attempt to read from a buffer before the completion of the updates to the buffer from the compute shader.In order to serialize these operations, issue the following call:

```
glMemoryBarrier(GL_FRAMEBUFFER_BARRIER_BIT);
```

The memory barrier ensures that OpenGL ES will flush any caches that the hardware may have used for the image store operations, if these could affect subsequent frame buffer operations.

The final step is to issue a frame buffer blit to transfer the texture contents to the back buffer, as was done in the previous demo:

```
glBlitFramebuffer(0, /* srcX0 */
0, /* srcY0 */
rendertarget_width,
rendertarget_height,
0, /* dstX0 */                0, /*
dstY0 */
rendertarget_width,
rendertarget_height,
GL_COLOR_BUFFER_BIT,
                  GL_NEAREST); /* filter */
```

4.3 About the OpenGL ES implementation

This section gives details of the capabilities of the Adreno architecture within an OpenGL ES 3.1 context.

4.3.1 GL constant values

Table 4-1 GL_MAX constant values – OpenGL ES 3.1

Pname	Adreno 330	Adreno 420
GL_MAX_3D_TEXTURE_SIZE	8192	16384
GL_MAX_ARRAY_TEXTURE_LAYERS	512	2048
GL_MAX_COLOR_ATTACHMENTS	4	8
GL_MAX_COMBINED_ATOMIC_COUNTERS	0	48
GL_MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS	0	230400
GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	230272	230272
GL_MAX_COMBINED_SHADER_STORAGE_BLOCKS	0	24
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS	32	96
GL_MAX_COMBINED_UNIFORM_BLOCKS	28	84
GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	230400	230400
GL_MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS	0	8
GL_MAX_COMPUTE_ATOMIC_COUNTERS	0	8
GL_MAX_COMPUTE_SHADER_STORAGE_BLOCKS	0	24
GL_MAX_COMPUTE_TEXTURE_IMAGE_UNITS	0	16
GL_MAX_COMPUTE_UNIFORM_BLOCKS	0	14
GL_MAX_COMPUTE_UNIFORM_COMPONENTS	0	1024
GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS	0	1024
GL_MAX_COMPUTE_WORK_GROUP_COUNT	0	65535

Pname	Adreno 330	Adreno 420
GL_MAX_COMPUTE_WORK_GROUP_SIZE	0	1024
GL_MAX_CUBE_MAP_TEXTURE_SIZE	8192	16384
GL_MAX_DRAW_BUFFERS	4	8
GL_MAX_ELEMENT_INDEX	2147483648	2147483648
GL_MAX_ELEMENTS_INDICES	2147483648	2147483648
GL_MAX_ELEMENTS_VERTICES	2147483648	2147483648
GL_MAX_FRAGMENT_ATOMIC_COUNTERS	0	8
GL_MAX_FRAGMENT_INPUT_COMPONENTS	64	128
GL_MAX_FRAGMENT_SHADER_STORAGE_BLOCKS	0	4
GL_MAX_FRAGMENT_UNIFORM_BLOCKS	14	14
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS	896	896
GL_MAX_FRAGMENT_UNIFORM_VECTORS	224	224
GL_MAX_FRAMEBUFFER_HEIGHT	8192	16384
GL_MAX_FRAMEBUFFER_SAMPLES	4	4
GL_MAX_FRAMEBUFFER_WIDTH	8192	16384
GL_MAX_INTEGER_SAMPLES	1	1
GL_MAX_PROGRAM_TEXEL_OFFSET	7	7
GL_MAX_RENDERBUFFER_SIZE	4096	16384
GL_MAX_SAMPLE_MASK_WORDS	1	1
GL_MAX_SAMPLES	4	4
GL_MAX_SERVER_WAIT_TIMEOUT	4294967296000000 ns	4294967295000000 ns
GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS	0	24
GL_MAX_TEXTURE_IMAGE_UNITS	16	16
GL_MAX_TEXTURE_LOD_BIAS	15.984375	15.996094
GL_MAX_TEXTURE_SIZE	8192	16384
GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	64	128
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	4	4
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	4	4
GL_MAX_UNIFORM_BLOCK_SIZE	65536	65536
GL_MAX_UNIFORM_BUFFER_BINDINGS	28	84
GL_MAX_VARYING_COMPONENTS	64	128
GL_MAX_VARYING_VECTORS	16	32
GL_MAX_VERTEX_ATOMIC_COUNTERS	0	8
GL_MAX_VERTEX_ATTRIB_BINDINGS	16	32
GL_MAX_VERTEX_ATTRIB_RELATIVE_OFFSET	2147483648	2147483648
GL_MAX_VERTEX_ATTRIBS	16	32
GL_MAX_VERTEX_OUTPUT_COMPONENTS	64	128
GL_MAX_VERTEX_SHADER_STORAGE_BLOCKS	0	4
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS	16	16
GL_MAX_VERTEX_UNIFORM_BLOCKS	14	14
GL_MAX_VERTEX_UNIFORM_COMPONENTS	1024	1024
GL_MAX_VERTEX_UNIFORM_VECTORS	256	256

Pname	Adreno 330	Adreno 420
GL_MAX_VIEWPORT_DIMS	8192	16384

Table 4-2 Other GL constant values – OpenGL ES 3.1

Pname	Adreno 330	Adreno 420
GL_ALIASED_LINE_WIDTH_RANGE	1	1
GL_ALIASED_POINT_SIZE_RANGE	1	1
GL_MIN_PROGRAM_TEXEL_OFFSET	-8	-8
GL_SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT	0	4
GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT	32	32

5 Introducing Android Extension Pack

5.1 Overview

The AEP is provided as part of the Adreno SDK. It contains a set of OpenGL ES 3.1 extensions.

Read this chapter for a quick introduction to the concepts behind these extensions and the advantages they could bring to an application. Look for the links to access more detailed information.

5.2 Geometry shaders

Geometry shaders are a new shader type and are optionally inserted into the rendering pipeline immediately ahead of the of Fragment Shader stage.

The geometry input to the geometry shader can be the following:

- Points
- Lines
- Lines with adjacency data
- Triangles
- Triangles with adjacency data

These inputs are used to generate output that can be:

- Points
- Line strips
- Triangle strips

The original geometry is discarded.

Geometry shaders can be used for layered rendering, in which primitives can be rasterized to multiple separate images held within a single texture object. Examples of rendering destinations that can be used in this way include:

- 3D texture slices
- Layers of 2D array textures
- Cube-map texture faces

Geometry shaders can be used for debugging, e.g., for the visualization of normal data. They can also be used for billboarding and for Bokeh effect rendering.

For more information about geometry shaders, see:

- GL_EXT_geometry_shader extension specification at http://www.khronos.org/registry/gles/extensions/EXT/EXT_geometry_shader.txt
- Khronos Wiki entry for Geometry Shaders at http://www.opengl.org/wiki/Geometry_Shader

NOTE: The Wiki entry includes discussion of functionality available only in desktop OpenGL and not in OpenGL ES.

5.3 Tessellation shaders

The term *tessellation shaders* covers two new shader stages: tessellation control and tessellation evaluation. These may optionally be inserted into the rendering pipeline immediately after the vertex shader stage (and before geometry shaders).

Tessellation can be used to reduce the amount of geometry that needs to be fed into the graphics pipeline. The input into the tessellation process is a set of vertices known as patch vertices. These represent a piece of geometry in a way which is defined by the application. The tessellation control shader stage is responsible for determining the granularity of the tessellation performed by the next stage, the fixed-function tessellator. The tessellator generates the vertices that will make up the output geometry of the process. The tessellation evaluation shader stage then assigns positions and attribute values to these vertices. The input patch vertices are discarded.

Tessellation can be used to process geometry that has been fed through the vertex shader.

NOTE: Geometry generated during tessellation can be further subdivided by the geometry shader.

Common applications of tesselation shaders include the following:

- LoD-driven terrain rendering – Tessellation granularity of a particular patch depends on its distance from the camera
- LoD-driven mesh smoothing – Amount of smoothing applied to the patch depends on how close it is
- Displacement mapping – Dynamically adapt the complexity of displacement for vertices generated by the tessellation process

Find more information about tessellation shaders here, see:

- GL_EXT_tessellation_shader extension specification at http://www.khronos.org/registry/gles/extensions/EXT/EXT_tessellation_shader.txt
- Khronos Wiki entry for Tessellation Shaders at http://www.opengl.org/wiki/Tessellation

NOTE: The Wiki entry includes discussion of functionality available only in desktop OpenGL and not in OpenGL ES.

5.4 Advanced blending modes

A number of new blending modes are available:

- Color burn
- Color dodge
- Darken
- Difference
- Exclusion
- Hard light
- HSL color
- HSL hue
- HSL luminosity
- HSL saturation
- Lighten
- Multiply
- Overlay
- Screen
- Soft light

These new modes make available a number of advanced blending equations that can be used to perform more sophisticated color blending operations than the blend modes in core OpenGL ES.

For further information, see the extension specification at http://www.khronos.org/registry/gles/extensions/KHR/blend_equation_advanced.txt.

NOTE: There are some important limitations that affect when and how these new blending modes can be used. Read the extension specification for details.

5.5 ASTC LDR

ASTC LDR is a new texture compression format. The level of image degradation is in general remarkably low given the compression ratio achieved. The user is able to control the quality/size trade-off by selecting one of a number of different block sizes.

NOTE: The format supports NPOT textures. It can hold data for up to four 8-bit components per texel and supports both linear RGB and sRGB color spaces.

If working with floating point data, ASTC LDR is not suitable—consider using ASTC HDR instead.

For further information, see the extension specification at http://www.khronos.org/registry/gles/extensions/KHR/texture_compression_astc_hdr.txt.

6 Designing OpenGL ES Guidelines

6.1 Overview

Designing a well performing OpenGL ES application can be a challenging task. Every single ES application should try to:

- Adjust its behavior to the capabilities of the underlying hardware
- Choose the fastest and the most appropriate mechanism for the rendering context version in use in the application
- Recognize available extensions and make a good use of them
- Make sure the draw calls are organized in an efficient manner so that:
  - Number of all draw calls that do not rasterize any fragments is reduced to a minimum
  - Meshes produced by the draw calls are sorted from the nearest to the furthest, so that the amount of overdraw is reduced to a minimum
  - Draw calls are first sorted by render target, then by active program object, and finally by any other state adjustments that need to be applied for the affected draw operations to work correctly
- Avoid making redundant API calls that attempt to set an ES state to a particular value, when exactly the same value has already been assigned earlier
- Use multiple rendering contexts if an application needs to compile shaders, or stream geometry or texture data on-the-fly

This chapter discusses these and many more aspects that ES application designers should always keep in mind when creating software.

Further information describes the evolution of Open GL ES and the different approaches that have been taken to address the issue of CPU bottlenecks, which are currently the biggest problems affecting 3D application performance.

6.2 OpenGL ES as a client-server architecture

Fundamentally, OpenGL ES is conceptually based around a client-server architecture, where ES implementation provides rendering services to applications. A single application can open multiple connections by creating a number of ES rendering contexts. Each connection is assigned default state configuration, which can be modified or used (eg. for compute or rendering operations) by sending ES commands through that connection.

On desktop and embedded platforms, this translates to a model where the concept of a server is replaced by a driver which runs in both kernel- and user-mode layers of the operating system. The user-mode part of the driver acts like a shim layer, which intercepts the ES commands from the application and queues them in a command buffer. When the buffer runs out of space or a flushing command is executed, the buffer is transmitted to the kernel-mode part of the driver which then transmits it to the GPU for actual execution.

As with any design, the model has a number of positives and drawbacks. As the OpenGL ES API continues to advance with newer versions and hardware capabilities improve, the API overhead shrinks. An example of this is how generic vertex attribute array management has evolved over time in OpenGL ES to improve performance issues stemming from the original model design.

When issuing a draw call, the vertex shader used for the rendering process usually takes per-vertex attribute data from some form of external storage. In OpenGL ES 1.1 and ES 2.0, there were two options to choose from, as to the data source:

- A process-side buffer maintained by the application.
- Alternatively, assuming the hardware had sufficient amount of free committable memory, the data stored in VRAM (or any other memory region that the driver deemed appropriate, given the usage hints passed when allocating that store). In OpenGL ES 1.1, this functionality was considered a client-side capability, as opposed to ES 2.0 where it was named generic vertex attribute array. Despite the different names, the general idea behind these concepts was the same.

The first data source described in the list above is very expensive. Since all client-side buffers can be freely changed by the application at any time, OpenGL ES implementations must cache contents of all process-side buffer regions required to execute the draw call. Only after the data is cached, can the execution flow be returned to the application. There are two problems with this strategy:

- These regions can take significant amount of space that the driver would need to allocate on-the-fly and then maintain. In embedded environments with limited memory availability, this can be a serious issue.
- A memory copy operation is not free.

The biggest problem with client-side buffers is that the GPU cannot usually access memory regions of the type directly. Instead, it needs to be copied to another memory block that the hardware has direct access to. An important observation to be made at this point is that generic vertex attribute arrays backed by buffer object storage (also known as vertex buffer objects) are not affected by this problem; after all, the data is likely to be already stored in a GPU-friendly memory region.

Modifying the configuration of generic vertex attribute arrays is often a costly activity on its own. Since OpenGL ES 1.1 or ES 2.0 did not include support for any kind of state container that would encapsulate this information, implementations were forced to change the configuration on-the-fly between draw calls, which increased the time required to render a single frame. If an application wanted to use a buffer object-backed storage, it often had to update the GL_ARRAY_BUFFER binding with a glBindBuffer() call before proceeding with an actual draw operation. Therefore, binding operations can be expensive.

Vertex Array Objects have been introduced with OpenGL ES 3.0 to reduce the inefficiency. Since VAOs cache GL_ARRAY_BUFFER bindings that are current at the time each generic vertex attribute array configuration is updated, the only thing applications have to do now when rendering a set of meshes is to keep switching between mesh-specific Vertex Array Objects, prior to issuing a corresponding set of draw calls. This is instead of doing a number of expensive API calls.

OpenGL ES 3.0 discourages developers from using client-side buffers as a backing for generic vertex attribute array data by not allowing the application to use those buffers for any Vertex Array Object other than the default one holding ID 0. The support was not completely dropped to maintain backward compatibility: All OpenGL ES 2.0 applications must work flawlessly in OpenGL ES 3.0 contexts, and that requirement would have not been met had client-side buffer support been removed for the default Vertex Array Object.

OpenGL ES 3.0 also introduced a handful of new mechanisms which let applications order the GPU to directly write to VRAM-backed buffers or to have it treat buffer object regions as a source for texture data. For more information, see OpenGL ES 3.0 Specification at http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.3.pdf.

- Pixel Buffer Objects – Bind a buffer object to a GL_PIXEL_PACK_BUFFER binding point, so that glReadPixels() calls will write the result data to the specified buffer object region. By associating a buffer object with GL_PIXEL_UNPACK_BUFFER binding point, any calls that take texture data (like glCompressedTex*Image() and glTex*Image()) will use that memory region as a source, instead of client-side buffers.
- Transform Feed-back – Data saved by vertex shader to any of the output variables can be stored in a buffer object region.

These mechanisms steer OpenGL ES in a direction where the GPU generates the data it then reuses for different purposes. By avoiding CPU intervention, the API overhead is further reduced. Open GL ES 3.1 introduces even more instruments that can be used to make the GPU more self-sufficient. E.g.:

- Compute Shaders – Leverage the SIMD architecture of the GPU and allow the GPU programs to modify buffer object storage and texture mipmap contents.
- Images – Shaders can now modify buffer object storage whenever they wish
- Indirect Draw Calls – Instanced draw calls can now be fired using arguments stored in buffer object storage

In many cases, by combining these tools, per-frame number of RAM<->VRAM transfers can be greatly reduced, resulting in significantly improved application performance.

6.3 OpenGL ES as a graphics pipeline

OpenGL ES is an asynchronous state machine, as even though the API calls usually execute quickly, it does not mean they have been handled by the GPU. Instead, they were most likely queued for execution which happens at some point in the future.

A pipeline flush, which corresponds to the transmission of all queued commands down to the GPU's Command Buffer, and/or stall, which happens when the driver needs to wait for the hardware to finish processing flushed commands, can occur if any of the API functions described below are called by the application:

- eglSwapBuffers()

- glCheckFramebufferStatus()
- glClientWaitSync() or glWaitSync() calls
- glCompressedTexImage2D() and glCompressedTexImage3D()
- glCompressedTexSubImage2D() and glCompressedTexSubImage3D()
- Any of the glDraw*() draw calls, if any of the enabled vertex attribute arrays are configured to use a client-side buffer region
- Any of the glGet*() getter calls
- Any of the glIs*() getter calls.
- glMapBufferRange(), unless GL_MAP_UNSYCHRONIZED_BIT flag is used
- glReadPixels()
- glTexImage2D() and glTexImage3D()
- glTexSubImage2D() and glTexSubImage3D()
- glBufferSubData()

These functions can introduce lags into the rendering pipeline for the following reasons:

- Any API call that relies on client-side data must be executed imminently. Should the execution be deferred until an undefined moment of time in the future, by the time ES starts handling that call, the buffers available at the time of the call might have had already gone out of scope or have been released.
- Any API call that returns a piece of information to the caller may require the pipeline to be flushed. Had it not been flushed and the values had to be retrieved by querying the hardware, which is an implementation-specific detail, the value or set of values they return could be invalid, as it/they would not represent the latest ES state configuration.
- As per EGL specification, front/back buffer swap operation performs an implicit flush operation. This is to ensure that the frame is rendered as soon as possible, so that it can then be shown on the display.
- Buffer object storage may not be mapped into process space before the queued API instructions finish executing. Otherwise, it could run into a situation where the process reads data storage that is still being written to. If the GL_MAP_UNSYNCHRONIZED_BIT flag is specified, this can be ignored by the driver since the application is aware of the risk.
- Operations that need to access any render-target that has earlier been rendered to (like glReadPixels()) cannot operate on buffers that may still being rendered to. Any draw calls stuck in the message queue must be executed before the process can retrieve the visual data or copy it into a buffer object region.

As described in Section 4.1, OpenGL ES 3.1 also introduced a new tool called images. These allow shader invocations to issue read and write operations that operate directly on the contents of texture mipmap(s). Because shader invocations run asynchronously and in loosely defined order, care must be taken to ensure that any ES operations that follow an image-based draw call and that make use of the possibly modified memory, are correctly synchronized. This can be achieved by calling glMemoryBarrier() or glMemoryBarrierByRegion(). These functions take a single bitfield argument,where each bit corresponds to a certain type of operation that may rely on the modified memory region. More information about these bits can be found in reference pages at http://www.khronos.org/opengles/sdk/docs/man31/.

As a reminder, it is important to batch draw operations. When looking at a single draw call invocation, the majority of time is spent on verifying GL state configuration (see Section 6.4.8 for more information). There are a number of ways to reduce that cost:

- For OpenGL ES 2.0 (and higher), consider using indexed triangle strip-based draw calls, which is often the most efficient way of drawing geometry on Adreno architecture. However, this approach may result in a performance loss, so always carry out performance testing to determine the method to be used.
- For OpenGL ES 3.0 and ES 3.1 applications, use indexed instanced draw calls for all the geometry, whose topology is constant, and whose appearance can be satisfactorily changed by passing different data via attributes. In all other cases, use indexed ranged draw calls: glDrawRangeElements() tells ES what the min/max values for the index data set of the draw call are. This lets OpenGL ES implementation precache the vertex data for the specified index range, which can improve the rendering performance.
- If the application is written with OpenGL ES 3.1 in mind, is is possible to prebake a buffer object storage by filling it with draw call arguments for all the static geometry for a particular scene graph. Once that storage is ready, issue indexed draw calls instead of the regular ones. This will move a large portion of the validation cost to GPU in Adreno architecture, resulting in a significantly improved rendering efficiency.

NOTE: Indirect draw calls are implicitly instanced. This means that if the engine has been written with instanced draw calls in mind, further improvements can be made for the rendering performance by migrating to indirect ones.

These optimizations should always be kept in mind.

6.4 Designing a high-performance OpenGL ES application

6.4.1 Advanced lighting/shadowing

The availability of programmable shaders makes it possible to apply per-pixel lighting everywhere. To use the available fragment shader instructions more efficiently, per-vertex lighting or light maps can be used.

Similarly, specular highlights, cube lighting, and reflections maps that are part of the original texture or prebaked can be used. Assuming the application does not support a time-of-day feature, shadows and lights can be prebaked and loaded as a texture.

6.4.2 Avoid excessive clipping

Consider tessellating any triangles, whose area is significantly larger than the average for a given mesh. This will help avoid excessive clipping, which could become a performance bottleneck.

Consider the following example, where a static room changes from 722 triangles to 785 triangles. More triangles were added to the overall geometry, but the performance improves because there is no excessive clipping of the oversized triangles.

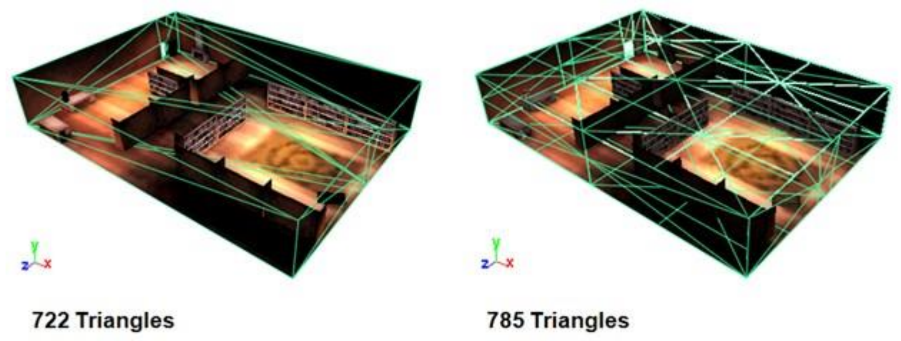

Figure 6-1 Geometry design to avoid clipping

6.4.3 Avoid unnecessary clear operations

The Adreno driver handles glClear() calls by drawing viewport-covering quads into the frame buffer, using clear color values specified earlier by the application. Clear calls operate at the normal pixel rate.

For packed depth+stencil attachments (ones that use GL_DEPTH24_STENCIL8 or GL_DEPTH32F_STENCIL8 internal formats), the Z-buffer and the stencil buffer should always be cleared together. Clearing either depth or stencil buffer for a packed attachment results in reduced performance.

Avoid redundant clears as they are costly.

To avoid unnecessary implicit copy operations, always clear the depth buffer at the start of a frame.

6.4.4 Avoid unnecessary MakeCurrent calls

Calling the eglMakeCurrent function every frame within an application is not recommended.

6.4.5 QTI SIMD/FPU co-processor usage

Snapdragon CPU cores support an additional VeNum a SIMD/FPU co-processor that is instructionally compliant with ARM NEON instruction set. This SIMD/FPU unit can be coded for jobs to offload the calculations from the CPU or vertex shaders, e.g., skinning, simulations, or physics. There is a simulator for testing and development on the PC and for sample assembly code, as well as code that initializes and runs jobs on the coprocessor.

6.4.6 Implement multiple threads

Snapdragon processors today support multiple CPU cores. Distribute jobs to multiple threads in the engine to take advantage of threads that can be physically executed in a concurrent manner.

6.4.7 Implement level-of-shading system

Because pixel shader instructions are scarce on mobile platforms, a so-called level-of-shading system can be implemented, in which an algorithm is scalable, e.g., in three or five levels, depending on the distance from the camera, e.g., a detailed shading may not be necessary on an object until the camera is close to it. When the object is at a distance, a simpler shading can be used.

6.4.8 Minimize GL state changes

ES API calls are enqueued in a command buffer prior to being dispatched to the kernel-mode part of the driver, and then to the GPU. It is therefore important for the application to only issue those API calls that really modify current rendering context configuration.

As an example, even though the following set of calls is not going to do any harm to the rendering process, it is very likely to result in a reduced rendering performance, owing to the fact that frame buffer binding operations are expensive and the glBindFramebuffer() calls presented in the code snippet below are redundant:

```
const GLuint fbo_id = 1;
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo_id);
glDrawArrays     (..);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo_id);
glDrawElements   (..);
```

Texture thrashing is costly too, so always sort the render calls by texture and then by state vector to optimize the performance.

If using ES 3.0 or ES 3.1, utilize sampler objects to separate texture sampling state from texture objects. This saves time the hardware would need to spend configuring texture units separately for each texture touched by shader samplers.

If the application operates on an OpenGL ES 3.1 context, consider using vertex attribute bindings. These split generic vertex attribute array state into two separate entities:

- Array of vertex buffer binding points, where each binding point holds information about:
  - Bound buffer object
  - Start offset
  - Stride
- Array of generic vertex attribute format information records, where each record specifies:
  - Component count
  - Normalization flag status

  - Offset of the first element relative to the start of the vertex buffer binding specific attribute is to fetch from
  - Reference to one of the new vertex buffer binding points described above

Re-using the same vertex buffer binding points for multiple attributes, or modifying the data source of generic vertex attribute arrays by switching to a different vertex buffer binding point, is very likely to be a more lightweight operation than completely redefining generic vertex attribute array configuration with a glVertexAttribPointer() call.

6.4.9 Minimize number of draw calls

OpenGL ES needs to perform a variety of different tasks before it can execute a single draw call, e.g.:

- Frame buffer completeness must be verified
- Incomplete textures must be identified
- Processing of the currently bound Vertex Array Object
- If client-side vertex attribute arrays are enabled, the data needs to be transferred to a GPU-accessible memory region

In OpenGL ES 3.0, always consider using immutable textures. These are always complete, hence the costly verification process can be skipped for them.

In OpenGL ES 3.1, some portion of the validation process can be moved to hardware if an indirect draw call is issued.

Under Android, each call to native code implementation from Java comes with an additional cost that quickly adds up.

For these reasons, it is recommended to keep the number of draw calls issued to a minimum, and to reduce the number of vertices that enter the rendering pipeline. The latter can be achieved by describing the geometry using triangle strips which help to reduce the amount of data that the hardware needs to process. Starting with OpenGL ES 3.0, it is possible to use primitive restart functionality to render disjoint strips with a single draw call.

6.4.10 Scaling resolutions

Different values of the target resolution and frame rate should be tested on actual devices, weighing visual fidelity versus the resulting performance budget. The physical resolution of the display dictates the ideal frame buffer resolution.

Use frame buffer objects to render portions of the scene at a smaller resolution. Then, during a post-processing pass, which may be occurring anyway, the scene can be scaled up to the native resolution of the display. UI and HUD elements can then be rendered at the native resolution.

This approach is not uncommon in console games, in both previous and current generations. It provides the flexibility to choose a lower resolution, since the texture filter hardware smooths out the image.

6.4.11 Avoid intra-frame resolves

Do not use any draw call patterns for the scene that might cause the driver to resolve out GMEM contents prematurely and later restore them. The easiest way to prevent performance killing is to ensure that the frame buffer is cleared immediately after every call to glBindFrameBuffer. Scenes should be drawn as shown in Figure 6-2.

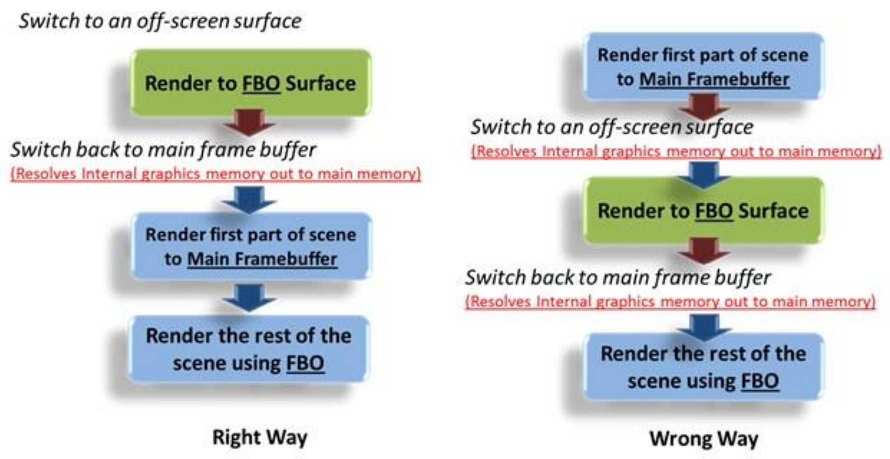
Switch to an off-screen surface
Render to FBO Surface
Switch back to main frame buffer
(Resolves Internal graphics memory out to main memory)
Render first part of scene to Main Framebuffer
Render the rest of the scene using FBO
Right Way
Render first part of scene to Main Framebuffer
Switch to an off-screen surface
(Resolves Internal graphics memory out to main memory)
Render to FBO Surface
Switch back to main frame buffer
(Resolves Internal graphics memory out to main memory)
Render the rest of the scene using FBO
Wrong Way

Figure 6-2 Intra-frame resolves

6.4.12 Identify bottlenecks

To implement advanced visual effects, experimentation and benchmarking are required to assess the target platform. Figure 6-3 shows tests to identify bottlenecks and suggests the order in which to perform the tests. Based on the results of these tests, the platform-specific budget could be estimated and any algorithm requiring a complex shader needs to be adjusted accordingly.

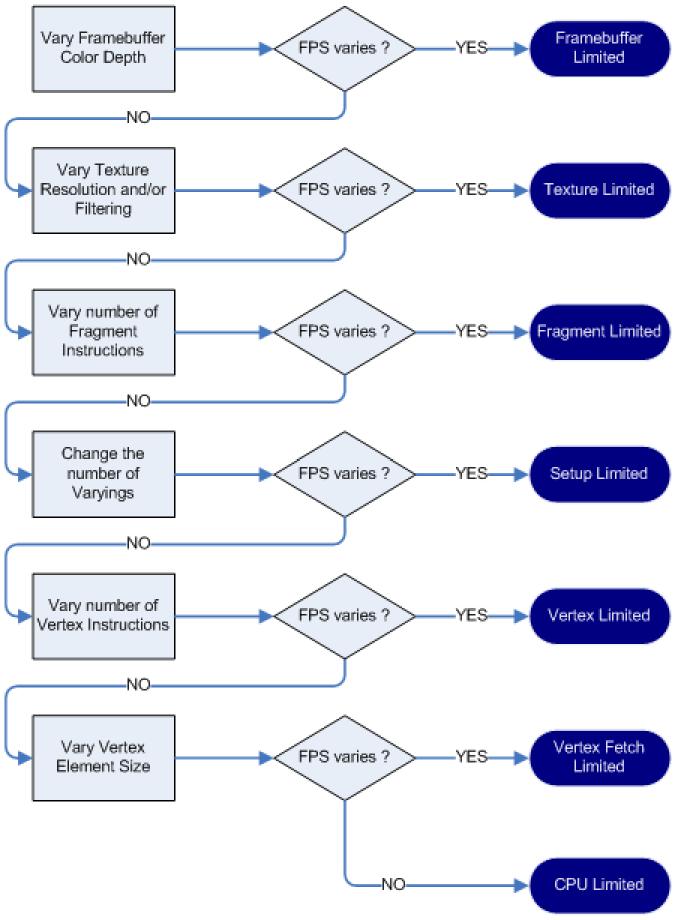

Figure 6-3 Identifying application bottlenecks in GPU

6.4.13 Blur effect

Many advanced shaders, especially post-processing effects and image transitions generally used in UI systems, require blur algorithms.

Typically, separable filter kernels are used. These are based on the algorithms developed by the mathematician Gauss. The algorithm could be heavy to implement into a pixel shader as a whole. Calculating the offsets of those kernels in the vertex shader or on the CPU can make a difference. Vertex shaders can prefetch the texture and arithmetic instructions are reduced in the pixel shader.

Using a lower mipmap level can be an effective alternative. Creating a quarter-size blurred image, then fetching this image to blur a fullsize image, is an old trick commonly used in image space effects.

Another way to achieve a stronger blur effect is to switch on bilinear filtering to blur while fetching an image. On Adreno GPUs, this is slightly more expensive than point sampling the texture.

Any progressive blur, as applied by Masaki Kawase on the Xbox, in which an image is blurred and the sampled image is blurred again, will generate an additional performance hit with each iteration. But multiple iterations are necessary in case of progressive blur to achieve a strong blurred effect.

6.4.14 Store equations in textures/look-up textures

As soon as an algorithm can be refactored to store in a texture, one texture fetch can replace numerous arithmetic instructions. Further, several complex algorithms can be stored in textures and fetched on demand, allowing much greater flexibility in visualizing complex algorithms.

For mobile devices, the number of arithmetic instructions replaced by this approach has to be very high, because texture bandwidth is very limited. A 16:1 ratio is a good rule of thumb here. As long as the texture fetch might replace 16 or more arithmetic instructions, it can be worthwhile.

6.4.15 Use front-to-back rendering

The Adreno GPU contains special circuitry to optimize blocks of pixels that would otherwise fail the depth test. The GPU tries to detect such blocks of pixels early in the pipeline to save further processing.

To assist the GPU, rendering should be as close as possible to front-to-back. Having the CPU sort individual triangles is excessive, but, sorting frustum-culled bounding boxes from the nearest to the furthest is more feasible.

Another example would be a frame consisting of a landscape and a skybox. To avoid overdraw, the application would first draw the land, then the skybox.

6.4.16 Use back face culling

A simple performance mistake is to turn off back face culling. This can cause a worst-case doubling of the number of pixels that need to be shaded. For opaque, nonplanar objects, back faces should always be culled.

NOTE: The FlexRender deferred rendering algorithm obeys the GL cull mode, so back-face culled polygons can improve the vertex processing performance.

6.4.17 Use LoD in effects

When thinking of LoD systems, think of the polygon count of meshes. That is a good practice, and should be employed for mobile applications. Also consider having a LoD system for shader effects. E.g., fade away specular highlights and normal mapping as an object recedes and disable these completely after the distance of the object to the camera exceeds a certain threshold.

6.4.18 Use mipmapping

Not only are mipmaps important for improved visual quality, but they are also necessary for better texture cache performance. With hardware memory constraints in mind, it is recommended to use them whenever feasible.

6.4.19 Use static and baked-in lighting

Dynamic lights, shadow maps, normal mapping and per-pixel effects should be reserved for visually worthy objects in the scene. Spend shader instructions on object that matter and do not waste GPU cycles on meshes that are irrelevant or located far in the back plane. For the latter, cut corners where possible, e.g., disable specular lighting calculations and use baked-in light maps.

6.4.20 Use 2D array textures

2D Array Textures are a unique GPU feature that can assign an array of textures to a single texture stage. The shader can simply use a third texture coordinate to select which texture it wishes to sample from.

Using 2D Array Textures can simplify state changes and increase the batching of draw calls. 2D Array Textures can be used to minimize texture cache thrash for materials that have multiple textures. Each texture layer in the stack can be a separate texture needed by the material. The fragment shaders would have to be written such that the Z coordinate of the UV argument of the texture() is assigned texture layer index number.

6.4.21 Use vertex buffer objects

It is recommended to use Vertex Buffer Objects instead of client-side vertex arrays for the best performance. Relying on client-side vertex arrays has significant performance implications because the CPU is used to copy the vertex data into the GPU memory for each draw call, causing a drain on memory bandwidth and lowering performance. Using VBOs avoids the additional overhead.

6.5 Handle synchronize and flush operations

6.5.1 Light- and heavyweight resolves

One of the reasons OpenGL ES implementations may introduce pipeline stalls is resolves. A resolve is a memory copy operation, which is one of the most expensive operations the driver performs while running an application.

There are two types of recognized resolves:

- Lightweight resolve – A memory copy operation that takes data located in internal graphics memory and stores it in the external memory, e.g., when the back buffer is to be displayed.
- Heavyweight resolve – A memory copy operation which first copies data from internal graphics memory to external memory, so that the bins can finish rendering, and then copies it back to internal graphics memory. Because of the double-copy, heavyweight resolves cost twice as much as the lightweight resolves. An example of a heavyweight resolve is a glReadPixels() call.

The following is a list of ES API commands that can induce a heavyweight resolve:

- eglSwapBuffers() – This call triggers the driver to flush all of its buffered rendering to the hardware. By default, the swap command copies the depth buffer content from the previous frame into each bin before starting the new frame. Starting the frame by clearing the depth buffer prevents the hardware from having to copy the buffer between frames. The swap command assumes that the color buffer will be overwritten, so the color buffer needs to be cleared only if the rendering algorithm does not paint every pixel in the buffer.
- glBindFramebuffer() – If the bound draw buffers are not cleared, the driver assumes that the previous data in the frame buffer object is needed, and the previous image for this object must be copied from slow external memory to fast internal memory. If this call is placed directly after a swap and before clearing the depth buffer that starts the new frame, only a lightweight resolve is necessary.
- glBufferData(), glBufferSubData(), glTexImage2D(), glTexSubImage2D() – These calls update memory buffers. If these functions are called in the middle of the frame, the driver creates an extra copy of the data and starts copying. If they are called after the swap and before the depth buffer clear, no resolve occurs and no extra driver overhead ensues.
- glCopyTexImage2D(), glCopyTexSubImage2D() – It is recommended to use frame buffer objects to avoid the heavyweight resolves that occur if any of the two functions are used.
- glReadPixels() – This call causes a complete flush of the GPU and idles the graphics pipeline. It is very expensive and should be avoided. Ideally, it is used directly before a swap so that it forces only a lightweight resolve instead of a heavyweight resolve, which would happen if called in the middle of a frame.

The following actions will also result in a heavyweight resolve:

- Querying objects that force the driver to return results mid-frame
- Changing frame buffer attachment configuration
- Changing contents of multisampled textures that are also rendering to

A post-processing pipeline is a typical usage scenario, in which the programmer needs to make sure only lightweight resolves are involved; e.g., a depth-of-field effect would require the following data flow:

1. After the swap, bind a frame buffer object, clear it, and render the full scene contents into it.
2. Bind another smaller frame buffer object, clear it, and render a down-sampled blurred version of the scene for future use as the out-of-focus scene.
3. Bind the back buffer, then combine contents of the two render-targets used in the previous steps to simulate the depth-of-field effect by picking either blurry or sharp pixels, depending on the distance calculated from the data in the depth buffer.

This forces a lightweight resolve of the first two frame buffer objects and another lightweight resolve on the back buffer during the swap operation.

The following is a template for a TBR-friendly rendering loop.

```
…
// put all the glTexImage2D, glTexSubImage2D, glBufferData and
glBufferSubData commands here
// FBO block – if you use one or more FBOs
for{int i = 0; i < numFBOs; i++}
{
glBindFrameBuffer(target[i], framebuffer[i]);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
// draw scene here
}
// clear color and depth buffer
glClear(GL_STENCIL_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
// glReadPixel would go here … if really needed
eglSwapBuffers(dsp, backBuffer);
```

6.5.2 Getter calls

Getter calls in OpenGL ES induce a pipeline flush which may be expensive. If the application does not use third-party software that could unknowingly modify OpenGL ES state instead of querying ES state via glGet*() calls, the software should cache the information internally.

6.6 Manage resources with OpenGL ES

When looking at a call trace of an average OpenGL ES 1.1 or ES 2.0 application, the majority of the traffic is related to state modification. The core version of these versions of the API did not offer any state containers, except buffer objects and texture objects. Given memory constraints of ES1.1- and early ES2.0-generation platforms and the lack of support for Vertex Array Objects in OpenGL ES, it was difficult for the majority of the applications to avoid frequent calls to glVertexAttribPointer(). These were often preceded with a glBindBuffer() call to set up the GL_ARRAY_BUFFER binding, per a single frame, and if these API calls occur frequently for a single frame, this affects performance. If the application can afford to use OpenGL ES 3.0 or ES 3.1 contexts, or the OpenGL ES 1.1 or ES 2.0 context provides support for VAOs via extensions, always use Vertex Array Objects.

If any of the shaders need to sample the same underlying texture data storage with different sampling configurations via a number of shader samplers from within a single draw call, or the application needs to change texture sampling settings for a specific texture object many times per frame, consider using sampler objects.

In the first of these use cases, without sampler objects the application would create at least two texture objects holding the same data, effectively wasting VRAM. In the second use case, performance would be sub-optimal. Sampler objects allow preconfigueation of the sampling configurations for intended use for accessing texture data. This could be done during loading time. Then, the application can switch quickly between these configurations as the frame is rendered. Use sampler objects whenever the application is able to use an OpenGL ES 3.0 or 3.1 context. This functionality has not been made available for OpenGL ES 1.1 or 2.0 contexts through an API extension.

The application should avoid creating objects or reinitializing object storage during the frame rendering process, regardless of the OpenGL ES version used. These tasks are very expensive and can take an indeterminate amount of time to finish. They should only be carried out during loading time. It is a frequently mistake for applications to call functions like glTexImage*() or glBufferData(), instead of correspondingly glTexSubImage*() and glBufferSubData(). The difference between the two is substantial. The former release any storage that might have been initialized earlier for a particular texture or buffer object and allocate new memory region of user-specified size. The latter use the existing storage and replace its region with new data. In terms of the cost associated with each call, it is permitted to think of these sets as malloc() and memcpy().

Finally, make sure that the application never makes redundant OpenGL ES API calls. Never assume that the OpenGL ES implementation will ignore API calls that do not change rendering context state and that they will never be passed down to the hardware.

Table 6-1 lists the API calls that modify OpenGL ES state. Try to reduce the number of calls per frame to these functions given the way the rendering engine is designed.

Table 6-1 State-modifying calls in OpenGL ES

API call	ES 2.0?	ES3.0 ?	ES3.1?
glBindBuffer	✓	✓	✓
glBindBufferBase	✗	✓	✓
glBindBufferRange	✗	✓	✓
glBindFramebuffer	✓	✓	✓
glBindImageTexture	✗	✗	✓
glBindProgramPipeline	✗	✗	✓
glBindRenderbuffer	✓	✓	✓
glBindSampler	✗	✓	✓
glBindTexture	✓	✓	✓
glBindTransformFeedback	✗	✓	✓
glBindVertexArray	✗	✓	✓
glBindVertexBuffer	✗	✗	✓
glBlendColor	✓	✓	✓
glBlendEquation	✓	✓	✓
glBlendEquationSeparate	✓	✓	✓
glBlendFunc	✓	✓	✓
glBlendFuncSeparate	✓	✓	✓
glClearColor	✓	✓	✓
glClearDepthf	✓	✓	✓
glClearStencil	✓	✓	✓
glColorMask	✓	✓	✓
glCullFace	✓	✓	✓
glDepthFunc	✓	✓	✓
glDepthMask	✓	✓	✓

API call	ES 2.0?	ES3.0 ?	ES3.1?
glDepthRangef	✓	✓	✓
glDisable	✓	✓	✓
glDisableVertexAttribArray	✓	✓	✓
glDrawBuffers	✗	✓	✓
glEnable	✓	✓	✓
glEnableVertexAttribArray	✓	✓	✓
glFramebufferParameteri	✗	✗	✓
glFramebufferRenderbuffer	✓	✓	✓
glFramebufferTexture2D	✓	✓	✓
glFramebufferTextureLayer	✗	✓	✓
glFrontFace	✓	✓	✓
glGenerateMipmap	✓	✓	✓
glHint	✗	✓	✓
glLineWidth	✓	✓	✓
glPixelStorei	✓	✓	✓
glPolygonOffset	✓	✓	✓
glProgramParameteri	✗	✓	✓
glProgramUniform*	✗	✗	✓
glReadBuffer	✗	✓	✓
glSampleCoverage	✓	✓	✓
glSampleMaski	✗	✗	✓
glSamplerParameter*	✗	✓	✓
glScissor	✓	✓	✓
glStencilFunc	✓	✓	✓
glStencilFuncSeparate	✓	✓	✓
glStencilMask	✓	✓	✓
glStencilMaskSeparate	✓	✓	✓
glStencilOp	✓	✓	✓
glStencilOpSeparate	✓	✓	✓
glTexParameter*	✓	✓	✓
glUniform*	✓	✓	✓
glUseProgram	✓	✓	✓
glVertexAttrib*	✓	✓	✓
glVertexAttribBinding	✗	✗	✓
glVertexAttribDivisor	✗	✓	✓
glVertexAttribFormat	✗	✗	✓
glVertexAttribI*	✗	✓	✓
glVertexAttribIFormat	✗	✗	✓

API call	ES 2.0?	ES3.0 ?	ES3.1?
glVertexAttribIPointer	✗	✓	✓
glVertexAttribPointer	✓	✓	✓
glVertexBindingDivisor	✗	✗	✓
glViewport	✓	✓	✓

6.7 Efficient state management

One of the important things when implementing OpenGL ES applications is to ensure that the number of API calls that modify the OpenGL ES configuration is reduced to a minimum.

Table 6-2 lists the types of state containers OpenGL ES supports, along with a short description of what state each object type holds.

Table 6-2 Object types in OpenGL ES

Type	ES 2?	ES 3?	ES 3.1?	Description
Buffer object	✓	✓	✓	Encapsulates a memory region that is allocated by the driver, along with the following information: ▪ (ES2.0 / ES3 / ES3.1) Storage properties (access flags, size, usage hint, etc.) ▪ (ES3.0 / ES3.1) Mapping status
Frame buffer object	✓	✓	✓	▪ Encapsulates description of a frame buffer, along with state descriptors of all the attachments Frame buffer state includes: ▪ (ES3 / ES3.1) Draw buffer configuration ▪ (ES3 / ES3.1) Read buffer configuration ▪ (ES3.1) Default frame buffer properties ▪ (height, width, etc.) Each attachment point consists of the following state: ▪ (ES2 / ES3 / ES3.1) Name & type of the attached object ▪ (ES2 / ES3 / ES3.1) Binding properties, as configured by glFramebufferRenderbuffer() or glFramebufferTexture*() calls. ▪ (ES3 / ES3.1) Data type of components of the attachment ▪ (ES3 / ES3.1) Size in bits of attached image's components

Type	ES 2?	ES 3?	ES 3.1?	Description
Pipeline object			✓	▪ Encapsulates a rendering pipeline consisting of all shader stages which will be used for the draw calls, whenever no program object is active Apart from shader programs defining the pipeline, the following state is stored: ▪ ID of a program object that glUniform*() commands will update ▪ Validation status & info log for the pipeline object
Program object	✓	✓	✓	Contains a list of attached shaders (one per stage) and, potentially, a program that is an outcome of linking those. Also stores the following information: ▪ (ES2 / ES3 / ES3.1) Flag telling if the last linking attempt has succeeded ▪ (ES2 / ES3 / ES3.1) Flag telling if the last validation attempt has succeeded ▪ (ES3.1) Flag telling if the program object can be bound for separate pipeline stages ▪ (ES2 / ES3 / ES3.1) Flag telling if the program object has been marked for deletion ▪ (ES2 / ES3 / ES3.1) List of active attributes and uniforms ▪ (ES3.1) Local work-group size for a linked compute program ▪ (ES2 / ES3 / ES3.1) Linking info log ▪ (ES3 / ES3.1) Program binary blob & its properties ▪ (ES3 / ES3.1) Transform feedback properties (buffer mode, varyings) ▪ (ES3 / ES3.1) List of active uniform blocks, along with their properties, binding points, and information on which shader refers to each uniform block ▪ (ES3.1) Number of active atomic counter buffers used by the program
Query object		✓	✓	Encapsulates the result value for an asynchronous query, as well as a Boolean telling if the result can be retrieved already
Renderbuffer object		✓	✓	Encapsulates properties of the renderbuffer specified at the creation time
Sampler object		✓	✓	Encapsulates all state that affects the texture sampling process: ▪ (ES3.0 / ES3.1) Comparison function and mode ▪ (ES3.0 / ES3.1) Magnification and minification filter ▪ (ES3.0 / ES3.1) Maximum and minimum level of detail ▪ (ES3.0 / ES3.1) S/T/R wrap modes

Type	ES 2?	ES 3?	ES 3.1?	Description
Shader object	✓	✓	✓	Encapsulates the following state related to a single shader instance: ▪ (ES2.0 / ES3.0 / ES3.1) Flag telling if the shader has been marked for deletion ▪ (ES2.0 / ES3.0 / ES3.1) Flag telling if the last compilation attempt was successful ▪ (ES2.0 / ES3.0 / ES3.1) Info log ▪ (ES2.0 / ES3.0 / ES3.1) Shader source ▪ (ES2.0 / ES3.0 / ES3.1) Type of the shader
Sync object		✓	✓	Encapsulates the following state used for client/ server synchronization: ▪ (ES3.0/ES3.1) Synchronization object properties (condition, flags, status), as configured by a glFenceSync() call ▪ (ES3.0/ES3.1) Type of the sync object
Texture object	✓	✓	✓	Encapsulates texture mipmaps, a set of properties per each mipmap, as well as the following state: ▪ • (ES3.1) Depth/stencil texture mode switch ▪ (ES3.0 / ES3.1) Flag telling if the texture format and size has been made immutable ▪ (ES3.1) Number of texture levels of the immutable texture ▪ (ES3.0 / ES3.1) Swizzle configuration ▪ (ES2.0 / ES3.0 / ES3.1) Texture sampling state – see "Sampler Object" row for the list of properties (ES3.1) Each texture mipmap carries the following state (queriable with glGetTexLevelParameter*()): ▪ Flag telling if the texture contents are compressed ▪ Flag telling whether the mipmap uses a fixedsample pattern (in case of multisample textures) ▪ Internal format used for the data ▪ Number of bits used to store Alpha/Blue/ Depth/Green/Red/Shared Exponent/Stencil data ▪ Number of samples ▪ Resolution ▪ Type of Alpha/Blue/Depth/Green/Red components
Transform feedback object		✓	✓	Encapsulates the following state used for the transform feed-back process: ▪ (ES3.0 / ES3.1) Buffer object regions bound to transform feedback attribute streams ▪ (ES3.0 / ES3.1) Flag telling if the transform feedback process is ongoing ▪ (ES3.0 / ES3.1) Flag telling if the transform feedback process has been paused

Type	ES 2?	ES 3?	ES 3.1?	Description
Vertex array object		✓	✓	Encapsulates all state related to generic vertex attribute array configuration. Specifically: ▪ (ES3.0 / ES3.1) Generic vertex attribute array properties stored separately for each vertex attribute array (divisor, "is integer?" flag, "normalized" flag, pointer, size, stride, type), as specified by glVertexAttrib*Pointer() call ▪ (ES3.0 / ES3.1) GL_ARRAY_BUFFER binding that was active at the glVertexAttrib*Pointer() call for each vertex attribute array ▪ (ES3.0 / ES3.1) GL_ELEMENT_ARRAY_BUFFER binding ▪ (ES3.1) Vertex buffer binding for each vertex attribute array ▪ (ES3.1) Vertex buffer binding properties (instance divisor, offset, relative offset, stride, vertex binding ID)

After ensuring that the application takes full advantage of the state containers available in the version of OpenGL ES in use, the next step to improve rendering performance is to ensure that the application does not make redundant API calls. It is a common pitfall for OpenGL ES applications to activate a program object that is already active, or to assign the values to shader uniforms that are exactly the same as the existing uniform values. To keep the overhead low, even though these calls are redundant, the driver may pass them down to hardware, effectively wasting time that could be spent on more important matters.

Lastly, it is always a good idea to sort draw calls to reduce the number of state changes that the OpenGL ES application will make. QTI recommends that draw calls be sorted in the following order:

1. By render target – Switching between render targets (using glBindFramebuffer or glFramebuffer* calls) is very expensive, so only reconfigure the frame buffer bindings or active draw frame buffer attachments after the rendering of all objects for the current frame buffer object configuration is complete. If the object needs to be drawn using multiple programs into multiple frame buffers, it is cheaper to switch between the programs first and then the frame buffers. This approach minimizes the number of calls to glBindFramebuffer and/or glFramebuffer* at the cost of an increased number of glUseProgram calls.
2. By program – For a given frame buffer, ensure the number of program switches is reduced to the minimum. For instance, if a scene consists of three objects, two of which use the same fragment shader, it is a good idea to render these two objects first, then change the program and proceed with rendering the remaining object.
3. By other state – Other state changes are less expensive, so further ordering of draw calls depends on the application. A rule of a thumb is that the fewer state switches, the better.

6.8 Energy saving tips

The rendering process is a complex activity that consumes a lot of power. There are a few aspects of rendering that engine programmers should consider when designing applications, to ensure that the battery consumption level stays reasonable. They are as follows:

- Is it acceptable for the application to render a frame every once in a while, or is the user expecting the visuals to be regularly updated?

  As an example, in a photo browser application, it would be acceptable to lock the rendering pipeline once a photo has been shown, and keep the rendering process locked until the user started interacting with the image in some way; e.g., if they tried to zoom into it or move in a certain direction. At that point, it would resume the rendering process. Rendering a stream of frames that look identical makes no sense and consumes more power.

  However, if any part of the UI is animated, it could justify more recurrent updates. To reduce the cost of rendering the UI-only frame, the application could cache the way the photo is presented when a regular frame is rendered, and then for the less expensive frames, it could blit that texture to the back buffer and render the HUD over it.

- If the application needs frequent redraws, how regular do these updates have to be?

  A video player application would be acceptable to cap its rendering rate at the frame rate of the video clip that is being played. However a video game, which could provide a smoother visual experience when allowed to render more frames per second, could paint as many frames as possible if the device is connected to a power source, or could be restricted to draw up to 30 frames per second whenever the device is powered by the battery.

The intensity of battery consumption can usually be controlled by the application and the scalability of that drainage depends on what the application does and how its rendering pipeline is constructed.

The application can use a frame rate limiter to reduce the number of frames that get drawn per second. There are a number of ways to implement this functionality:

- Provided that it is an acceptable solution to use the display refresh rate as the basis for reducing the number of frames that are drawn per second, the quickest way is to call the eglSwapInterval function.

  This function takes a single argument, which specifies the minimum number of video frames that are displayed before a buffer swap occurs. This means that if the display has a refresh rate of 60 Hz and a value of 2 was used, then no more than 30 frames would be rendered every second. This is enforced by blocking the execution flow in eglSwapBuffers until the front/back buffers are switched by hardware.

NOTE: Hardware updates the display contents at regular intervals. If the application calls eglSwapBuffers too late, it may miss the vsync time window for a given frame and remain blocked until the next video sync event occurs.

Using a value of 0 causes eglSwapBuffers calls to instantly flip the front buffer with the back buffer, usually causing a tearing effect, which is significantly more noticeable during dynamic animations.

By default, eglSwapInterval is configured to use a value of 1.

- More complex engines often distribute the workload to multiple threads and execute the computations outside the rendering pipeline. When it is time to render a frame, they use the latest data that is available. In this case, it may be more reasonable to use a bespoke frame-rate limiter, which wakes up sufficiently early for each frame, draws the scene, and finally calls eglSwapBuffers() before vsync occurs.

  Actual implementations of a customized frame-rate limiter vary and usually involve heuristics to a certain degree, but the general idea is as follows:

```
time_t current_time;
for (each frame)
{    time_t wakeup_time =
getNextWakeupTime()
   while ( (current_time = currentTime()) <
wakeup_time)    {       time_t sleep_time =
getSleepTime(current_time,
wakeup_time);
      sleep(sleep_time);    }
   draw();
   eglSwapBuffers(display, draw_surface_to_swap);
}
```

NOTE: sleep() does not guarantee that the thread will be returned execution right after the specified time interval passes. However, time quanta allocated to threads by modern operating systems are usually sufficiently small for 3D applications to safely rely on this approach.

6.9 OpenGL ES and multithreading

All rendering commands in OpenGL ES must operate within a rendering context, which encapsulates ES objects, frame buffer configuration, and various context-wide state information. Before the application can proceed with issuing rendering commands, it must perform the following:

1. Initialize EGL for a display using eglInitialize().
2. Create draw and read surface(s) using eglCreatePbufferSurface(), eglCreatePixmapSurface(), or eglCreateWindowSurface() call(s).
3. Create a rendering context using eglCreateContext().
4. Bind a triple {ES context, draw surface, read surface} to the current thread by calling eglMakeCurrent().

The application also needs to retrieve ES function pointers using eglGetProcAddress(). This action can be performed at any time, because it does not depend on a specific display or on the currently bound rendering context.

Once a rendering context is set up and bound to a specific thread, the application is free to issue ES calls from within that thread. Each OpenGL ES API call made from then on will be queued for actual execution at some point in the future when the pipeline is flushed.

If the application needs to use multiple rendering contexts, it simply repeats Steps 2-4 for each new context, with the assumption that only one rendering context will be assigned to a particular thread at the same time. This does not prevent multiple rendering contexts from being active, as long as each one of them gets their own hosting thread.

Multiple rendering contexts are not assigned separate hardware command buffers. Instead, calls in the command queue are serialized and processed one after another.

Despite the fact that OpenGL ES in general is an asynchronously executed API, some of the API commands take a while before they return the execution flow. Putting aside those of the functions that wait until the pipeline flushes or the queued commands finish executing on the GPU side, e.g., waits issued on fence syncs injected into command stream, there is a class of functions that take client-side data and transfer it to a memory region chosen by the driver:

- glBufferData()
- glBufferSubData()
- glCompressedTexImage2D() and glCompressedTexImage3D()
- glCompressedTexSubImage2D() and glCompressedTexSubImage3D()
- glTexImage2D() and glTexImage3D()
- glTexSubImage2D() and glTexSubImage3D()

Depending on the amount of data that needs to be transferred, these functions can take a fairly lengthy period of time before they return execution flow to the caller. Each of the functions can only leave as soon as they are entirely sure that all data has reached the destination. Had they returned earlier, the caller would be free to modify the source buffer or could release it, regardless of the undergoing memory copy operations.

Lengthy API calls do not fit well into the picture of smooth rendering process. An application that aims at a frame rate of 60 FPS has a mere ~16,7 ms time budget for each frame. If the time needed to fill the back buffer with contents exceeds that time window, the performance goal can no longer be met and the overall user experience degrades.

Multiple rendering contexts allow the applications to issue the memory transfers on a separate thread (or threads), while keeping the rendering process ongoing in another thread. While the general performance of the GPU may reduce during the process, the approach allows the applications to show things like an animated loading screen or to draw a simple 3D scene that would keep the user busy, as the geometry and texture data for the next level is being loaded into VRAM.

Auxiliary rendering contexts can also be used for transparent shader compilation or program linking. Since the two activities are handled exclusively by CPU, but can only be invoked from within a rendering context, deferring these tasks to helper threads can help maintain the rendering performance of the main rendering context, as long as each context is assigned a separate CPU core.

Creating more rendering contexts than there are CPU cores available is inefficient and is not recommended.

Objects created in one rendering context cannot be accessed by default in another context. For that to happen, auxiliary contexts need to have a parent rendering context assigned during creation time. For performance reasons, it is recommended to create all contexts during startup, rather than later during runtime.

Not all OpenGL ES objects can be shared between rendering contexts. The following object types can be accessed by all rendering contexts which share the same namespace:

- Buffer objects
- Program objects
- Renderbuffer objects
- Sampler objects
- Shader objects
- Sync objects
- Texture objects (except for the texture object named 0)

The following objects are cannot shared between rendering contexts:

- Frame buffer objects
- Program pipeline objects
- Query objects
- Transform feedback objects
- Vertex array objects

Changes applied to a shared object in rendering context A are not instantly visible to rendering context B; e.g., consider a texture object. If updating the storage of the texture in rendering context A, for the changes to become visible in context B, the following actions need to be undertaken (in the order specified):

1. The command that changed the storage contents need to actually complete before it can be accessed by the other context. This can be ensured by:
   a. (ES 2.0/ES 3.0/ES 3.1) calling glFinish() in rendering context A
   b. (ES 3.0/ES 3.1) injecting a fence into command buffer in rendering context A and waiting for it to become signaled in rendering context B; this approach is much faster
2. In rendering context B, the texture object needs to be rebound to texture target for the changes to be guaranteed to have propagated.

Since API commands executed in multiple threads get serialized in the driver layer, it does not make sense to distribute the rendering process across multiple rendering contexts. Not only will that make the application significantly more complex, but it will also result in reduced performance This is due to the extra glFinish() call or fence waits that are necessary for the inter-context object synchronization. The following two code snippets show the inefficient usage of multiple contexts in a simple example where:

- Rendering context A renders geometry to a texture
- Rendering context B clears its draw buffers and waits for the texture of context A to become available, so that it can use it for a draw call

Inefficient context sharing usage – Rendering thread A

```
fence = NULL;
glBindFramebuffer     (..);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D,
texture_id,                        0); /*
level */
glBindVertexArray(..);
glDrawArrays     (..)
fence = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE,
                    0); /* flags */
```

Inefficient context sharing usage – Rendering thread B

```
while (fence == NULL)
{ }
glClear   (GL_COLOR_BUFFER_BIT);
glWaitSync(fence,            0,
/* flags */
GL_TIMEOUT_IGNORED);
glBindTexture(GL_TEXTURE_2D,
texture_id); glDrawArrays (..)
```

7 Comparing OpenGL ES Versions and Optional Features

7.1 OpenGL ES features by version

Table 7-1 compares the feature sets of OpenGL ES 2.0, OpenGL ES 3.0, OpenGL ES 3.1, and desktop OpenGL 4.4.

These APIs cover a wide range of features that are not all listed. The listed features are ones that a developer is most likely to find useful.

Table 7-1 Feature set comparison

Feature	ES2 (Adreno 2xx)	ES3 (Adreno 3xx)	ES3.1 (Adreno 4xx)	GL4.4
1D textures	✗	✗	✗	✓
1D array textures	✗	✗	✗	✓
2D array textures	✗	✓	✓	✓
2D multisample array textures	✗	✗	Optional[a]	✓
2D multisample textures	✗	✗	✓	✓
2D textures	✓	✓	✓	✓
3D textures	✗	✓	✓	✓
Advanced blending modes	Optional [b]	Optional [b]	Optional [b]	Optional [b]
Arrays of arrays support in GLSL	✗	✗	✓	✓
Atomic counters	✗	✗	✓	✓
Atomic integer operations in GLSL	✗	✗	✓	✓
Buffer objects (immutable)	✗	✗	✗	✓
Buffer objects (mutable)	✓	✓	✓	✓
Buffer textures	✗	✗	Optional [c]	✓
Compressed textures (ASTC LDR)	Optional [d]	Optional [d]	Optional [d]	Optional [d]
Compressed textures (ETC1)	Optional [e]	✓	✓	✓
Compressed textures (ETC2)	✗	✓	✓	✓
Compute shaders	✗	✗	✓	✓
Conditional rendering	✗	✗	✗	✓
Cubemap textures	✓	✓	✓	✓

Feature	**ES2 (Adreno 2xx)**	**ES3 (Adreno 3xx)**	**ES3.1 (Adreno 4xx)**	**GL4.4**
Cubemap array textures	Χ	Χ	Optional [f]	✓
Depth/stencil texture mode	Χ	Χ	✓	✓
Explicit uniform locations in GLSL	Χ	Χ	✓	✓
Fence sync objects	Χ	✓	✓	✓
Fragment shaders	✓	✓	✓	✓
Frame buffer blitting	Optional [g]	✓	✓	✓
Geometry shaders	Χ	Χ	Optional [m]	✓
Image support	Χ	Χ	✓	✓
Indirect draw calls	Χ	Χ	✓	✓
Instanced draw calls	Optional [h]	✓	✓	✓
Integer GLSL types	Χ	✓	✓	✓
Integer textures	Χ	✓	✓	✓
Multiple render target support	Χ	✓	✓	✓
Multisample textures	Χ	Χ	✓	✓
Occlusion queries	Optional [i]	✓	✓	✓
Packing functions in GLSL	Χ	✓	✓	✓
Per-sample shading	Χ	Optional [j]	Optional [j]	✓
Pixel buffer objects	Optional [g]	✓	✓	✓
Primitive restart (fixed index)	Χ	✓	✓	✓
Program binaries	Optional [k]	✓	✓	✓
Program interface query	Χ	Χ	✓	✓
Rectangle textures	Χ	Χ	Χ	✓
Sampler objects	Χ	✓	✓	✓
Separate shader objects	Optional [l]	Optional [l]	✓	✓
Shader binaries	✓	✓	✓	✓
Shader storage buffer objects	Χ	Χ	✓	✓
Subroutines	Χ	Χ	Χ	✓
Tessellation shaders	Χ	Χ	Optional [m]	✓
Texture objects (immutable)	Optional [n]	✓	✓	✓
Texture objects (mutable)	✓	✓	✓	✓
Texture swizzles	✓	✓	✓	✓
Timer queries	Optional [o]	Optional [o]	Optional [o]	✓
Transform feedback	Χ	✓	✓	✓
Uniform buffer objects	Χ	✓	✓	✓
Vertex array objects	Optional [p]	✓	✓	✓
Vertex attribute bindings	Χ	Χ	✓	✓

Feature	ES2 (Adreno 2xx)	ES3 (Adreno 3xx)	ES3.1 (Adreno 4xx)	GL4.4
Vertex shaders	✓	✓	✓	✓

[a]Supported via GL_OES_texture_storage_multisample_2d_array extension
[b]Supported via GL_KHR_blend_equation_advanced extension
[c]Supported via GL_EXT_texture_buffer and GL_ANDROID_extension_pack_es31a extensions
[d]Supported via GL_KHR_texture_compression_astc_ldr or GL_ANDROID_extension_pack_es31a extensions
[e]Supported via GL_OES_compressed_ETC1_RGB8_texture extension
[f]Supported via GL_EXT_texture_cube_map_array or GL_ANDROID_extension_pack_es31a extensions
[g]Only available via vendor-specific extensions
[h]Supported via GL_EXT_instanced_arrays extension
[i]Supported via GL_EXT_occlusion_query_boolean extension
[j]Supported via a combination of GL_OES_sample_shading and GL_OES_sample_variables extensions OR via GL_ANDROID_extension_pack_es31a extension
[k]Supported via GL_OES_get_program_binary extension
[l]Supported via GL_EXT_separate_shader_objects extension
[m]Supported via GL_EXT_tessellation_shader or GL_ANDROID_extension_pack_es31a extensions
[n]Supported via GL_EXT_texture_storage extension
[o]Supported via GL_EXT_disjoint_timer_query extension
[p]Supported via GL_OES_vertex_array_object extension

Adreno 3xx supports the optional OpenGL ES 1.x extensions listed in Table 7-2.

Table 7-2 Supported optional OpenGL ES 1.x extensions

Extension	Short description
GL_AMD_compressed_ATC_texture	Support for ATC compressed texture formats
GL_APPLE_texture_2D_limited_npot	Support for Non-Power-Of-Two (NPOT) dimensions for 2D textures
GL_ARB_vertex_buffer_object	Support for vertex buffer objects
GL_EXT_texture_filter_anisotropic	Enables the OpenGL application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering
GL_EXT_texture_format_BGRA8888	Provides an additional format and type combination for use when specifying texture data
GL_EXT_texture_type_2_10_10_10_REV	Adds a new texture data type, unsigned 2.10.10.10 ABGR, which can be used with RGB or RGBA formats
GL_OES_blend_equation_separate	Provides a separate blend equation for RGB and alpha to match the generality available for blend factors
GL_OES_blend_func_separate	Extends blending capability by defining a function that allows independent setting of the RGB and alpha blend factors for blend operations that require source and destination blend factors
GL_OES_blend_subtract	Similar to default blending function, but produces the difference of left and right hand sides rather than the sum
GL_OES_compressed_ETC1_RGB8_texture	Enables direct support of compressed textures in the Ericsson Texture Compression (ETC) formats in OpenGL ES

Extension	Short description
GL_OES_compressed_paletted_texture	Enables direct support of palletized textures in OpenGL ES
GL_OES_depth_texture	Defines a new texture format that stores depth values in the texture
GL_OES_depth24	Enables 24-bit depth components as a valid render buffer storage format
GL_OES_draw_texture	Defines a mechanism for writing pixel rectangles from one or more textures to a rectangular region of the screen
GL_OES_EGL_image	Provides a mechanism for creating texture and renderbuffer objects sharing storage with specified EGLImage objects
GL_OES_EGL_image_external	Provides a mechanism for creating EGLImage texture targets from EGLImages
GL_OES_framebuffer_object	Defines a simple interface for drawing to rendering destinations other than the buffers provided to GL by the window system
GL_OES_matrix_palette	Provides the ability to support vertex skinning in OpenGL ES
GL_OES_packed_depth_stencil	Enables the interleaving of depth and stencil buffers into one buffer
GL_OES_point_size_array	Extends how points and point sprites are rendered by allowing an array of point sizes, instead of a fixed input point size given by PointSize
GL_OES_point_sprite	Enables applications to use points rather than quads
GL_OES_read_format	Enables the querying of an OpenGL implementation for a preferred type and format combination for use with reading the color buffer with the ReadPixels command
GL_OES_rgb8_rgba8	Enables RGB8 and RGBA8 renderbuffer storage formats
GL_OES_stencil_wrap	Extends the StencilOp functions to support GL_INCR_WRAP and GL_DECR_WRAP modes
GL_OES_texture_cube_map	Adds cube map support
GL_OES_texture_env_crossbar	Enables the use of texture color from other texture units as source for the COMBINE environment function
GL_OES_texture_float	Adds texture formats with 16- (aka half float) and 32-bit floating-point components
GL_OES_texture_half_float	Adds texture formats with 16- (aka half float) and 32-bit floating-point components
GL_OES_texture_half_float_linear	Expands upon the OES_texture_half_float and OES_texture_float extensions
GL_OES_texture_mirrored_repeat	Extends the set of texture wrap modes to include mirror repeat
GL_OES_texture_npot	Adds support for the REPEAT and MIRRORED_REPEAT texture wrap modes and the minification filters supported for NPOT 2 D textures, cubemaps and for 3D textures, if the OES_texture_3D extension is supported
GL_QCOM_binning_control	A Qualcomm specific extension that enables control of the deferred rendering process on Adreno GPUs

Extension	Short description
GL_QCOM_extended_get	A Qualcomm specific extension that enables instrumenting of the driver for debugging of OpenGL ES applications
GL_QCOM_tiled_rendering	A Qualcomm specific extension that allows the application to specify a rectangular tile rendering area and have full control over the resolves for that area

Adreno 3xx and Adreno 4xx support the OpenGL ES 2.0 extensions listed in Table 7-3.

Table 7-3 OpenGL ES 2.0 extensions supported by Adreno

Extension name	Description
GL_AMD_compressed_ATC_texture	Enables support for ATC compressed texture formats
GL_AMD_program_binary_Z400	Provides support for a program binary format called GL_Z400_BINARY_AMD
GL_EXT_texture_filter_anisotropic	Permits the OpenGL ES application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering.
GL_EXT_texture_format_BGRA8888	Provides an additional format and type combination for use when specifying texture data
GL_EXT_texture_type_2_10_10_10_REV	Adds a new texture data type (unsigned 2.10.10.10 ABGR) which can be used with RGB or RGBA formats
GL_NV_fence	Provides a finer granularity of synchronizing ES command completion than standard OpenGL ES
GL_OES_compressed_ETC1_RGB8_texture	Provides direct support of compressed textures in the Ericsson Texture Compression (ETC) formats
GL_OES_depth_texture	Defines a new texture format that stores depth values in a texture
GL_OES_depth24	Enables 24-bit depth components as a valid renderbuffer storage format
GL_OES_EGL_image	Provides a mechanism for creating texture and renderbuffer objects that share storage with so-called EGLImage objects; these objects can be backed by data coming from special hardware or other sources that are not directly related to OpenGL ES
GL_OES_EGL_image_external	Provides a mechanism for creating EGLImagetexture targets from EGLImages
GL_OES_element_index_uint	Allows GL_UNSIGNED_INT to be passed as the type argument to glDrawElements
GL_OES_fbo_render_mipmap	Enables rendering to any mipmap level of a texture that is attached to a frame buffer object
GL_OES_fragment_precision_high	Provides the ability to determine within the API if high-precision fragment shader varyings are supported
GL_OES_get_program_binary	Allows the application to use OpenGL ES as an offline compiler
GL_OES_packed_depth_stencil	Enables the interleaving of depth and stencil buffers into one buffer
GL_OES_rgb8_rgba8	Enables GL_RGB8 and GL_RGBA8 renderbuffer storage formats

Extension name	Description
GL_OES_standard_derivatives	Provides access to the standard derivative built-in function
GL_OES_texture_3D	Provides support for 3D textures
GL_OES_texture_float	Provides texture formats with 32-bit floating-point components
GL_OES_texture_half_float	Provides texture formats with 16-bit (half float) floating-point components
GL_OES_texture_half_float_linear	Expands on the GL_OES_texture_half_float and GL_OES_texture_float extensions; introduces support for the GL_LINEAR magnification filter and GL_LINEAR and mipmap-based minification filters for both 16-bit and 32-bit floating-point texture formats
GL_OES_texture_npot	Removes certain restrictions limiting the use of texture wrap modes and minification filters with nonpower-of-two textures; this applies to 2D textures, cube maps, and, if the GL_OES_texture_3D extension is available, 3D textures
GL_OES_vertex_array_object	Introduces support for vertex array objects
GL_OES_vertex_half_float	Adds a 16-bit floating-point data type (half float) to vertex data specified using vertex arrays
GL_OES_vertex_type_10_10_10_2	Provides support for the 10:10:10:2 data format vertex type
GL_QCOM_alpha_test	A Qualcomm-specific extension that reintroduces the alpha test per-fragment operation
GL_QCOM_binning_control	A Qualcomm-specific extension that provides control of the deferred rendering process on Adreno GPUs
GL_QCOM_driver_control	A Qualcomm-specific extension that exposes special control features in a graphics driver
GL_QCOM_extended_get	A Qualcomm-specific extension that enables the instrumentation of the driver for debugging OpenGL ES applications
GL_QCOM_extended_get2	A Qualcomm-specific extension that enables the instrumentation of the driver for debugging OpenGL ES applications
GL_QCOM_perfmon_global_mode	A Qualcomm-specific extension that enables performance monitoring
GL_QCOM_tiled_rendering	A Qualcomm-specific extension that enables the application to specify a rectangular tile rendering area and have full control over the resolves for that area
GL_QCOM_writeonly_rendering	A Qualcomm-specific extension that defines a specialized write-only rendering mode that may offer a performance boost for simple 2D rendering

Table 7-4 lists the optional OpenGL ES 3.x extensions.

Table 7-4 Supported optional OpenGL ES 3.x extensions

Extension	Short description
GL_ANDROID_extension_pack_es31a	Umbrella extension string used to indicate support for all extensions in the Android Extension Pack
GL_EXT_occlusion_query_boolean	Allows an application to query whether any pixels are drawn when rending a group of primitives
GL_EXT_disjoint_timer_query	Allows the application to query the amount of time needed to complete a set of GL commands
GL_EXT_geometry_shader	Introduces support for the geometry shader stage (see Chapter 5)
GL_EXT_gpu_shader5	Extends OpenGL ES 3.1 Shading Language capabilities
GL_EXT_primitive_bounding_box	Allows geometry and tessellation shaders to provide additional optimization hints to the OpenGL ES implementation
GL_EXT_shader_implicit_conversions	A subset of the functionality provided by GL_EXT_gpu_shader5, mostly in the area of implicit conversions
GL_EXT_shader_io_blocks	Extends the functionality of interface blocks to provide a means of passing data, including arrays, between shader stages
GL_EXT_tessellation_shader	Introduces support for the tessellation control and tessellation evaluation shader stages (see Chapter 5)
GL_EXT_texture_buffer	Introduces support for buffer textures
GL_EXT_texture_cube_map_array	Introduces support for cubemap array textures
GL_EXT_texture_view	Introduces support for texture views
GL_KHR_blend_equation_advanced	Adds more complex options for the blending stage
GL_KHR_texture_compression_astc_ldr	Adds support for the ASTC texture compression format
GL_OES_sample_shading	Gives applications greater control over fragment shaders when using multisampling
GL_OES_sample_variables	Gives applications greater control over fragment shaders when using multisampling
GL_OES_texture_storage_multisample_2d_array	Introduces support for immutable two-dimensional multisample array textures

Links to the specifications of each of these extensions can be found in the Khronos Registry at http://www.khronos.org/registry/gles/.

7.2 Optional language features

An OpenGL ES implementation can provide functionality that goes beyond that defined in the OpenGL ES core specification. It does this by providing support for one or more OpenGL ES extensions.

To make effective use of extensions, there are two things to do:

- It may be ecessary to be able to determine what extensions are supported by the OpenGL ES context that is being used. This allows an implementation of a fallback code path for the case when the extension that is planned to be used turns out to be unavailable on a particular platform.
- In shader code, it may be necessary to specifically request that support for the extension should be enabled.

7.2.1 ES extension enumeration

This section explains how to enumerate extensions supported by the OpenGL ES implementation that is handling the application.

There are two ways to do this:

- Use a glGetString call with the name argument set to GL_EXTENSIONS. The function returns a NULL-terminated string with a list of extensions, where each extension is separated by a space.

  This method is supported by OpenGL ES 2.0, OpenGL ES 3.0 and OpenGL ES 3.1.
- Retrieve the number of extensions supported by the current OpenGL ES context. This is done using a glGetIntegerv call with the pname parameter set to GL_NUM_EXTENSIONS. Then use a call to glGetStringi for each extension index in the valid range, again using a pname parameter value of GL_EXTENSIONS. This is to receive a single extension name per call, avoiding the need for string parsing code.

  This method is supported by OpenGL ES 3.0 and OpenGL ES 3.1.

OpenGL ES extensions either expand the core functionality of the API or provide a faster means of performing certain tasks. It is recommended that to implement alternative code paths to take advantage of these features when available.

For more information about extensions supported by Adreno in an OpenGL ES 3.1 context, see Chapter 5.

7.2.2 Enabling extensions in shaders

This section explains how to enable extension support in OpenGL ES Shading Language shaders.

If an extension introduces new entry points or enums to the API, these can be used at any time, without explicitly requesting OpenGL ES to enable them for a particular rendering context. However, shaders written in OpenGL ES Shading Language need to specifically make a request for each extension before the new functionality can be used.

Before using the extended features of the OpenGL ES Shading Language, first include a special declaration to inform the compiler that a nonstandard functionality is used in the code. This must be positioned ahead of the first use of any new OpenGL ES Shading Language function or type.

This is done using a #extension declaration, as follows:

```
#extension <extension_name> : <behavior>
```

where:

- <extension_name> is replaced with the name of extension that describes the additional functionality that is needed. Can also use "all" here to refer to all OpenGL ES Shading Language extensions supported by the compiler.
- <behavior> is replaced with one of the following values:
    - Disable – The compiler will behave as if the extension is not a part of the language definition. If <extension_name> is not a known extension, a warning will be logged to the shader info log.
    - Enable – The compiler will behave as if the extension is a part of the language definition. If <extension_name> is not a known extension, a warning will be logged to the shader info log.
    - Require – The compiler will behave as if the extension is a part of the language definition. If <extension_name> is not a known extension, the compilation will fail.
    - Warn – The compiler will behave as if the extension is a part of the language definition. If any use of extension-specific functionality is detected, a warning will be logged to the shader info log.

NOTE: It is possible to use more than one of of these declarations, even for the same extension name, as subsequent declarations override the previous ones.

By default, support for all OpenGL ES Shading Language extensions is disabled and needs to be explicitly enabled.

8 Understanding the Developer Tools

8.1 Adreno profiler

Improving the performance of a 3D application is a challenging process. Without a proper toolset, developers often find themselves resorting to a trial and error method to find a bottleneck or identify the source of a visual glitch, forcing an application rebuild for each attempt. This is a time-consuming and cumbersome process.

Rarely do developers have access to a raw list of the ES commands that the application issues. This makes the optimization process tricky, since the developer is likely to know and understand what their ES application does in general, but may not have complete visibility into the rendering process. This means that it is possible for redundant ES calls to be issued, lowering the rendering efficiency. It would be beneficial for ES application developers to have access to a tool that warns about areas which could benefit from improved GL state management handling.

The reason for poor rendering performance is often unclear at first. Having real time insight into detailed GPU utilization, texture cache miss or pipeline stall statistics would make it easier to identify the reason for a slowdown.

Sometimes the rendering process consists of many draw calls and it is difficult to identify specifically which of these draw calls is responsible for rendering a broken mesh. It is helpful to have the ability to highlight the geometry that was drawn as a result of any specific draw call.

These are some of the situations where the Adreno Profiler can help, resulting in better rendering performance and shorter application development times.

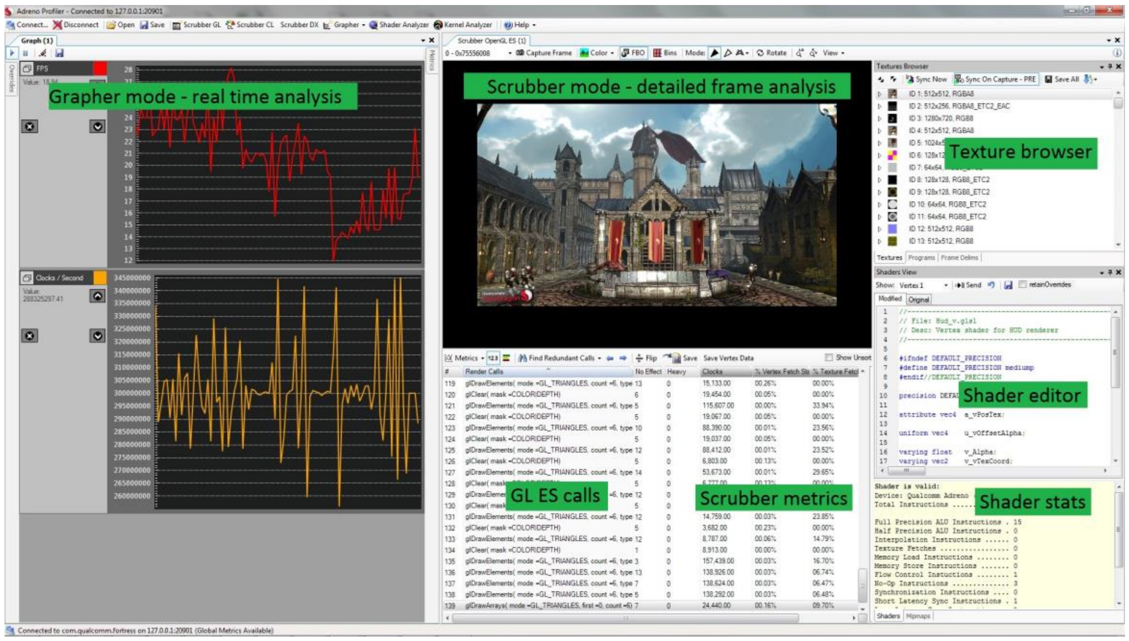
Grapher mode - real time analysis
Scrubber mode - detailed frame analysis
Texture browser
Shader editor
GL ES calls
Scrubber metrics
Shader stats

Figure 8-1 Adreno profiler

The tool works in three different modes, each suiting a different purpose:

- Scrubber mode – Capture frames from an animation running on Qualcomm-powered embedded platforms and inspect the call trace in detail
- Grapher mode – Plot real time performance data streaming from the embedded graphics driver and the system
- Shader Analyzer mode – Analyze fragment and vertex shaders offline to determine how expensive they will be to run on an actual Adreno device

8.1.1 Scrubber mode

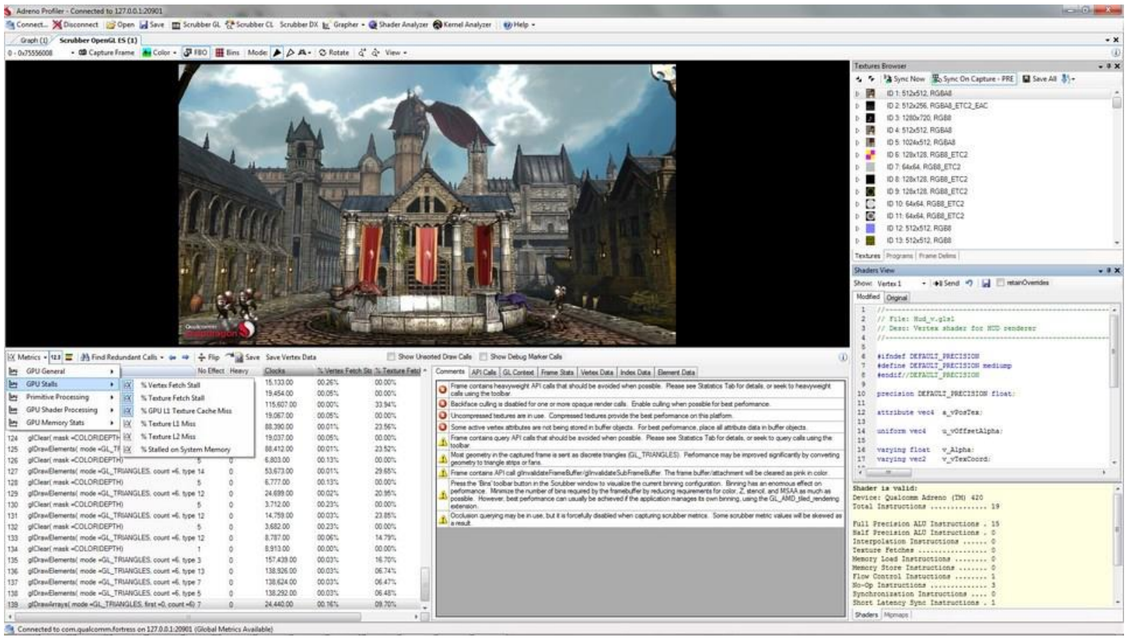

Figure 8-2 Adreno profiler: scrubber mode

This mode allows frame capturing for offline analysis. The call trace can then be used for the following:

- Capture per-render-call GPU metrics, e.g., clock cycles expended, vertex and fragment counts, and more (OpenGL ES 2.0 applications only)
- Collect and display important performance statistics
- Download, inspect, and override textures
- Modify any context state variable for a particular draw call and see the resulting change in the emulator
- Scrub the captured call trace through the integrated OpenGL ES emulator
- Search the trace for specific state settings
- View, edit, and override shaders for immediate, real time performance feedback for shader optimization (OpenGL ES 2.0-based applications only)

One important use cases for overrides is to replace an original texture with a smaller version, essentially using compressed textures in an application. Because artwork baking can take time, this feature allows checking what performance improvement can be anticipated, once a compressed texture is in place.

8.1.2 Grapher mode

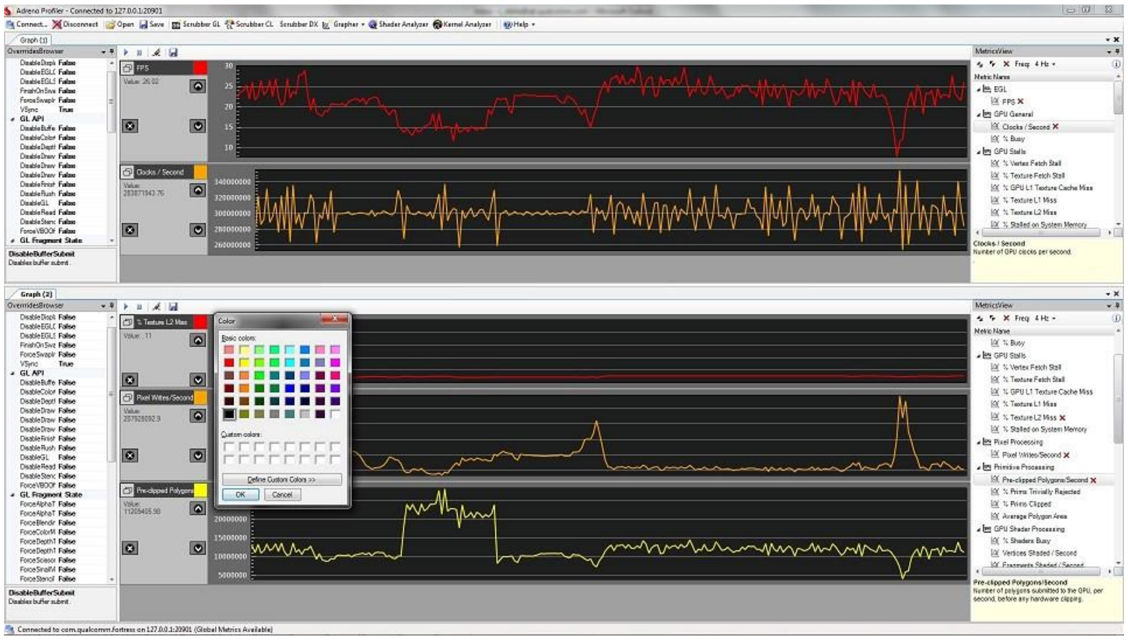

Figure 8-3 Adreno profiler: grapher mode

This mode plots real time performance data streaming from the embedded graphics driver, including:

- GPU
    - Average triangle area
    - Clock rate
    - Detailed instruction counts, e.g., ALU, shader branching, and texture fetching
    - Frames per second
    - Primitive and fragment throughput at each pipeline stage
    - Stalls, e.g., texel fetch and general purpose registers (GPRs)
    - Texture cache misses
    - Overdraw
- System
    - A feedback notification when there has been an expensive driver operation, such as a resolve for when a blocking pipeline is flushed
    - Global and per-context GPU activity
    - Metrics that represent either GPU activity for the current context or Global GPU activity across all contexts

While capturing performance metrics, it is possible to also override the operation of the embedded graphics driver in many ways, e.g.:

- Overriding texture parameter
- Replacing shaders
- Switching blending on and off

8.1.3 Shader analyzer mode

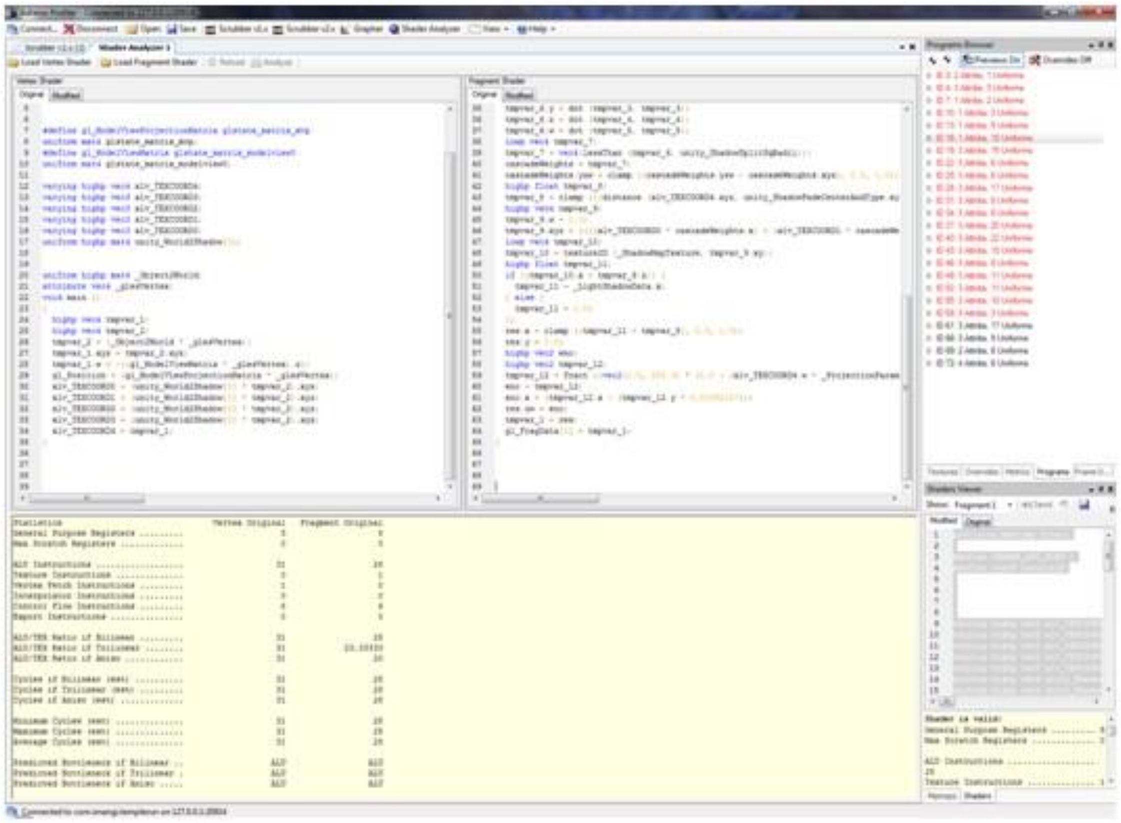

Figure 8-4 Adreno profiler: shader analyzer mode

In this mode, the developer can paste fragment and vertex shader code into the tool, which reports on important metrics that could affect shader execution time. These include, but are not limited to:

- Number of GPRs used by the shaders
- Number of ALUs used by the shaders

8.2 Adreno SDK

8.2.1 Adreno texture compression and visualization tool

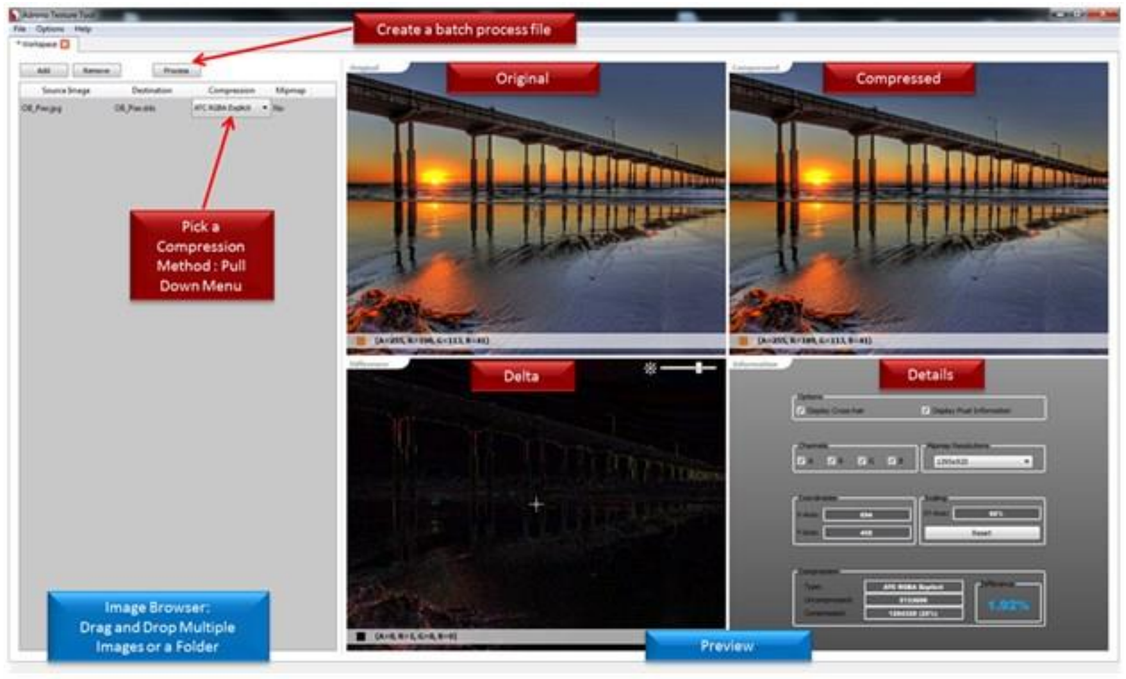
Create a batch process file
Original
Compressed
Pick a Compression Method : Pull Down Menu
Delta
Details
Image Browser: Drag and Drop Multiple Images or a Folder
Preview

Figure 8-5 Adreno texture tool

The Adreno Texture Compression and Visualization Tool enables the compression of textures into the following formats supported by Adreno GPUs:

- 3Dc (single-component)
- 3Dc (two-component)
- ASTC LDR
- ASTC HDR
- ATC RGBA (explicit)
- ATC RGBA (interpolated)
- EAC (single-component)
- EAC (two-component)
- ETC1 RGB8
- ETC2 RGBA8
- ETC2 RGB8
- ETC2 RGB8 Punchthrough Alpha
- S3TC (DXT1 RGBA)

- S3TC (DXT3 RGBA)
- S3TC (DXT5 RGBA)

The application allows the user to:

- Visually compare the looks of the compressed texture with the original noncompressed version
- Zoom into the textures for a detailed view
- Run a best-fit compression analysis based on texture size or quality
- Visualize each color channel separately, including alpha (if available)
- Have multiple workspaces open at the same time
- Save texture workspaces that can be processed within an art pipeline
- Generate mipmaps in a texture
- Save files out to open-standard .ktx and .dds formats
- Set up projects for batch processing, so multiple textures can be compressed in a single run

Also included is a 32- and 64-bit Photoshop plug-in that can be used to load compressed texture formats.

8.2.2 Adreno texture converter

The Compression and Visualization Tool allows on-demand visual inspection of the compressed data. This is useful for deciding which texture compression format should be used for a particular image, but it has limited use in an authoring pipeline where the assets:

- May change in the future; using a GUI to regularly compress hundreds of updated textures is cumbersome
- May need to be compressed into a number of different internal formats, e.g., due to portability reasons
- May need to be stored in a noncompressed internal format of a lower precision than the original asset, e.g., if wanting to compress a RGB888 texture down to a RGB565 format
- Need to be compressed during runtime, taking platform-specific capabilities into consideration; unlike OpenGL, OpenGL ES does not support runtime texture compression and the application must take care of the encoding process if necessary

The Adreno Texture Converter is a dynamic library provided to address these concerns. It allows a function called Qonvert, which takes two arguments describing source and destination image properties. This function performs the conversion and fills the destination with compressed data.

The Adreno Texture Converter can perform the following tasks:

- Downscale the source image if the destination image must be smaller than the original
- Flip source and destination image horizontally and vertically
- Swizzle source and destination image components
- Use nondefault line stride for source and destination image data
- Transform source data into a normal map

The following filters can be used to downscale the source image:

- Bicubic
- Bilinear
- Kaiser window
- Mean
- Nearest-neighbor

The following algorithms can be used to generate normal maps from input images:

- Prewitt operator
- Roberts cross operator
- Sobel operator

Table 8-1 lists the formats that can be specified for both source and destination images.

Table 8-1 Texture converter supported formats

Category	Texture format
16-bit floating-point formats	R16F
	RG16F
	RGB16F
	RGBA16F
Alpha formats	Alpha_16F
	Alpha_32F
	Alpha_8
Compressed formats	3DC X
	3DC XY
	ASTC HDR
	ASTC LDR
	ATC RGB
	ATC RGBA (explicit alpha)
	ATC RGBA (interpolated alpha)
	EAC R (signed)
	EAC R (unsigned)
	EAC RG (signed)
	EAC RG (unsigned)
	ETC1 RGB8
	ETC2 RGB8
	ETC2 sRGB8
	ETC2 RGBA8
	ETC2 RGB8 Punchthrough Alpha1
	ETC2 sRGB8 Alpha8
	ETC2 sRGB8 Punchthrough Alpha1
	DXT1 RGB

Category	Texture format
	DXT1 RGBA
	DXT3 RGBA
	DXT5 RGBA
Depth formats	Depth_16
	Depth_24
	Depth_32
Depth+Stencil formats	Depth_24_Stencil_8
Luminance formats	Luminance_16F
	Luminance_32F
	Luminance_8
Luminance+Alpha formats	Luminance_16F_Alpha_16F
	Luminance_32F_Alpha_32F
	Luminance_8_Alpha8
32-bit floating-point formats	R32F
	RG32F
	RGB32F
	RGBA32F
Normalized signed fixed-point formats	RG_S88
Normalized unsigned fixed-point formats	BGR565
	RGB565
	BGRA8888
	BGRA5551
	BGRx8888
	BGRA4444
	RGBA4444
Packed formats	RGB9E5
	RGB11_11_10F
Signed integer formats	R16I
	R32I
	R8I
	RG16I
	RG32I
	RG8I
	RGB16I
	RGB32I
	RGB8I
	RGB10_A2I
	RGBA16I
	RGBA32I
	RGBA8I
Unsigned integer formats	R16UI
	R32UI

Category	Texture format
	R8UI
	RG16UI
	RG32UI
	RG8UI
	RGB10A2UI
	RGB16UI
	RGB32UI
	RGB8UI
	R2GBA10UI
	RGB5A1UI
	RGBA16UI
	RGBA32UI
	RGBA8UI
YUV formats	AYUV_32
	I420_12
	I444_24
	NV12_12
	NV21_12
	UYVY_16
	YUYV_16
	YV12_12

Library binaries, headers, and detailed documentation can be found in the Adreno SDK.

9 Optimizing Applications

9.1 Shader tips

This section presents various tips and tricks to help optimize an OpenGL ES application on Adreno architectures.

9.1.1 Compile and link during initialization

The compilation and linking of shaders is a time-consuming process. It is expensive compared to other calls in OpenGL ES. It is recommended that shaders are loaded and compiled during initialization, and that glUseProgram is then invoked to switch between shaders as necessary during the rendering phase.

For OpenGL ES 2.0, ES 3.0, and ES 3.1 contexts, the use of blob binaries is recommended. After compiling and linking a program object, it is possible to retrieve the binary representation, or blob, using one of the following functions:

- glGetProgramBinaryOES – If using an OpenGL ES 2.0 context with the GL_OES_get_program_binary extension available
- glGetProgramBinary – If using an OpenGL ES 3.0 or 3.1 context (core functionality)

The blob can then be saved to persistent storage. The next time the application is launched, it is not necessary to recompile and relink the shader. Instead, read the blob from persistent storage and load it directly into the program object using glProgramBinaryOES or glProgramBinary. This can significantly speed up application launch times.

Important

Many OpenGL ES implementations have a habit of incorporating build time GL state into program objects when they are linked. If that program is then used for a draw call issued in the context of a different GL state configuration, then the OpenGL ES implementation is obliged transparently to rebuild the program on-the-fly. This behavior is legal in the OpenGL ES specification. However, it can cause serious problems for developers. It often takes a significant amount of time to rebuild the program object, which can lead to severe frame drops. The rebuild was not requested by the application, so the delay is unexpected and the reason for it may not be apparent.

On Adreno platforms, this is not an issue. The Adreno drivers never recompile shaders. It is safe to assume that program objects will never be rebuilt other than at a specific request.

9.1.2 Use built-ins

Built-in functions are an important part of the OpenGL ES Shading Language specification and should always be used in preference to writing custom implementation. These functions are often optimized for specific shader profiles and for the capabilities of the hardware for which the shader was compiled. As a result, they will usually be faster than any other implementation.

The built-in functions are described in the OpenGL ES Shading Language specification at https://www.khronos.org/registry/gles/specs/3.1/GLSL_ES_Specification_3.10.pdf. They are also covered in the OpenGL ES Reference Pages at https://www.khronos.org/opengles/sdk/docs/man31/, which clarifies Shader Language version support for each of the functions.

9.1.3 Use the appropriate data type

Using the most appropriate data type in code can enable the compiler and driver to optimize code, including the pairing of shader instructions.

Using a vec4 data type instead of float could prevent the compiler from performing aforementioned optimizations. Small mistakes can have a large impact on performance.

Another example is the following code should take a single instruction slot:

```
int4 ResultOfA(int4 a)
{      return a + 1; }
```

Now suppose a slight error is introduced into the code. For the example, the floating point constant value 1.0 is used, which is not the appropriate data type.

```
int4 ResultOfA(int4 a)
{      return a + 1.0;
}
```

The code could now consume eight instruction slots. The variable *a* is converted to vec4, then, the addition is done in floating point. Finally, the result is converted back to the return type int4.

9.1.4 Reduce type casting

It is also recommended to reduce the number of type cast operations performed. The following code might be suboptimal:

```
uniform sampler2D ColorTexture;
in       vec2      TexC;
vec3 light(in vec3 amb, in vec3 diff)
{
    vec3 Color = texture(ColorTexture, TexC);
    Color *= diff + amb;
return Color; }
(..)
```

Here, the call to the texture function returns a vec4. There is an implicit type cast to vec3, which requires one instruction slot. Changing the code as follows might reduce the instruction numbering by one:

```
uniform sampler2D ColorTexture;
in      vec2      TexC;
vec4 light(in vec4 amb, in vec4 diff)
{
    vec4 Color = texture(Color, TexC);
    Color *= diff + amb;
return Color; }
```

9.1.5 Pack scalar constants

Packing scalar constants into vectors consisting of four channels substantially improves the hardware fetch effectiveness. In the case of an animation system, this increases the number of available bones for skinning.

Consider the following code:

```
float scale, bias;
vec4 a = Pos * scale + bias;
```

By changing the code as follows it might take one less instruction, because the compiler can optimize the line to a more efficient instruction (mad):

```
vec2 scaleNbias;
vec4 a = Pos * scaleNbias.x + scaleNbias.y;
```

9.1.6 Keep shader length reasonable

Excessively long shaders can be inefficient. If there is a need to include a large number of instruction slots in a shader relative to the number of texture fetches, consider splitting the algorithm into several parts. Values that are generated by one part of the algorithm for later reuse by another part can be stored into a texture and later retrieved via a texture fetch. However, this approach could be expensive in terms of memory bandwidth. Usage of trilinear, anisotropic filtering, wide texture formats, 3D and cube map textures, texture projection, texture lookup with gradients of different LoD, or gradients across a pixel quad may also increase texture sampling time and reduce the overall benefit.

9.1.7 Sample textures in an efficient way

To avoid texture stalls, follow these rules:

- Avoid random access – Hardware operates on blocks of 2x2 fragments, so the shaders are more efficient if they access neighboring texels within a single block.
- Avoid 3D textures – Fetching data from volume textures is expensive owing to the complex filtering that needs to be performed to compute the result value.

- Limit the number of textures sampled from shaders – Usage of four samplers in a single shader is acceptable, but accessing more textures in a single shader stage could lead to performance bottlenecks.
- Compress all textures – This allows better memory usage, translating to a lower number of texture stalls in the rendering pipeline.
- Consider using mipmaps – Mipmaps help to coalesce texture fetches and can help improve performance at the cost of increased memory usage.

In general, trilinear and anisotropic filtering are much more expensive than bilinear filtering, while there is no difference in cost between nearest and bilinear filtering.

Texture filtering can influence the speed of texture sampling. A bilinear texture lookup in a 2D texture on a 32-bit format costs a single cycle. Adding trilinear filtering doubles that cost to two cycles. Mipmap clamping may reduce this to bilinear cost though, so the average cost might be lower in real-world cases. Adding anisotropic filtering multiplies with the degree of anisotropy. That means a 16x anisotropic lookup can be 16 times slower than a regular isotropic lookup. However, because anisotropic filtering is adaptive, this hit is taken only on fragments that require anisotropic filtering. It could be only a few fragments in all. A rule of a thumb for real-world cases is that anisotropic filtering is, on average, less than double the cost of isotropic.

Cube map texture and projected texture lookups do not incur any extra cost, while shader-specific gradients, based on the dFdx and dFdy functions, cost an extra cycle. This means a regular bilinear lookup that normally takes one cycle will take two with shader-specific gradients.

NOTE: These shader-specific gradients cannot be stored across lookups. If a texture lookup is done again with the same gradients in the same sampler, it will incur the one cycle hit again.

9.1.8 Threads in flight/dynamic branching

Branching is crucial for the performance of the shader. Every time the branch encounters *divergence*, or where some elements of the thread branch one way and some elements branch in another, both branches will be taken with predicates using NULL out operations for the elements that do not take a given branch. This is true only if the data is aligned in a way that follows those conditions, which is rarely the case for fragment shaders.

There are three types of branches, listed in order from best performance to worst on Adreno GPUs:

- Branching on a constant, known at compile time
- Branching on a uniform variable
- Branching on a variable modified inside the shader

Branching on a constant may yield acceptable performance.

9.1.9 Pack shader interpolators

Shader-interpolated values or varyings require a GPR to hold data being fed into a fragment shader. Therefore, minimize their use.

Use constants where a value is uniform. Pack values together as all varyings have four components, whether they are used or not. Putting two vec2 texture coordinates into a single vec4 value is a common practice, but other strategies employ more creative packing and on-the-fly data compression. OpenGL ES 3.0 and ES 3.1 introduce various intrinsic functions to carry out packing operations.

9.1.10 Minimize usage of shader GPRs

Minimizing the usage of GPRs can be an important means of optimizing performance. Inputting simpler shaders to the compiler helps guarantee optimal results. Modifying GLSL to save even a single instruction can save a GPR sometimes. Not unrolling loops can also save GPRs, but that is up to the shader compiler. Always profile shaders to make sure the final solution chosen is the most efficient one for the target platform. Unrolled loops tend to put texture fetches toward the shader top, resulting in a need for more GPRs to hold the multiple texture coordinates and fetched results simultaneously.

For example, if unrolling the loop presented below:

```
for (i = 0; i < 4; ++i) {      diffuse +=
ComputeDiffuseContribution(normal,
light[i]); }
```

The code snippet would be replaced with:

```
diffuse += ComputeDiffuseContribution(normal, light[0]);
diffuse += ComputeDiffuseContribution(normal, light[1]);
diffuse += ComputeDiffuseContribution(normal, light[2]);
diffuse += ComputeDiffuseContribution(normal, light[3]);
```

9.1.11 Minimize shader instruction count

The compiler optimizes specific instructions, but it is not automatically efficient. Analyze shaders to save instructions wherever possible. Saving even a single instruction is worth the effort.

9.1.12 Avoid uber-shaders

Uber-shaders combine multiple shaders into a single shader that uses static branching. Using them makes sense if trying to reduce state changes and batch draw calls. However, this often increases GPR count, which has an impact on performance.

9.1.13 Avoid math on shader constants

Almost every shipped game since the advent of shaders has spent instructions performing unnecessary math on shader constants. Identify these instructions in shaders and move those calculations off to the CPU. It may be easier to identify math on shader constants in the postcompiled microcode.

9.1.14 Avoid discarding pixels in the fragment shader

Some developers believe that manually discarding, also known as *killing*, pixels in the fragment shader boosts performance. The rules are not that simple for two reasons:

- If some pixels in a thread are killed and others are not, the shader still executes.
- It depends on how the shader compiler generates microcode.

In theory, if all pixels in a thread are killed, the GPU will stop processing that thread as soon as possible. In practice, discard operations can disable hardware optimizations.

If a shader cannot avoid discard operations, attempt to render geometry, which depends on them after opaque draw calls.

9.1.15 Avoid modifying depth in fragment shaders

Similar to discarding fragments, modifying depth in the fragment shader can disable hardware optimizations.

9.1.16 Avoid texture fetches in vertex shaders

Adreno is based on a unified shader architecture, which means the vertex processing performance is similar to the fragment processing performance. However, for optimal performance, it is important to ensure that texture fetches in vertex shaders are localized and always operate on compressed texture data.

9.1.17 Break up draw calls

If a shader is heavy on GPRs and/or heavy on texture cache demands, increased performance can result from breaking up the draw calls into multiple passes. Whether or not the results will be positive is hard to predict, so using real-world measurements both ways is the best method to decide. Ideally, a two-pass draw call would combine its results with simple alpha blending, which is not heavy on Adreno GPUs because of the GMEM.

Some developers may consider using a true deferred rendering algorithm, but that approach has many drawbacks, e.g., the GMEM must be resolved for a previous pass to be used as input to a successive pass. Because resolves are not free, it is a performance cost that must be recouped elsewhere in the algorithm.

9.1.18 Use medium precision where possible

On Adreno, operations on 16-bit floating point (mediump) values are twice as fast as on 32-bit (highp) values, and the power consumption is reduced by half. QTI recommends setting the default precision to mediump and promoting only those values that require higher precision.

However, there may be situations when highp must be used for certain varyings, e.g., texture coordinates, in shaders. These situations can be handled with a conditional statement and a preprocessor-based macro definition, as follows:

```
precision mediump float;
#ifdef GL_FRAGMENT_PRECISION_HIGH
    #define NEED_HIGHP highp
#else
    #define NEED_HIGHP mediump
#endif
varying            vec2 vSmallTexCoord;
varying NEED_HIGHP vec2 vLargeTexCoord;
```

9.1.19 Favor vertex shader calculations over fragment shader calculations

Typically, vertex count is significantly less than fragment count. It is possible to reduce GPU workload by moving calculations from the fragment shader to the vertex shader. This helps to eliminate redundant computations.

9.1.20 Measure, test, and verify results

Finding bottlenecks is necessary for optimization, whether the application is vertex bound, fragment bound, or texture fetch bound. Measure performance before attempting to make the code faster. Use tools to take these measurements, e.g., the Adreno Profiler or even software timers.

Do not assume something runs faster based solely on intuition. When code is modified to perform better, it can disable compiler/hardware optimizations that are more beneficial. Always measure timing before and after changes to assess the impact of modifications performed for the sake of optimization.

9.1.21 Prefer uniform buffers over shader storage buffers

As long as read-only access is sufficient for needs and the amount of space Uniform Buffers offer is enough, always prefer them over Shader Storage Buffers. They are likely to perform better under the Adreno architecture. This is true if the Uniform Buffer Objects are statically indexed in GLSL and are small enough that the driver or compiler is able to map them into the same hardware constant RAM that is used for the default uniform block uniforms.

9.1.22 Invalidate frame buffer contents as early as possible

An application should use glInvalidateFramebuffer and glInvalidateSubFramebuffer API calls to inform the driver that it is free to drop the contents (or regions thereof) of the current draw frame buffer. This is important for tiled rendering modes, because these hints can be used by the driver to reduce the amount of work the hardware has to perform.

9.1.23 Optimize vertex buffer object updates

If needing to modify Vertex Buffer Object contents on-the-fly when rendering a frame, be sure to batch all the VBO updates (glBufferSubData calls) before issuing any draw calls that use the modified VBO region. If using multiple VBOs, batch the updates for all the VBOs first, and then issue all the draw calls.

Failure to follow these recommendations can cause the driver to maintain multiple copies of an entire VBO, which results in reduced performance.

9.1.24 Eliminate subpixel triangles during tessellation

Tessellation allows for increased levels of detail and can reduce memory bandwidth and CPU cycles by allowing other game subsystems to operate on low-resolution representations of meshes. However, high levels of tessellation can generate subpixel triangles, which cause poor rasterizer utilization. It is important to utilize distance, screen space size, or other adaptive metrics for computing tessellation factors that avoid subpixel triangles.

9.1.25 Do back-face culling during tessellation

Hardware back-face culling occurs after the tessellation stage, which potentially wastes GPU resources tessellating back-facing primitives. These can be identified in the tessellation control shader stage and culled by setting their edge tessellation factors to 0.

NOTE: Include a slight “fudge” factor in this calculation if displacement mapping will be used in the tessellation evaluation shader stage, as this technique may change the visibility of primitives.

9.1.26 Disable tessellation whenever it is not needed

Whenever possible, disable the tessellation control shader and tessellation evaluation shader stages if the tessellation factor for the mesh would be ~1. This eliminates the use of unnecessary GPU stages during the rendering process.

9.2 Optimize vertex processing

This section describes tips and tricks that can help optimize the way OpenGL ES applications organize vertex data, so that the rendering process can run efficiently on Adreno architectures.

9.2.1 Use interleaved, compressed vertices

For vertex fetch performance, interleaved vertex attributes (“xyz uv | xyz uv | ...”, rather than “xyz | xyz | .. | uv | uv | ..”), work the most efficiently. The throughput is better with interleaved vertices and compressing vertices improves it further. Deferred rendering gives these optimizations an advantage.

For binning pass optimization, consider one array with vertex attributes and other attributes needed to compute position, and another interleaved array with other attributes.

Under OpenGL ES 2.0, the Adreno OpenGL ES implementation supports the GL_OES_vertex_half_float extension, which allows programmers to fill vertex buffers with half float coordinates. This functionality became a core feature of OpenGL ES 3.0. Half float coordinates can be used for texture or normal coordinates where lower precision will not hurt the final imagery, improving the rendering performance.

9.2.2 Use Z-only rendering

The GPU has a special mode to write Z-only pixels at twice the normal rate, e.g., when an application renders to a shadow map. The hardware needs to be told by the driver to enter this special rendering mode and, without a specific OpenGL state, the driver needs hints from the application.

Using an empty fragment shader and disabling the frame buffer write masks are good hints.

Some developers take advantage of double-speed, Z-only rendering by laying down a Z-prepass before rendering the main scene. Performance tests must still be run to determine if this is beneficial on the Adreno.

9.2.3 Consider geometry instancing

Geometry instancing is the practice of rendering multiple copies of the same mesh or geometry in a scene at once. This technique is used primarily for objects like trees, grass, or buildings that can be represented as repeated geometry without appearing unduly repetitive. However, geometry instancing can also be used for characters.

Although vertex data is duplicated across all instanced meshes, each instance could have other differentiating parameters, e.g., color, transforms, or lighting, changed to reduce the appearance of repetition.

As shown in Figure 9-1, all barrels in the scene could use a single set of vertex data that is instanced multiple times instead of using unique geometry for each one.

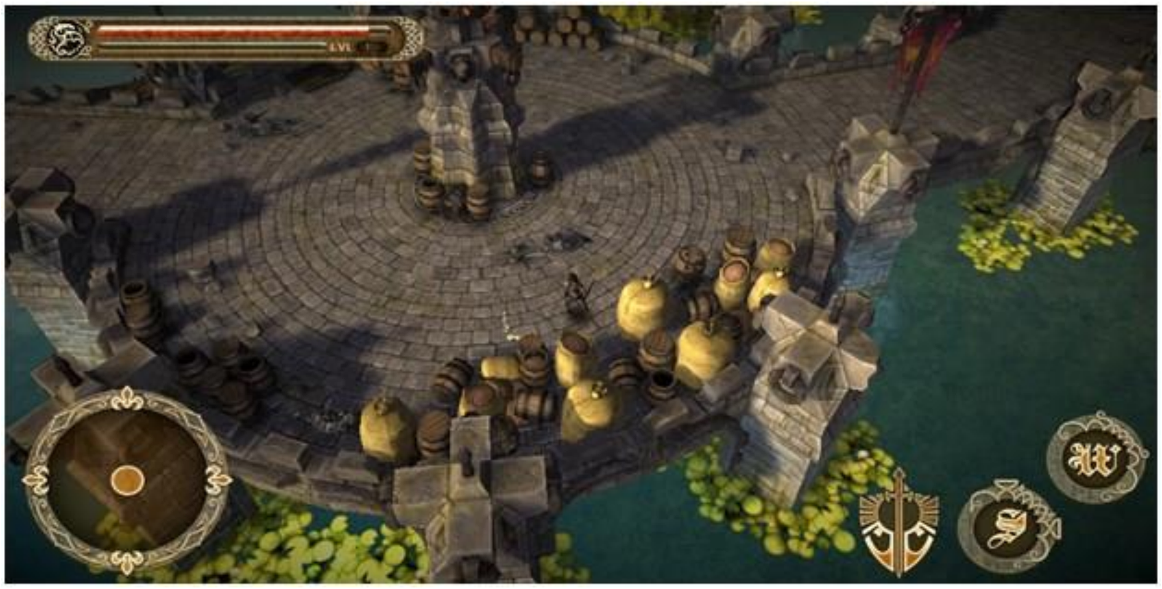

Figure 9-1 Geometry instancing for drawing barrels

Geometry instancing offers a significant savings in memory usage. It allows the GPU to draw the same geometry multiple times in a single draw call with different positions, while storing only a single copy of the vertex data, a single copy of the index data, and an additional stream containing one copy of a transform for each instance. Without instancing, the GPU would have to store multiple copies of the vertex and index data. See the Khronos OpenGL ES 3.0 API specifications for more details.

Below are two APIs that enable rendering multiple instances:

```
void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei
primcount);
```

```
void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const
void* indices, GLsizei primcount);
```

9.2.4 Select the best vertex format

The Adreno GPU provides hardware support for the following vertex formats:

- GL_BYTE and GL_UNSIGNED_BYTE
- GL_SHORT and GL_UNSIGNED_SHORT
- GL_FIXED
- GL_FLOAT
- GL_HALF_FLOAT
- GL_INT_2_10_10_10_REV and GL_UNSIGNED_INT_2_10_10_10_REV

When preparing vertex data sets, for optimal performance always use the vertex format that provides a satisfactory level of precision and also takes the least amount of space.

9.2.5 Use indirect indexed draw calls

Introduced in OpenGL ES 3.1, indirect draw calls move the draw call overhead from the CPU to the GPU. This provides a significant performance benefit over the regular draw calls under the Adreno architecture.

If an application is targeting the latest OpenGL ES version, consider using this new feature for improved rendering efficiency, e.g., if the renderer is based on the concept of a scene graph, it is possible to cache the draw call arguments necessary to render the mesh nodes in a buffer object store during loading time. The store can then be used during rendering time as an input to the glDrawArraysIndirect or glDrawElementsIndirect functions.

9.3 Texture compression strategies

Compressing textures can significantly improve the performance and load time for graphics applications, since it reduces texture memory and bus bandwidth use. Unlike desktop OpenGL, OpenGL ES does not provide the necessary infrastructure for the application to compress textures at runtime, so texture data needs to be authored off-line.

Significant texture compression formats supported by Adreno GPUs are:

- ATC – Proprietary Adreno texture compression format supporting RGB and RGBA component layouts
- ETC – Standard OpenGL ES 2.0 texture compression format supporting the RGB component layout only
- ETC2 – Standard OpenGL ES 3.0 texture compression format supporting R, RG, RGB, and RGBA component layouts, as well as sRGB texture data
- ASTC – Texture compression format introduced in the AEP, offering remarkably low levels of image degradation given the compression ratios achieved

Tip

QTI recommends the following strategy for selecting a texture compression format:

1. Use ASTC compression if it is available.
2. Otherwise, use ETC2 compression if available.
3. Otherwise, select the compression format as follows:
   a. ATC if using alpha
   b. ETC if not using alpha

ASTC is a newer format and may not be supported or optimized on all content creation pipelines. Also the sRGB formats for ASTC are more efficiently handled on Adreno hardware than the RGBA formats.

The following are popular methods for converting textures to Adreno hardware:

- On-device conversion
- Prepackaging
- Downloading

On-device conversion involves a time-consuming one-time conversion of texture assets that occurs when the game starts up. Prepackaging the correct textures results in the most optimized solution but requires alternative versions of the APK for each GPU. Downloading requires GPU detection and an internet connection but allows for even more control on the exact texture format for each GPU. In either case, the first step is to create the compressed textures.

Texture data can be compressed to any of these texture compression formats using the Adreno Texture Compression and Visualization Tool or Adreno Texture Converter Tool, both included in the Adreno SDK. There is a Compressed Texture tutorial in the SDK, which presents how to use compressed textures in OpenGL ES applications.

The effectiveness of these compression formats is throughout Figure 9-2 through Figure 9-15, consisting of one diffuse and one object-space normal RGB texture, compressed using the Adreno Texture Tool. The figures show the original textures and the compressed versions using each of the compression formats. For each example, there is a difference image showing the absolute difference between the original and the compressed version.

9.3.1 Diffuse texture tests

The diffuse texture used for this test is shown in Figure 9-2.

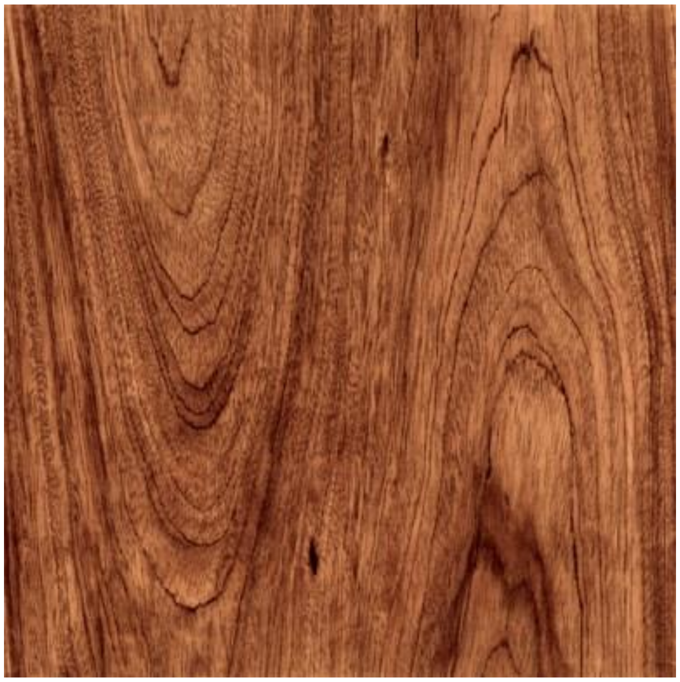

Figure 9-2 Diffuse texture used for the test

9.3.1.1 ATC compression

Figure 9-3 and Figure 9-4 show the use of the GL_ATC_RGB_AMD internal format.

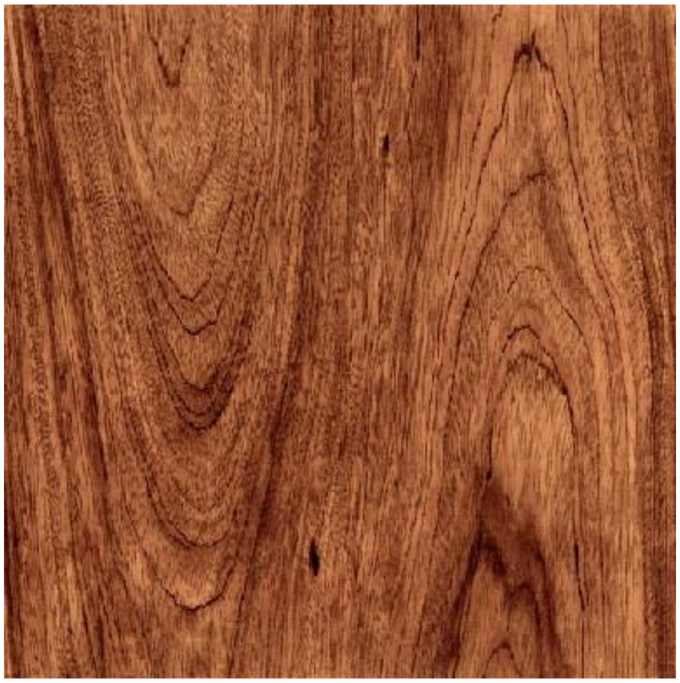

Figure 9-3 ATC compression result for GL_ATC_RGB_AMD

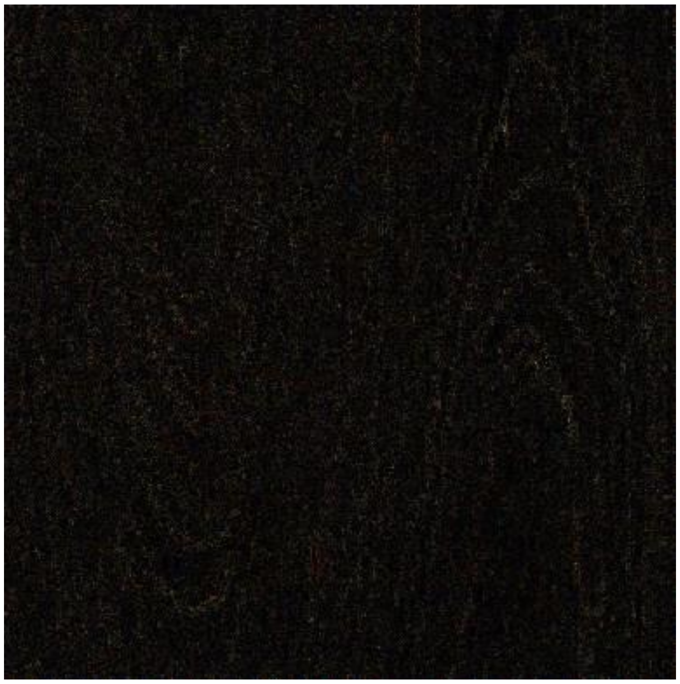

Figure 9-4 Difference between noncompressed and ATC compressed versions for GL_ATC_RGB_AMD

9.3.1.2 ETC1 compression

Figure 9-5 and Figure 9-6 show the use of the GL_ETC1_RGB8_OES internal format.

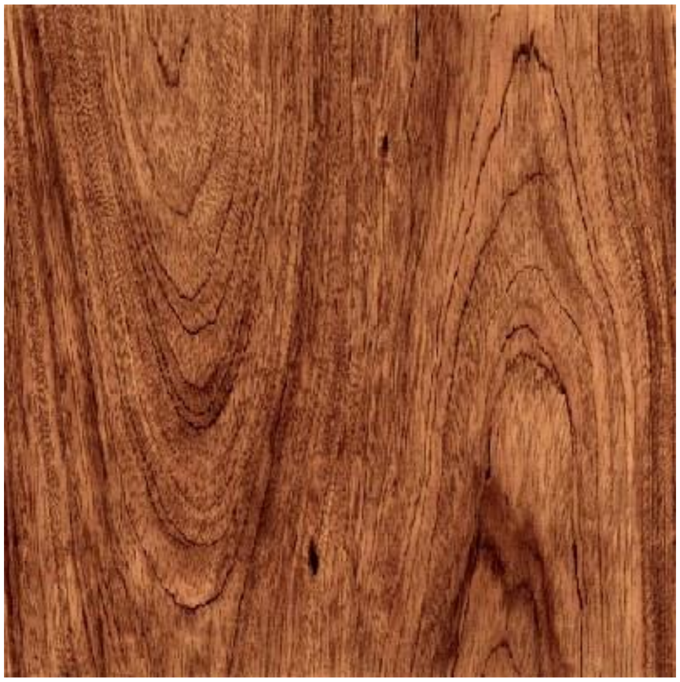

Figure 9-5 ETC1 compression result for GL_ETC1_RGB8_OES

Figure 9-6 Difference between noncompressed and ETC1-compressed versions for GL_ETC1_RGB8_OES

9.3.1.3 ETC2 compression

Figure 9-7 and Figure 9-8 show the use of the GL_COMPRESSED_RGB8_ETC2 internal format.

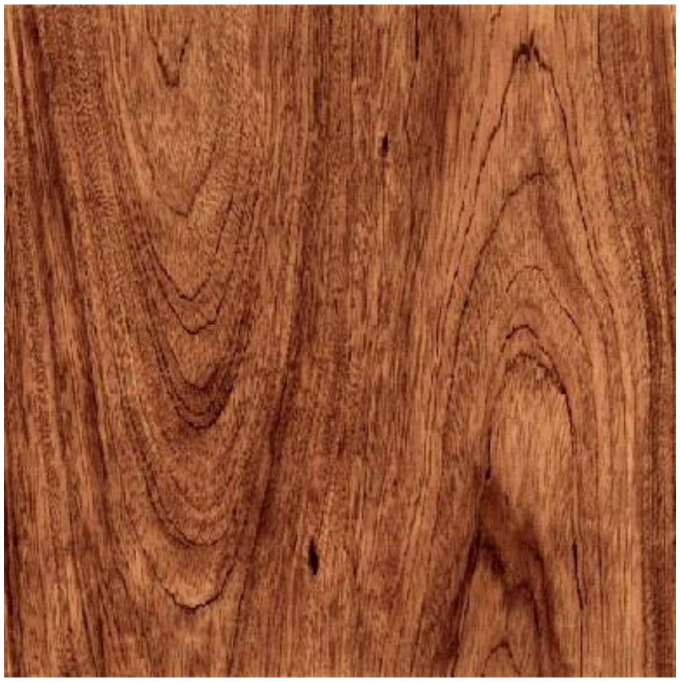

Figure 9-7 ETC2 compression result for GL_COMPRESSED_RGB8_ETC2

Figure 9-8 Difference between noncompressed and ETC2-compressed versions for GL_COMPRESSED_RGB8_ETC2

9.3.2 Normal texture tests

The normal texture used for this test is shown in Figure 9-9.

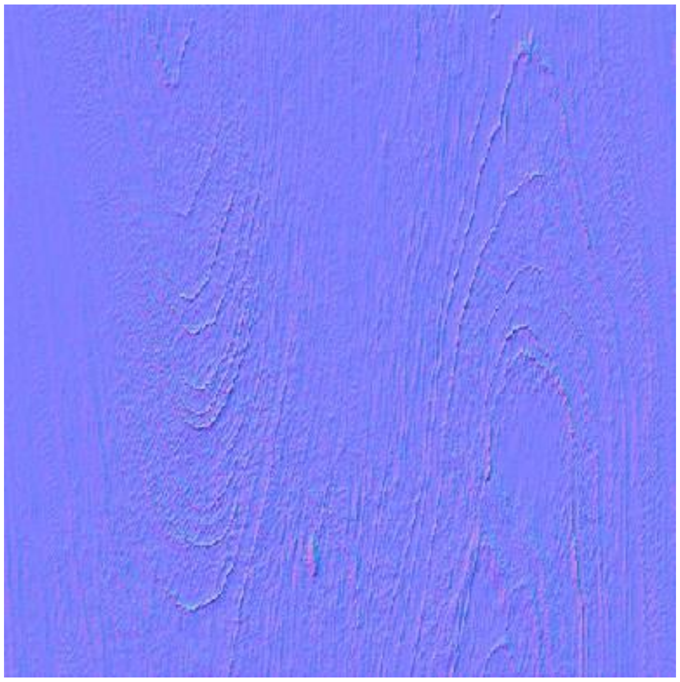

Figure 9-9 Normal texture used for the test

9.3.2.1 ATC compression

Figure 9-10 and Figure 9-11 show the use of the GL_ATC_RGB_AMD internal format.

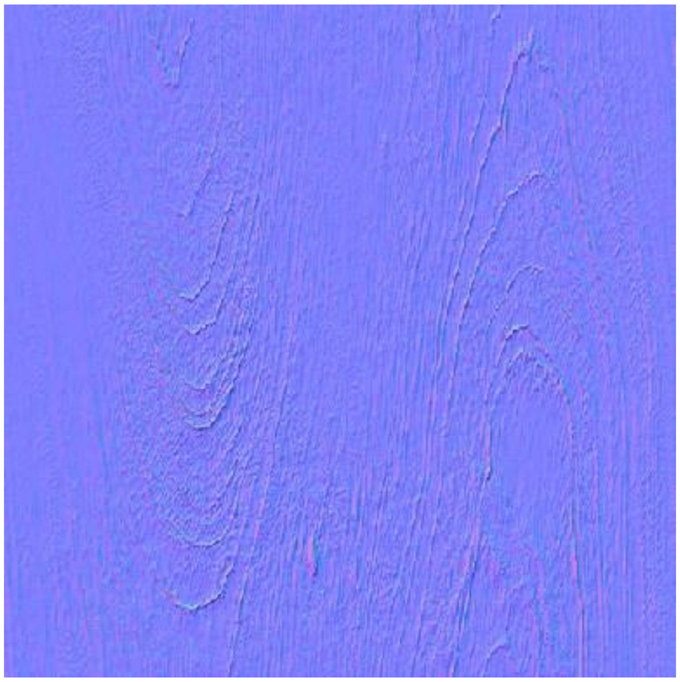

Figure 9-10 ATC compression result for GL_ATC_RGB_AMD

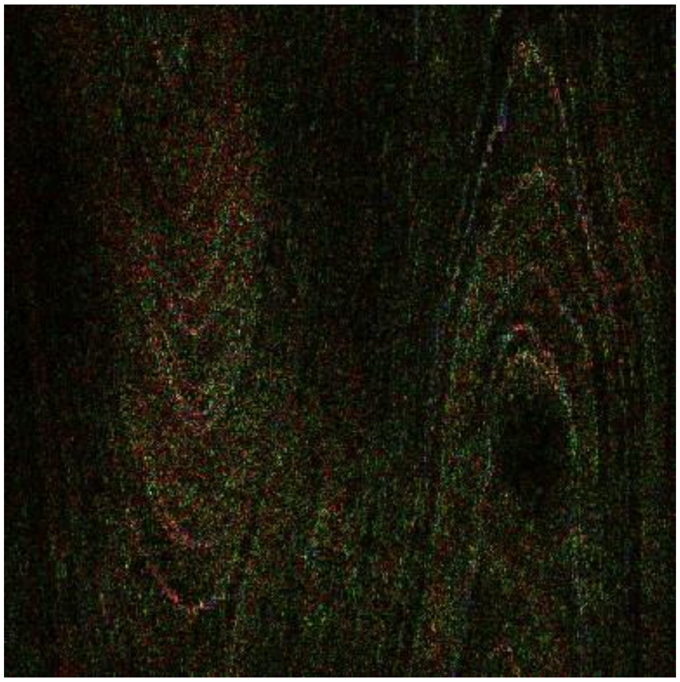

Figure 9-11 Difference between noncompressed and ATC-compressed versions for GL_ATC_RGB_AMD

9.3.2.2 ETC1 compression

Figure 9-12 and Figure 9-13 show the use of the GL_ETC1_RGB8_OES internal format.

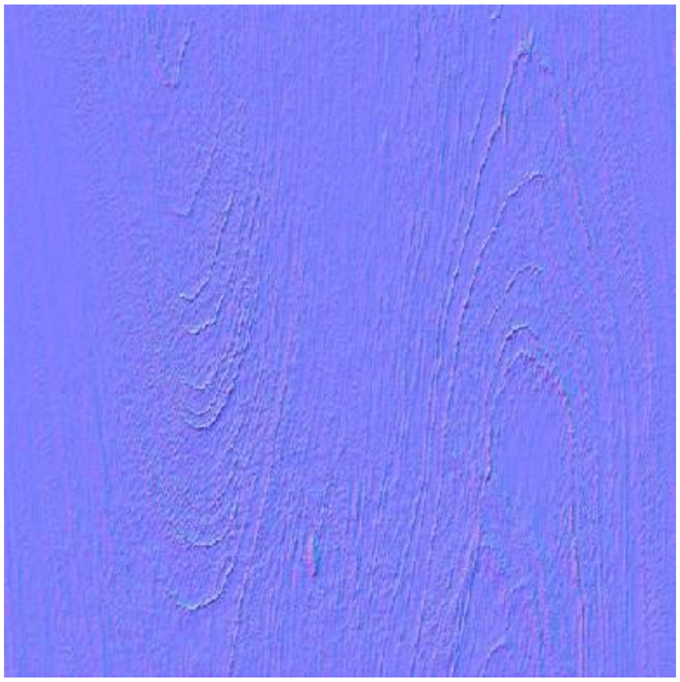

Figure 9-12 ETC1 compression result for GL_ETC1_RGB8_OES

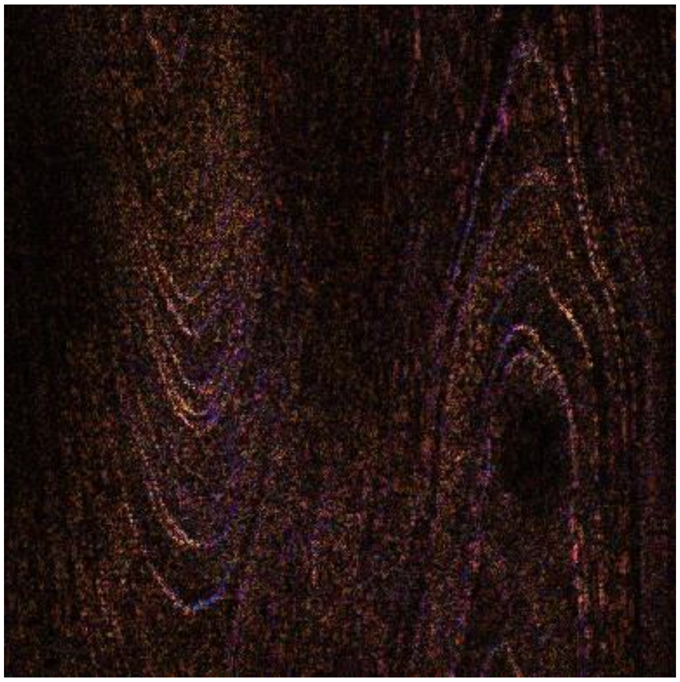

Figure 9-13 Difference between noncompressed and ETC1-compressed versions for GL_ETC1_RGB8_OES

9.3.2.3 ETC2 compression

Figure 9-14 and Figure 9-15 show the use of the GL_COMPRESSED_RGB8_ETC2 internal format.

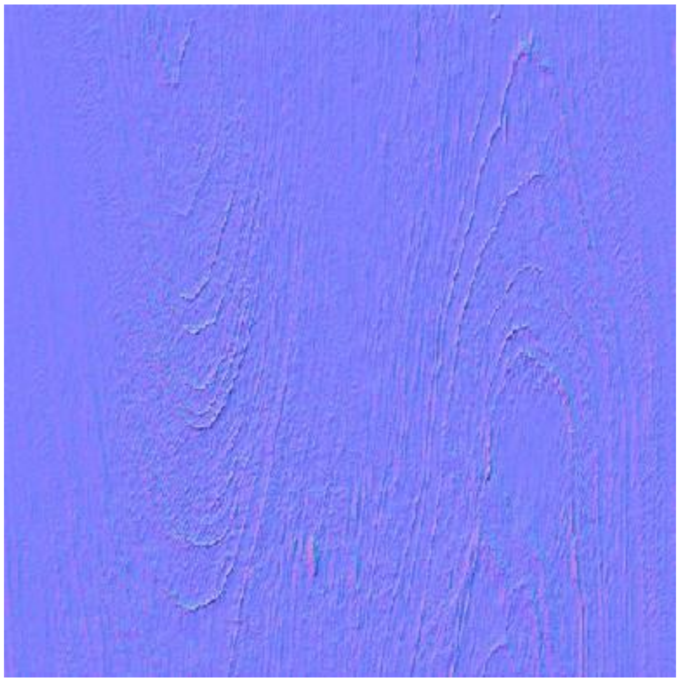

Figure 9-14 ETC2 compression result for GL_COMPRESSED_RGB8_ETC2

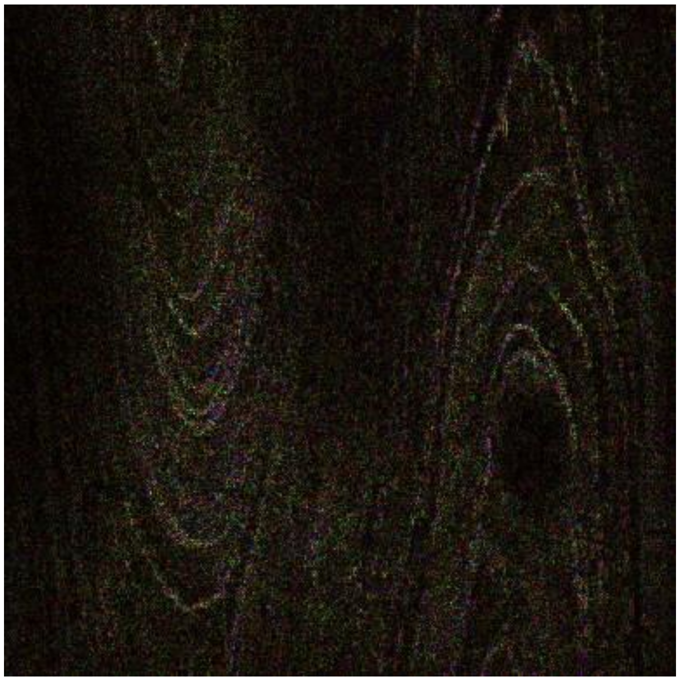

Figure 9-15 Difference between noncompressed and ETC2-compressed versions for GL_COMPRESSED_RGB8_ETC2

9.4 Bandwidth optimization

OpenGL ES applications can suffer from the bottleneck of being memory-bandwidth limited. This is a manifestation of the physical limitation of how much data the GPU can access within a given timeframe. The rate is not constant and varies according to a number of factors, including but not limited to:

1. Location of the data – Is it stored in RAM, VRAM, or one of the GPU caches?
2. Type of access – Is it a read or a write operation? Is it atomic? Does it have to be coherent?
3. Viability of caching the data – Can the hardware cache the data for subsequent operations that the GPU will be carrying out, and would it make sense to do this?

Cache misses can cause applications to become bandwidth-limited, which causes significant performance drops. These cache misses are often caused when applications draw or generate a large number of primitives, or when shaders need to access a large number of locations within textures.

There are two measures to take to minimize the problem of cache misses:

1. Improve the transfer speed rate – Ensure that client-side vertex data buffers are used for as few draw calls as possible; ideally, an application should never use them.
2. Reduce the amount of data the GPU needs to access to perform the dispatch or draw call that is hitting this constraint.

OpenGL ES provides several methods that developers can use to reduce the bandwidth needed to transfer specific types of data.

The first method is compressed texture internal formats, which sacrifice texture quality for the benefit of reduced mipmap size. Many of the compressed texture formats supported by OpenGL ES divide the input image into 4x4 blocks and perform the compression process separately for each block, rather than operating on the image as a whole. While it can seem to be inefficient from the point of view of data compression theory, it does have the advantage of each block being aligned on a 4-pixel boundary. This allows the GPU to retrieve more data with a single fetch instruction, because each compressed texture block holds 16 pixels instead of a single pixel, as in the case with an uncompressed texture. Also, the number of texture fetches can be reduced, provided that the shader does not need to sample texels that are too far apart.

The second method is to use packed vertex data formats. These formats are based on the premise that many vertex data sets will not suffer greatly from a reduction in the precision of their components. It is strongly recommended to use packed vertex data formats wherever possible.

For certain assets whose range span is known in advance, try to map the data onto one of the supported, packed vertex data formats. Taking normal data as an example, it is possible to map XYZ components onto the GL_UNSIGNED_INT_2_10_10_10_REV format by normalizing the 10-bit unsigned integer data range of <0, 1024> onto the floating point range <-1, 1>.

Table 9-1 lists the vertex data formats that can be used in OpenGL ES 2.0, ES 3.0, and ES 3.1 on Adreno hardware. When an OpenGL ES extension must be used to access a given format, it is necessary to use a different identifier for the format—this is also shown in the table.

Table 9-1 Vertex data format support in Adreno architecture

Format name	ES 2	ES 3	ES 3.1
GL_BYTE	Supported	Supported	Supported
GL_FIXED	Supported	Supported	Supported
GL_FLOAT	Supported	Supported	Supported
GL_HALF_FLOAT	Supported[a]	Supported	Supported
GL_INT	Not supported	Supported	Supported
GL_INT_2_10_10_10_REV	Supported[b]	Supported	Supported
GL_SHORT	Supported	Supported	Supported
GL_UNSIGNED_BYTE	Supported	Supported	Supported
GL_UNSIGNED_INT	Not supported	Supported	Supported
GL_UNSIGNED_INT_2_10_10_10_REV	Supported[c]	Supported	Supported
GL_UNSIGNED_SHORT	Supported	Supported	Supported

[a]Use GL_OES_vertex_half_float
[b]Use GL_OES_vertex_type_10_10_10_2
[c]Use GL_OES_vertex_type_10_10_10_2

The third method is to always use indexed draw calls. Always use an index type that is as small as possible while still being able to address all the vertices for the mesh being drawn. This reduces the amount of index data that the GPU needs to access for each draw call, at the expense of slightly more complicated application logic.

9.5 Depth range optimization

A common artifact in games is z-fighting, where two or more primitives have similar values in the depth buffer. The artifact becomes apparent when the camera view changes slightly and fragments from different primitives fight to win the depth test and become visible.

There are several ways to eliminate z-fighting:

- Modify the scene geometry to provide more distance between near planar surfaces (preferred).
- Choose an EGL configuration with a larger depth buffer, e.g., 24 bits vs 16 bits, though performance may be impacted by using a larger buffer.
- Tighten camera depth range (near and far clipping planes) allowing for more precision in the z direction, though this will not help coplanar primitives; consider rendering the scene with multiple ranges, one for near objects and one for far objects.

9.6 Other optimizations

The 3D rendering process is a compute-intensive activity. Screen resolutions are growing larger, with some about to reach Ultra HD resolution. This means that GPUs need to rasterize more fragments within the same fixed time period. Assuming a target frame rate of 30 fps, a game must not spend more than 33 ms on a single frame. If it does, then the number of screen updates per second will drop, and it will become more difficult for users to immerse themselves fully into the game.

Also, the busier the hardware is, the more heat it will generate. If, over an extended period of tim, the GPU is not left with any idle time between frames, the device may become hot and uncomfortable to hold. If the temperature exceeds a certain safety threshold, the device may even automatically reduce the GPU clock frequency to prevent it from overheating. This will further degrade the user experience.

To reduce the load on the rendering hardware, an application can reduce the size of the render target used, e.g., if the native screen resolution is 1080p (1920x1080), it could be rendered to a 720p (1280x720) render target instead. Since the aspect ratio of the two resolutions is identical, the proportions of the image will not be affected. The reduced-size render target will not completely fill the screen, but OpenGL ES provides a fix for this issue.

OpenGL ES 3.0 introduced support for frame buffer blit operations. The contents of a draw buffer can be blit from one frame buffer to another. As part of the blit operation, the API also supports upscaling, which can be used to copy the contents of a texture of a smaller resolution to another texture of a larger resolution. Use upscaling to scale up the reduced-size render target to match the full native display size. The best strategy depends on how intensive the computations are that the GPU has to perform for an application. The upscaling can be done either at the end of the rendering process or at some point in the rendering pipeline; e.g., one approach might be to render the geometry at 1:1 resolution, but apply postprocessing effects using render targets of a slightly lower resolution.

NOTE: Upscaling using a frame buffer blit is faster than the alternative approach of rendering a full screen quad directly to the back buffer, taking the reduced-size render target as a texture input.

Alternatively, control scaling through the Android API:

- For applications written in Java, configure the fixed-size property of the GLSurfaceView instance (available since API level 1). Set the property using the setFixedSize function, which takes two arguments defining the resolution of the final render target.
- For applications written in native code, define the resolution of the final render target using the function ANativeWindow_setBuffersGeometry, which is a part of the NativeActivity class, introduced in Android 2.3 (API level 9).

At every swap operation, the operating system takes the contents of the final render target and scales it up so that it matches the native resolution of the display.

This technique has been used successfully in console games, many of which make heavy demands on the GPU and, if rendering were done at full HD resolution, could be affected by the hardware constraints discussed.

A Gamepad Support

QTI recommends Logitech F710 and Nyko Playpad Pro gamepad controllers.

Table A-1 lists keycode mappings for the Logitech F710, as reported by Android.

Table A-1 Keycode mappings for Logitech F710

Logitech F710 button	Android (ICS) code	Keycode value	Scancode value
A	KEYCODE_BUTTON_A	96	304
B	KEYCODE_BUTTON_B	97	305
BACK	KEYCODE_BUTTON_SELECT	109	314
DPAD DOWN	KEYCODE_DPAD_DOWN	20	0
DPAD LEFT	KEYCODE_DPAD_LEFT	21	0
DPAD RIGHT	KEYCODE_DPAD_RIGHT	22	0
DPAD UP	KEYCODE_DPAD_UP	19	0
LEFT BUTTON	KEYCODE_BUTTON_L1	102	310
LEFT JOYSTICK BUTTON	KEYCODE_BUTTON_THUMBR	106	317
LEFT TRIGGER	(Analog Info)		
LOGITECH	KEYCODE_BUTTON_MODE	110	316
RIGHT BUTTON	KEYCODE_BUTTON_R1	103	311
RIGHT JOYSTICK BUTTON	KEYCODE_BUTTON_THUMBL	107	318
RIGHT TRIGGER	(Analog Info)		
START	KEYCODE_BUTTON_START	108	315
X	KEYCODE_BUTTON_X	99	307
Y	KEYCODE_BUTTON_Y	100	308

B Glossary

Term	Definition
2D array texture	A two-dimensional mipmapped texture type, where each mipmap level can hold multiple 2D images
2D texture	A two-dimensional mipmapped texture type, where each mipmap level can hold a single 2D image
3D texture	A three-dimensional mipmapped texture type, where each mipmap level can hold a set of texture slices defining 3D image data
Alpha blending	The process of blending incoming fragment values with the data already stored in the color buffer, controlled by the alpha component value of the source or the destination, or by both values
Alpha test	A rendering pipeline stage specific to OpenGL ES 1.1, which discards fragments, depending on the outcome of a comparison between the incoming fragment value and a constant reference value; this behavior can easily be reproduced with fragment shaders if using OpenGL ES 2.0 or later
Atomic counter	A special OpenGL ES Shader Language integer variable type; any read/write operations performed on the atomic counter are serialized
Atomic function	A thread safe function, guaranteeing that the value the function operates on will be correctly read from/written to, even when more than one thread is using the function at the same time
Back buffer	Off-screen color buffer used for rendering purposes
Back face culling	An optimization performed by OpenGL ES; suppresses fragment shader invocations for primitives that are not facing the camera
Binding point	An abstract concept with the usage where an OpenGL ES object can be attached to a specific binding point, allowing it to be subsequently accessed through an OpenGL ES function operating on that binding point, or to be used in the rendering process
Blend equation	Rendering context state, which specifies which equation (from a small set defined in the OpenGL ES specification) should be used for the blending process
Blend mode	A rendering mode, which when toggled on, causes incoming fragments to be blended with the existing color data, instead of simply overriding it
Blit operation	Memory copy operation that takes color/depth/stencil attachments of a read frame buffer and transfers their contents to the corresponding attachments of a draw frame buffer; input data can be optionally resized, prior to being transferred
Buffer object	A data container that can be managed by the OpenGL ES API; usually hosted in VRAM
Buffer subrange mapping	The process of mapping a region of storage of a buffer object into process space for read and/or write operations
Buffer swap operation	The act of flipping the back buffer of the default frame buffer with the front buffer