OpenGL® ES Application Developers' Guide for Adreno™ GPUs

Contents

1 About this Document

This document is a guide for developing and optimizing applications for Snapdragon™ S4-based mobile platforms that include the Adreno™ 300 series GPUs (Graphics Processing Units). The Adreno 300 series GPUs, or Adreno 3xx, include, for example, the Adreno 320 GPU, which is embedded within Qualcomm's Snapdragon S4 Pro (APQ8064, MSM8960T) and S4 Prime (MPQ8064) processors.

Adreno 3xx is explained from an application development perspective. Sample code within this document is copyright of QUALCOMM® Incorporated.

We recommend that you use the "OpenGL ES 2.0 Programming Guide" published by Addison Wesley as a reference for developing OpenGL ES 2.0 applications.

2 Understanding the Platform

2.1 Snapdragon: Overview

Snapdragon is one of the most powerful and widely used processors in today's Android and Windows smartphones and tablets. And soon you will see Snapdragon and Adreno in other types of devices like PCs, Smart TVs and set-top boxes. For a list of current commercial devices that include Snapdragon processors, see the Snapdragon website (http://www.qualcomm.com/snapdragon/devices).

Snapdragon processors bring together all the best-in-class mobile components on a single chip, ensuring that Snapdragon-based devices deliver the latest mobile user experiences in an extremely power-efficient, integrated solution.

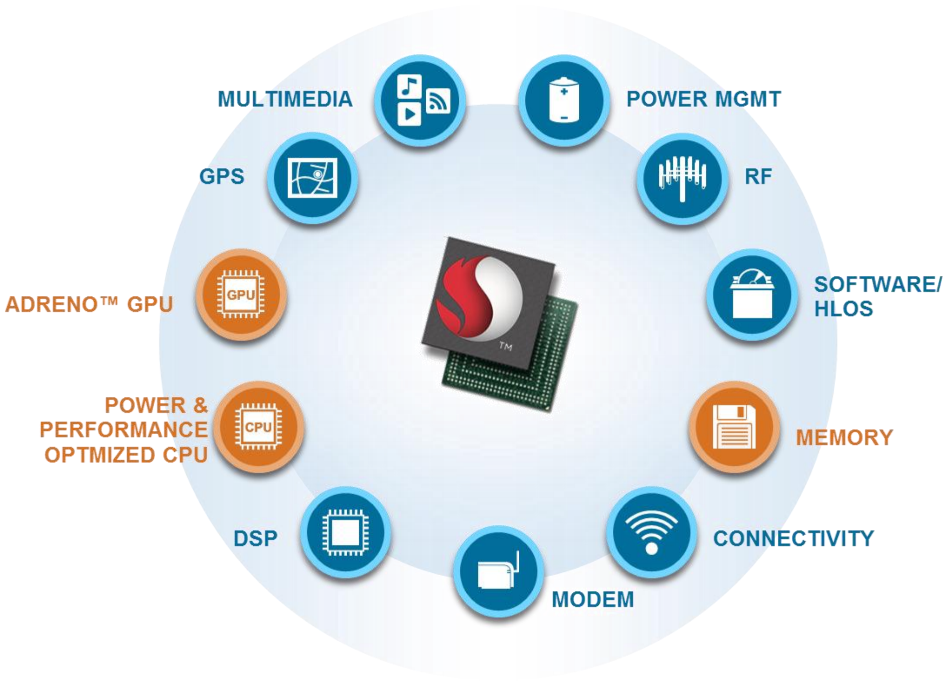


Figure 2-1 Components within Snapdragon system-on-chip processors

Snapdragon is a multiprocessor system that includes components like multimode modem, CPU, GPU, DSP, location/GPS, multimedia, power management, RF, optimizations to software and operating system, memory, connectivity (Wi-Fi, Bluetooth), etc. To learn more about Snapdragon processors, see: http://www.qualcomm.com/snapdragon.

This document focuses on the GPU and touches upon other related components like CPU and memory that are needed for graphics application development. Adreno 3xx is accessible to developers through standard APIs like OpenGL ES, OpenVG, etc. that are supported on Adreno 3xx.

Graphics-intensive applications like those in the following example use a subsystem within the Snapdragon processor for their execution. This subsystem consists of CPU, GPU, memory, and a display controller, which graphics applications use for computing different game engine functions, rendering the geometry and scene, caching the data and displaying the processed image on the screen.

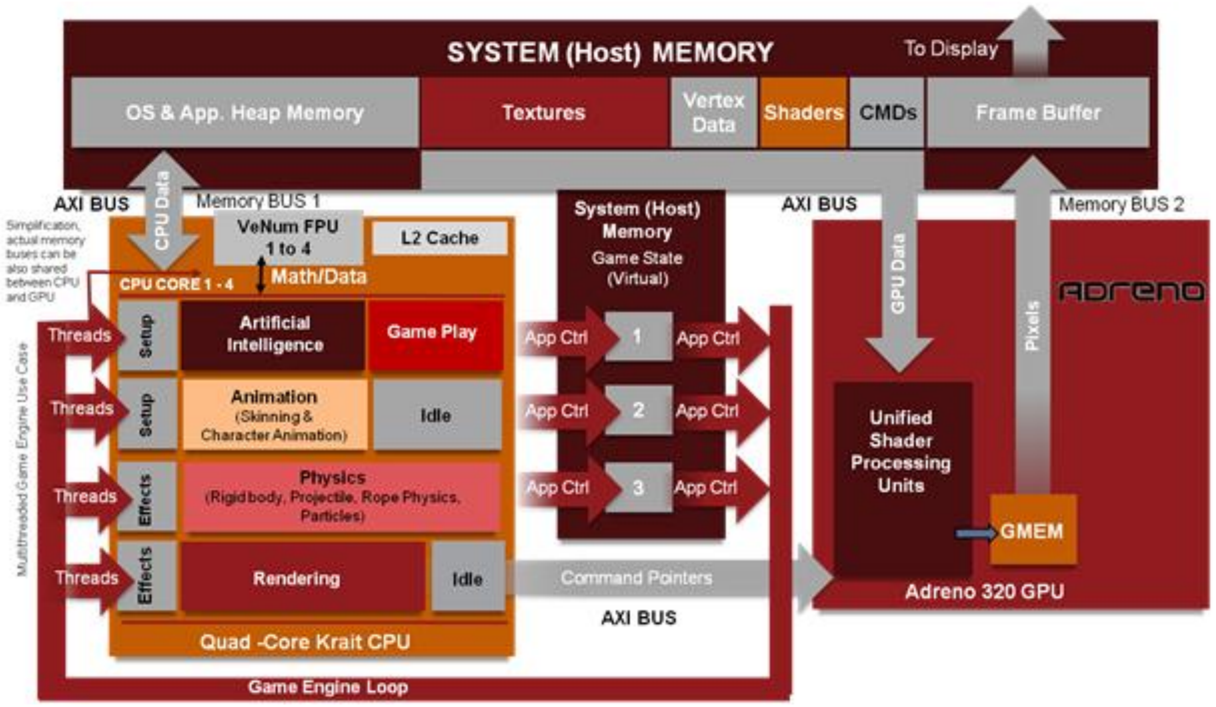


Figure 2-2 Overall system-level flow for graphics applications on Snapdragon S4 Pro (APQ8064)

The following sections address this game dataflow diagram one system component at a time.

CPU

The data flow in Figure 2-2 starts from the left with multiple threads within the game being processed by the quad-core CPU. Assume this hypothetical game has a coarse-threaded game engine architecture. Coarse-threaded architecture means that each of the game engine functions like artificial intelligence, audio and game physics are processed by that CPU core as individual process threads that are assigned by the game engine. A similar flow is possible for a fine-grained threaded game engine, where each of these functions is divided into multiple threads. Each thread in a function (part of the function instead of a complete function) is then processed by a CPU core. The coarse-threaded game engine tends to be more serial in flow on a CPU, whereas a fine-grained threaded engine tends to be parallel in flow.

In Figure 2-2, each game engine function is processed from left to right on the first core in a serial fashion. After processing the last function, the game state is stored in local memory, and the second CPU core taking that state as a starting point begins processing the next series of functions. While these CPU cores are processing the functions, each of the game engine functions can also use the VeNum co-processor for the math calculations. VeNum is a general-purpose SIMD (Single Instruction, Multiple Data) architecture extension that is used for efficient Vector Floating Point (VFP) processing of mathematical algorithms for these game engine functions. VeNum is instructionally compatible with the standard ARM® NEON™ general-purpose SIMD engine. There are multiple ways to use VeNum, including vectorizing compilers, using C instrinsics or using Assembler. For more details, see ARM's reference page: http://www.arm.com/products/processors/technologies/neon.php

System Memory

An advanced graphics application or game needs a significant amount of memory to store textures, vertex data, and other application-related data. All this data, along with the memory needed by the GPU to store frame buffer, is stored in system memory, where the CPU and GPU read these values efficiently as needed. System memory is shared by all applications and the operating system. With gaming and graphics applications becoming increasingly more advanced, most of the system memory needed by these applications is for storing textures. Following simple tricks like compressing these textures leads to significant savings in memory usage.

GPU

Once the CPU completes its computational cycle, the game state data is stored in memory, while processed geometry, texture data pointers and command pointers are passed on to the GPU for further processing. There are multiple important aspects of Adreno GPUs, as shown in Figure 2-2 and discussed later in this document. Here, the focus is on two components: GMEM, which is a tile buffer for storing color, depth and stencil values, and the shader processing unit.

For efficient shader processing, Adreno GPUs use a unified shader architecture. Unified shader architecture is the most effective use of the shader units, as none of the units sit idle when shader instructions are being processed.

Adreno GPUs have an internal local memory buffer that can store z, stencil and color values. The architecture that allows for dividing the final frame buffer into smaller portions, called bins, and resolving these bins one at a time in this local buffer is called binning. The traditional approach of resolving the whole frame buffer at once is known as direct rendering. Adreno 3xx provides a unique advantage in its support of both deferred and direct rendering, as well as the ability to automatically detect which is best to use at any particular moment. This is Qualcomm's unique FlexRender™ solution, which gives Adreno 3xx a substantial advantage over other mobile GPUs.

Display

The final processed pixel data from the frame buffer is updated onto the LCD screen by the display driver.

2.2 Graphics Subsystem

2.2.1 Adreno 3xx Features

Adreno 3xx inherits some of its important architectural benefits from earlier generations of the Adreno GPU and includes valuable new features, as listed below.

Texturing Features

Multiple Textures

Multiple texturing or multitexturing is the use of more than one texture at a time on a polygon. For instance, a light map texture may be used to light a surface as an alternative to recalculating that lighting every time the surface is rendered. Another multitexturing technique is bump mapping, which allows a texture to directly control the facing direction of a surface for the purpose of its lighting calculations. This can help make complex surfaces such as tree bark or rough concrete look much more realistic by adding lighting detail in addition to the usual coloring information. Bump mapping has become popular in recent video games as graphics hardware has become powerful enough to accommodate its use in real time.

Adreno 3xx 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 can reduce overdraw significantly, save ALU cost for fragment shaders, and avoid unnecessary vertex transforms.

Figure 2-3 is a simple example with two textures as static lightmaps on the road.

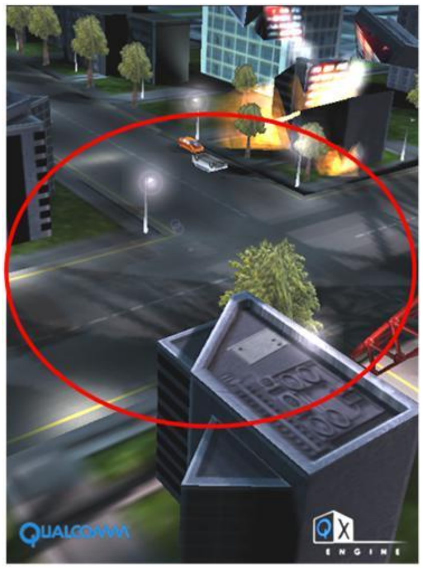


Figure 2-3 Multiple texture pass: static lightmaps example

To use multiple textures in your application, refer to the multitexture sample in the Adreno SDK OpenGL ES Tutorials.

Video Textures

More and more of today's games and graphics applications require a video texture, which consist of moving images that are streamed in real time from a video file. Adreno GPUs support video textures. An example use case for a video texture is a 3D user interface that has an integrated video player/camera view finder, as shown in Figure 2-4. Here, the video stream from the smartphone's camera is applied as a texture to the polygon in this 3D scene that comprises the camera viewfinder.

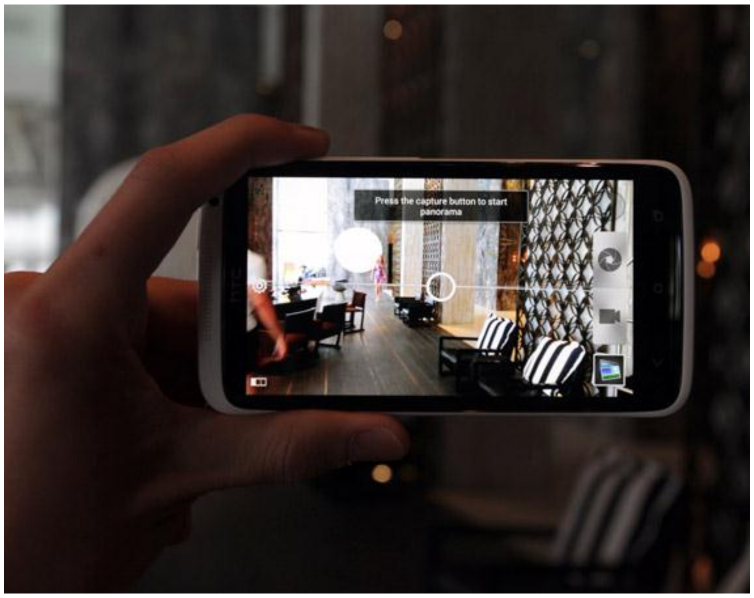


Figure 2-4 Video texture example

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

To use surface textures with camera on Android, you may need to request a special custom format during initialization, as shown here:

CameraParameters.set(“preview-format”,”yuv420sp-adreno”);

Format ”yuv420sp-adreno” is an optimized image format for Adreno GPUs.

Apart from using the standard Android API as suggested, if your application requires video textures, you can also use the standard OpenGL ES extension. Please refer to the extension, “GL_OES_EGL_image” here: http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image.txt.

Texture Compression

Compressing textures can significantly improve the performance and load time for graphics applications since it reduces texture memory and bus bandwidth use. Both offline and dynamic texture compression are possible on these platforms.

Important compression texture formats that are 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 will be supported in OpenGL ES 3.0 API (for R, RG, RGB and RGBA)

To convert textures to these texture compression formats, see Adreno Texture Compression and Visualization Tool (section 3.2.1.1) and Adreno Texture Converter (section 3.2.1.2).

To implement texture compression, refer also to the tutorial sample, “Compressed Texture,” in the Adreno SDK.

Floating Point Textures

Adreno 3xx supports the same texturing features as Adreno 2xx, 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 also includes rendering support for FP16 (full support) and FP32 (no blending).

The supported extensions for this feature are:

- GL_OES_texture_float
- GL_OES_texture_float_linear
- GL_OES_texture_half_float
- GL_OES_texture_half_float_linear

The GL_OES_texture_float extension API uses a 32-bit floating format, whereas GL_OES_texture_half_float extension API uses a 16-bit format. Many minification and magnification filters, along with GL_NEAREST and GL_NEAREST_MIPMAP_NEAREST, are supported by Adreno 3xx.

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). The texture represents six faces of a perfectly mirrored cube, as seen from the vantage points in each of the six cardinal directions, as shown in Figure 2-5. The texture coordinate is a vector that specifies which way to look from the center of the cube to get the desired texel. Adreno 3xx supports seamless edge support for cube mapping.

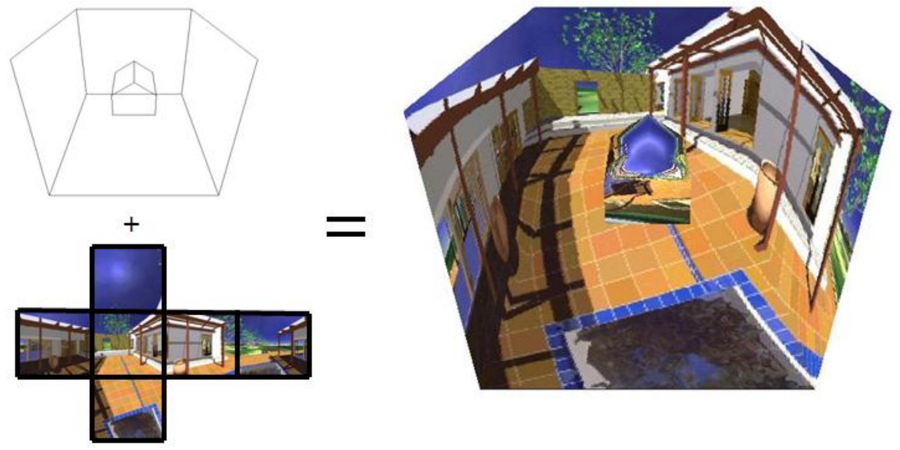


Figure 2-5 Cube mapping

For further details on how to use this feature, see the sample for Cubemap Texture in the Adreno SDK OpenGL ES Tutorials.

3D Textures

In addition to 2D textures and cubemaps, there is a ratified OpenGL ES 2.0 extension for 3D textures called GL_OES_texture_3D. This extension exposes methods for loading and rendering with 3D textures (or volume textures). Think of a three-dimensional texture as a stack of 2D textures. Figure 2-6 is an example of a “64 x 64 x 5” 3D texture. The depth of the 3D texture is 5.

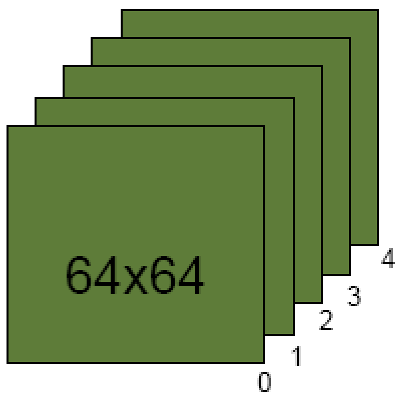


Figure 2-6 3D texture

Use 3D textures to minimize texture cache thrash for materials that have multiple textures.

To use this feature, read the extension “GL_OES_texture_3D” at: http://www.khronos.org/registry/gles/extensions/OES/OES_texture_3D.txt

Large Texture Size

Adreno 3xx supports texture sizes as big as 4096 x 4096 texels.

sRGB Textures and Render Targets

sRGB is a standard RGB color space created cooperatively by HP and Microsoft in 1996 for use on monitors, printers, and the Internet. Today’s smart phone and tablet displays also assume sRGB (nonlinear) color space. Hence, to get the best viewing experience with correct color view, it is important that the render targets and the textures match the same color space as the display, which is sRGB. Unfortunately, OpenGL ES assumes linear or RGB color space by default. But Adreno 3xx supports sRGB color space for render targets as well as textures, making this color-correct viewing experience possible.

To use sRGB format for renderbuffer or texture, follow the steps below:

- Create renderbuffer (*glRenderbufferStorage*) or a texture (*glTexImage2D*) using one of the sRGB formats (see complete list in “gl3.h” header file)
    - SRGB8,
    - SRGB8_ALPHA8,
    - COMPRESSED_SRGB8_ETC2,
    - COMPRESSED_SRGB8_ALPHA8_ETC2_EAC, or
    - COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC,
- Bind rendering buffer or texture to FBO using *glFramebufferRenderbuffer*() for renderbuffer or *glFramebufferTexture2D*() for texture
- Query color encoding: *glGetFramebufferAttachmentParameteriv*(..,..,GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING,..)

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 needs to be loaded, but streaming that texture takes time. So 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 bogging down 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 2-7. As a starting point for the LOD effect, we download Mips for levels 4-10 by setting the base level to 4 and minimum LOD to 4. Once the application starts, we download the Mip 3,2,1,0. Then by setting the base level to 0, we slowly, over multiple frames, change min LOD to 0. Thus, the texture gradually phases to the highest resolution.

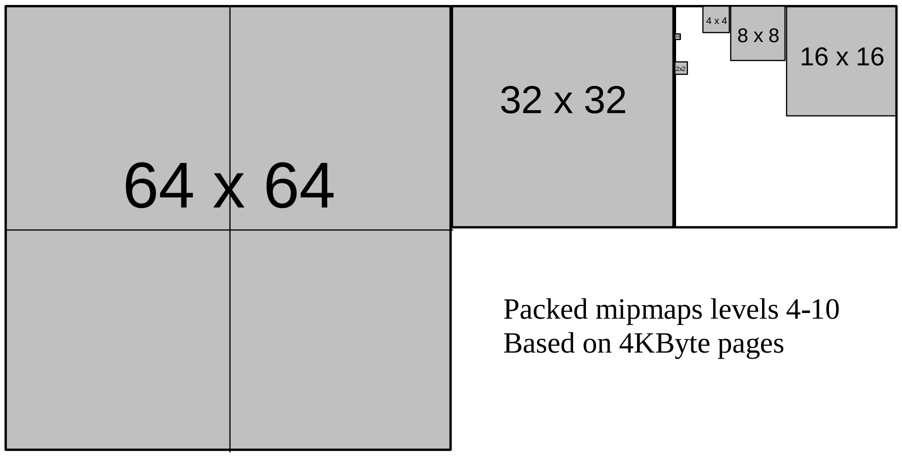


Figure 2-7 Texture LOD

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. 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. 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). 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). The initial value is -1000.

PCF for Depth Textures

Shadow mapping is the method of choice for creating 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 such as perspective shadow maps.

However, 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 2-8 is an example of that implementation.

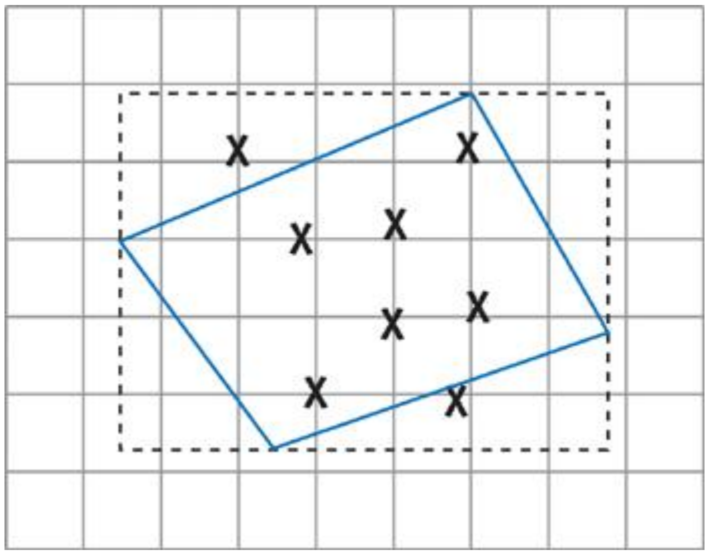


Figure 2-8 An example of percentage-closer filtering algorithm

Adreno 3xx 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 2-9.

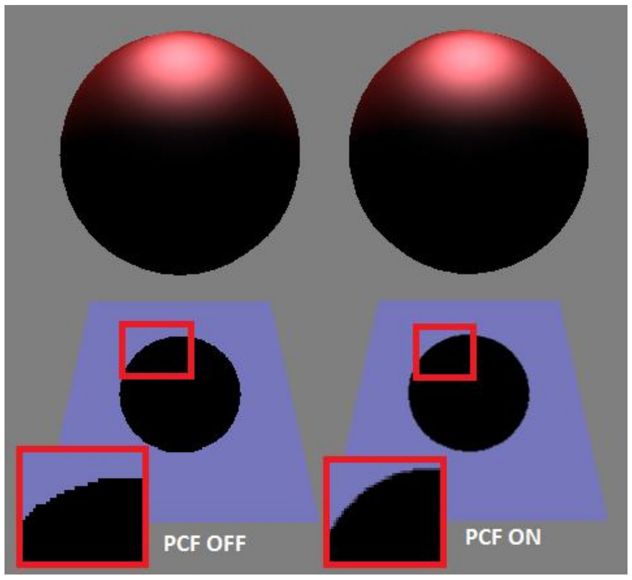


Figure 2-9 Percentage-closer filtering example from the Adreno SDK

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

Visibility Processing

Early Z Rejection

Early Z rejection provides a fast occlusion method as well as the rejection of unwanted render passes for objects that are not visible (hidden) from the view position. Figure 2-10 shows the red circle as an object that is hidden behind the green object, which is represented here by a green block. The rendering pass for this hidden object, which is not visible from the camera viewpoint, is avoided using the early Z rejection feature.

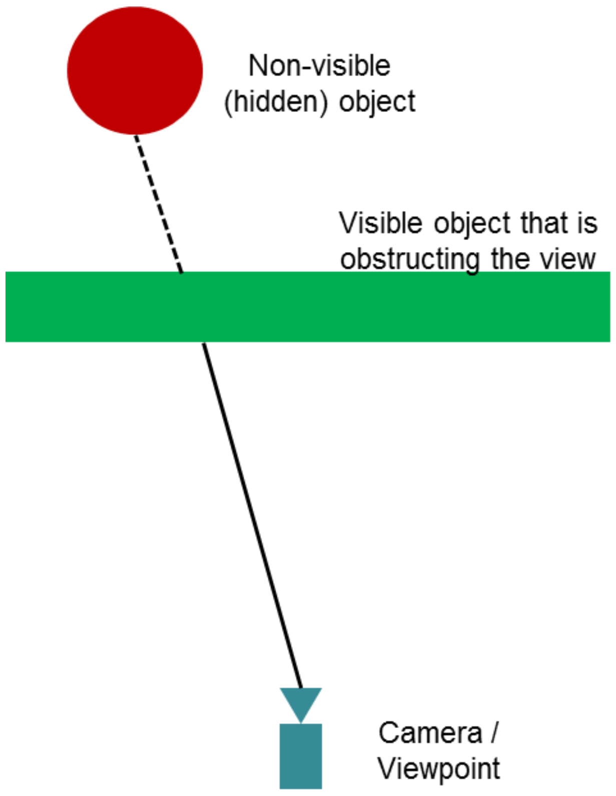


Figure 2-10 Early Z rejection: hidden object

Consider the example in Figure 2-11, which 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 you are 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 can reject occluded pixels at up to four times the drawn pixel fill rate.

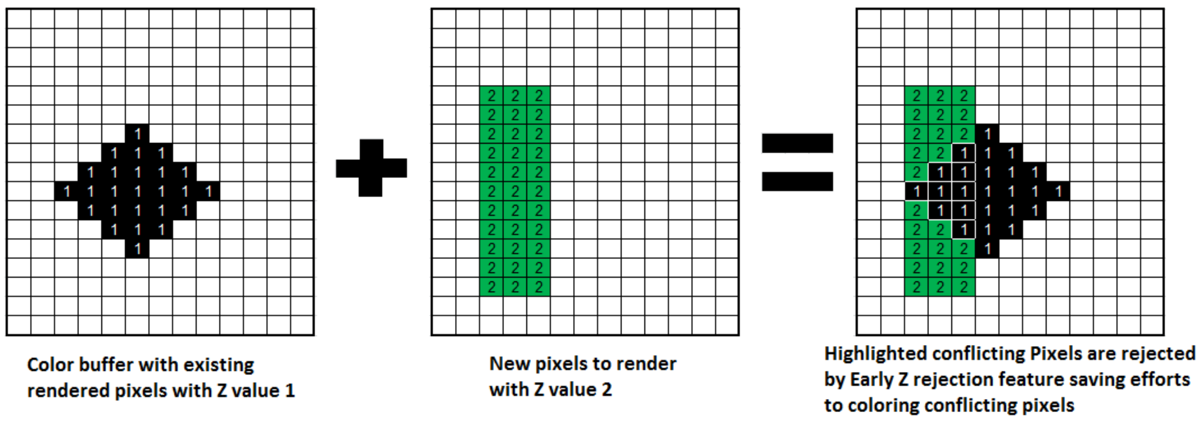

Color buffer with existing rendered pixels with Z value 1
New pixels to render with Z value 2
Highlighted conflicting Pixels are rejected by Early Z rejection feature saving efforts to coloring conflicting pixels

Figure 2-11 Early Z rejection

To get maximum benefit from this feature, we recommend drawing your 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 very useful for applications that have high-depth complexity.

FlexRender (Hybrid Deferred & Direct Rendering Mode)

Qualcomm is introducing its new FlexRender solution as part of Adreno 3xx. FlexRender refers to the GPU's ability to switch between indirect (binning or deferred) rendering and direct rendering to the frame buffer. Both rendering modes (deferred and direct) have their respective advantages, as follows.

Deferred (Binning) Mode Rendering Optimization

The deferred mode rendering mechanism breaks the scene frame buffer into small regions for rendering, thereby optimizing the overall rendering. The deferred mode rendering mechanism of the Adreno GPU uses a two-pass algorithm to render the scene. 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 2-12.

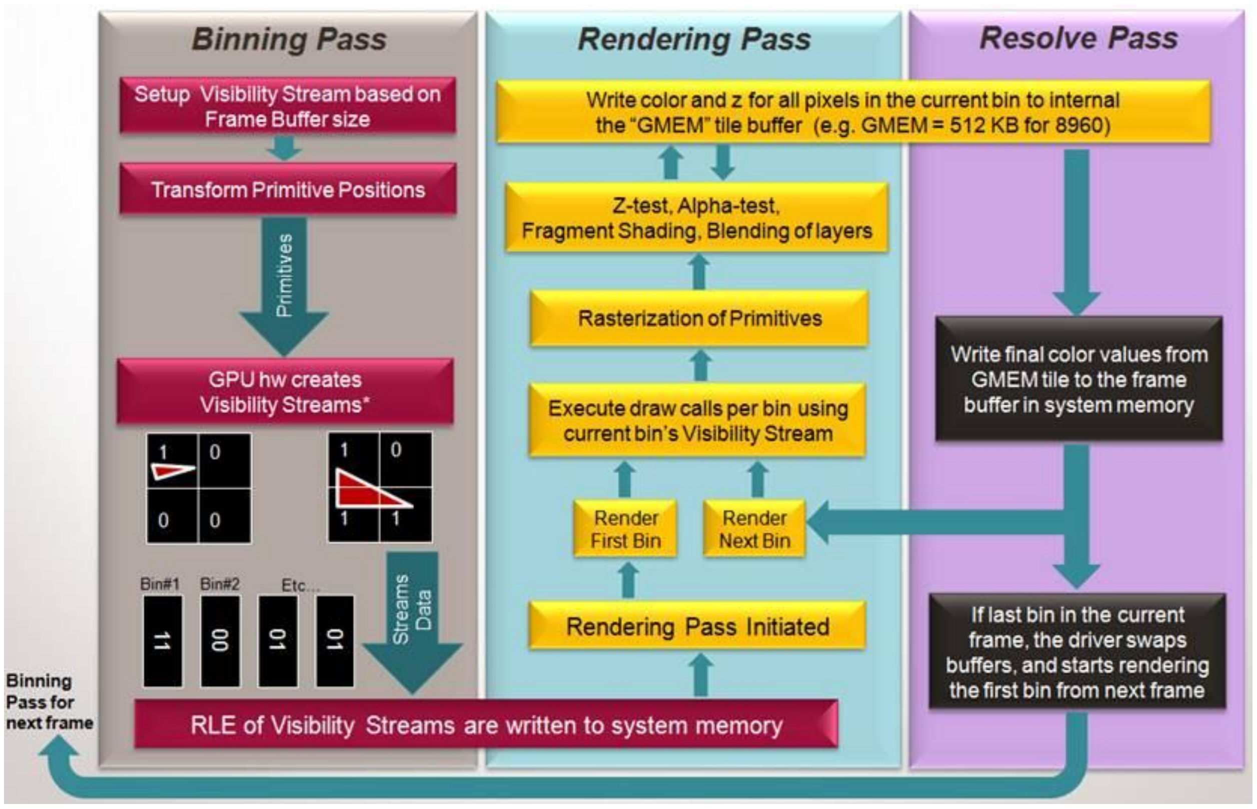

Binning Pass
Rendering Pass
Resolve Pass
Setup Visibility Stream based on Frame Buffer size
Transform Primitive Positions
Primitives
GPU hw creates Visibility Streams*
Bin#1
Bin#2
Etc...
11
00
01
01
Streams Data
Binning Pass for next frame
RLE of Visibility Streams are written to system memory
Write 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
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 2-12 Deferred style of rendering with Adreno 3xx

Direct Rendering

In direct rendering, the intermediate step of processing and rendering the scene per bin is eliminated. So there is no need for local memory for graphics (GMEM). The geometry is directly rendered into the final frame buffer, which is in-system memory. This approach is common in today's desktop GPUs.

Deferred vs. Direct Rendering

Deferred is very useful for cases with considerable depth complexity or overdraw with alpha blending. Deferred significantly reduces memory bandwidth requirements by processing pixels under each bin from internal fast access memory to graphics for these blending operations. Using direct rendering to perform these same blending operations, it is required to read from and write to the system memory (where the final frame buffer resides) for each pixel that is being processed for blending. Reading from and writing to the system memory for processing each pixel is a costly operation that is performed at the expense of battery consumption.

The scenarios where deferred is the better option include:

- The graphics scene in the application has a considerable amount of depth complexity with alpha blending or Z-buffering

- Heavy fragment processing is required for complex shader effect
- Certain bit blit operations like UI composition or transparent blended blits are performed within the application

But binning may not always be the best rendering option. In some situations, rendering a scene using direct rendering may be more efficient as compared to deferred. These situations occur when:

- The scene has a very small overdraw (least depth complexity or single layer of rendering).
- The application makes heavy usage of Render-to-Texture for reflections/refractions type effects.
- The application makes use of cumulative updates or partial scene updates that are timed mid-frame.
- The application has heavy geometry processing with extremely high poly meshes.

FlexRender

Clearly, there are advantages to both the direct and deferred rendering modes. To maximize performance, Adreno 3xx was designed to render with either direct mode or deferred, which it can do dynamically on its own or via a hint from the developer.

The Adreno 3xx GPU can analyze the rendering for a given render target and automatically choose between deferred and direct rendering mode. This is accomplished without the application having to provide any additional information. It is recommend that developers rely on Adreno GPU to determine the best possible mode for rendering.

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; i.e., arithmetic logic units or ALUs, that support both fragment and vertex shaders. Another type of shader architecture that is common in the mobile GPU industry uses dedicated vertex and fragment computational units. Unified shader architecture allows for more flexible use of the ALUs.

Adreno 3xx 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 illustrated in Figure 2-13.

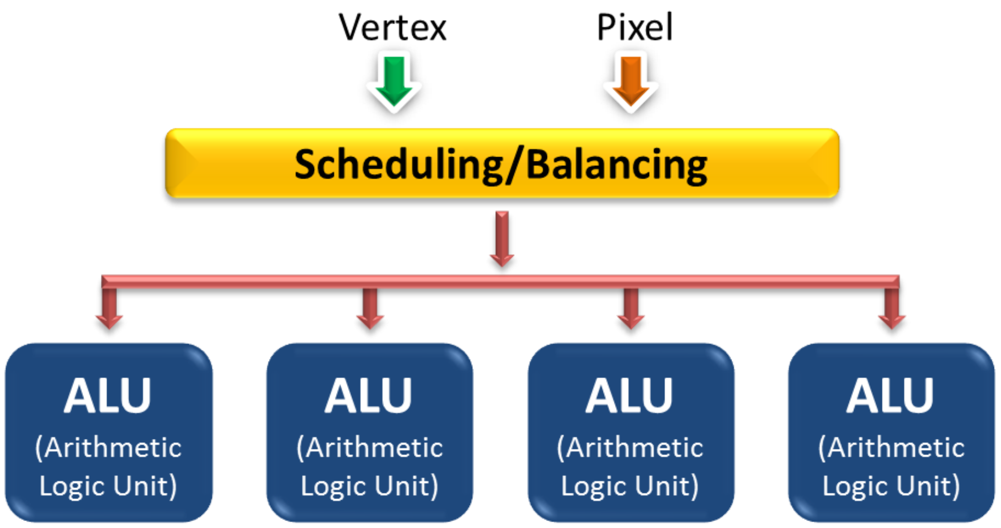

Vertex
Pixel
Scheduling/Balancing
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)

Figure 2-13 Unified Shader Architecture

Figure 2-13 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 illustrated in Figure 2-14. This allows for greater flexibility of pixel/vertex load balances.

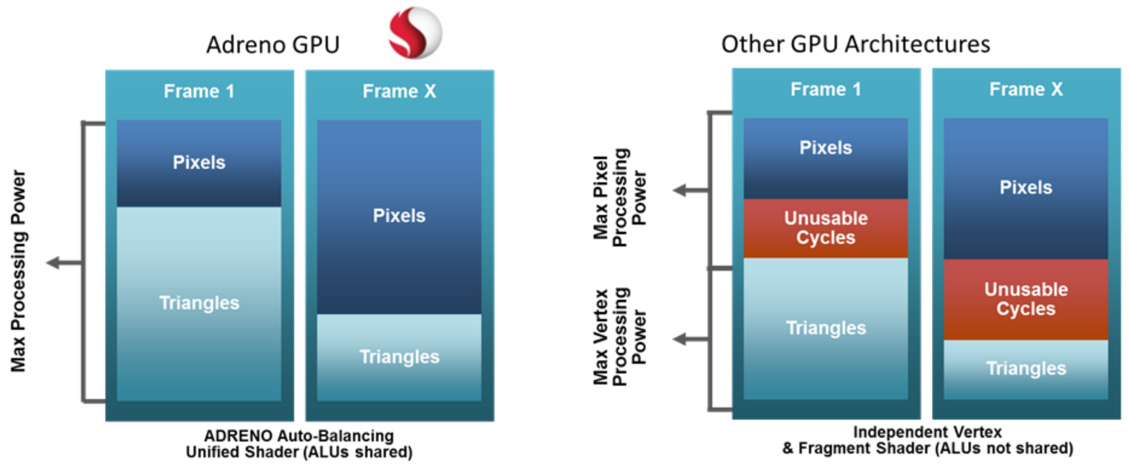


Figure 2-14 Flexibility in shader resources: Unified Shader Architecture

The Adreno 3xx shader architecture is also multithreaded. If a fragment shader's execution is stalled due to a texture fetch, for example, the execution is given to another shader. Multiple shaders are accumulated as long as there is room in the hardware.

In addition to performance advantages, a non-performance-related advantage of unified shaders is that shaders have access to all shader resources, and specifically, vertex texture access, which will be discussed further in this section.

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

Scalar Architecture

Adreno 3xx has a scalar component architecture. The smallest component Adreno 3xx can support natively is a scalar component. This results in more efficient use of the hardware resources for processing scalar components; it does not waste a full vector component to process the scalar.

The scalar architecture of Adreno 3xx 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 3xx, mediump is 16-bit floating point and highp is 32-bit floating point.

To get the maximum benefit from this feature, use the lowest possible precision. Good rules of thumb are:

- For colors and unit length vectors, use fixed or low precision.
- For others, use half if range and precision are fine; otherwise use float.

On mobile platforms, the key is ensuring that the fragment shaders stay in the lowest possible acceptable precision. Applying swizzles to low-precision (fixed/lowp) types is costly; converting between fixed/lowp and higher precision types is also quite costly.

We highly recommend using mediump where that is enough precision for your shader. For detailed guidelines on shaders, see chapter 4 of this document.

Advanced Geometry Support

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, though geometry instancing may also be used for characters. Although vertex data is duplicated across all instanced meshes, each instance may have other differentiating parameters (like color, transforms, lighting, etc.) changed to reduce the appearance of repetition. For example, in Figure 2-15 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.

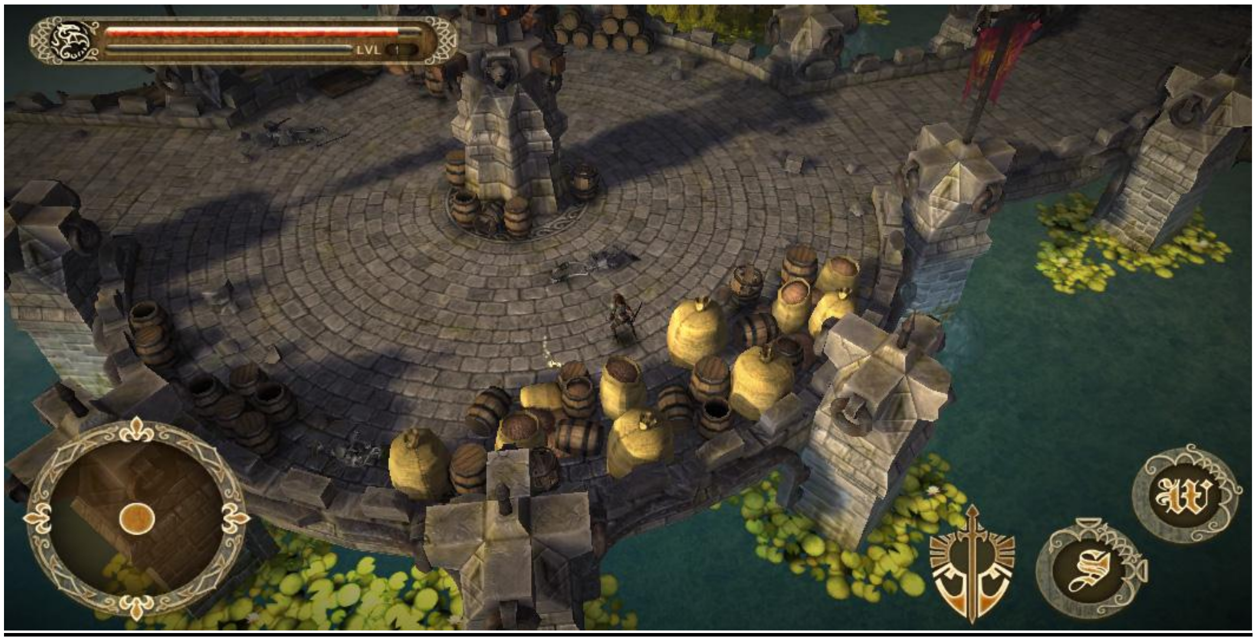


Figure 2-15 Geometry instancing for drawing barrels

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 each instance's transform. Without instancing, the GPU would have to store multiple copies of the vertex and index data.

Please refer to Khronos' OpenGL ES 3.0 API specifications for more details but below are two APIs that enable rendering multiple instances:

```
void glDrawArraysInstanced(GLenum  mode, GLint  first, GLsizei  count, GLsi
zei  primcount);

void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const
void* indices, GLsizei primcount);
```

Pixel / Vertex Format Support

Texture Formats

Texture formats supported by Adreno 3xx include:

- Normalized:
  - L8, A8, LA8, R8, RG8, RGB8, RGBA8, RGB565, RGB5_A1, RGBA4, RGB10_A2, R8_SNORM, RGB8_SNORM, RGBA8_SNORM
- Floating Point:
  - R16F, RG16F, RGB16F, RGBA16F, RGB9_E5, R11F_G11F_B10F, R32F, RG32F, RGB32F, RGBA32F
- Depth/Stencil:

  - DEPTH_COMPONENT24, DEPTH_COMPONENT16, DEPTH24_STENCIL8, DEPTH_COMPONENT32F, DEPTH32F_STENCIL8
- Integer:
  - R8UI, R8I, R16UI, R16I, R32UI, R32I, RG8UI, RG8I, RG16UI, RG16I, RG32UI, RG32I, RGB8UI, RGB8I, RGB16UI, RGB16I, RGB32UI, RGB32I, RGB10_A2UI, RGBA8UI, RGBA8I, RGBA16UI, RGBA16I, RGBA32UI, RGBA32I
- Compressed:
  - ETC2 RGB, ETC2A RGBA, ETC2_punchthroughA, EAC_R11_unsigned, EAC_R11_signed, EAC_RG11_unsigned, EAC_RG11_signed, ETC1_RGB8, ATI_TC_RGB, ATI_TC_RGBA

Vertex Formats

Vertex formats supported by Adreno 3xx include:

- Byte
- Short
- GL_FIXED
- GL_FLOAT
- GL_HALF_FLOAT
- 10.10.10 format stored as 32-bit word

Render Target Formats

Render target formats supported by Adreno 3xx include:

- Normalized:
  - R8, RG8, RGB8, RGBA8, RGB565, RGB5_A1, RGBA4, RGB10_A2
- Floating Point:
  - R16F, RG16F, RGBA16F, R11F_G11F_B10F, R32F, RG32F, RGBA32F
- Depth/Stencil:
  - DEPTH_COMPONENT24, DEPTH_COMPONENT16, DEPTH24_STENCIL8, DEPTH_COMPONENT32F, DEPTH32F_STENCIL8, STENCIL8
- Integer:
  - R8UI, R8I, R16UI, R16I, R32UI, R32I, RG8UI, RG8I, RG16UI, RG16I, RG32UI, RG32I, RGB8UI, RGB10_A2UI, RGBA8UI, RGBA8I, RGBA16UI, RGBA16I, RGBA32UI, RGBA32I

Z and Stencil

The various combinations of Z-buffer and stencil buffer supported by Adreno 3xx include 16, 24 and 32 bits of Z-buffer and 8 bits of stencil buffer.

Stencil buffer allows for masking out an irregularly shaped area using a stencil pattern. The stencil pattern can be created using different rendering commands. In Adreno 3xx, there are two separate stencil states available for front- and back-facing polygons. These are very useful for doing tricks like shadow volumes in a single-render pass.

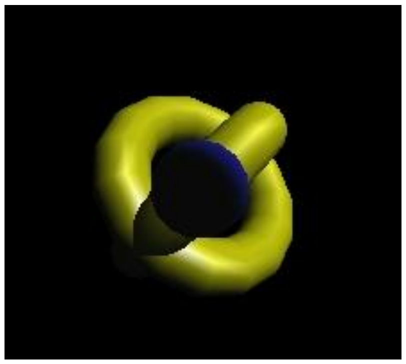


Figure 2-16 Shadow effect using stencil buffer

Other Featured Support

Index Types

A geometry mesh is 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 3xx natively supports 8-bit, 16-bit, and 32-bit index types. Most mobile applications will 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. The geometric primitives that graphics APIs render get rasterized onto a grid. Their edges may become deformed in that process, as shown in the staircase effect images of Figure 2-17. One of the biggest advantages of anti-aliasing is that it smooths out the edges of primitives and can lend a cleaner and more realistic appearance to renderings.

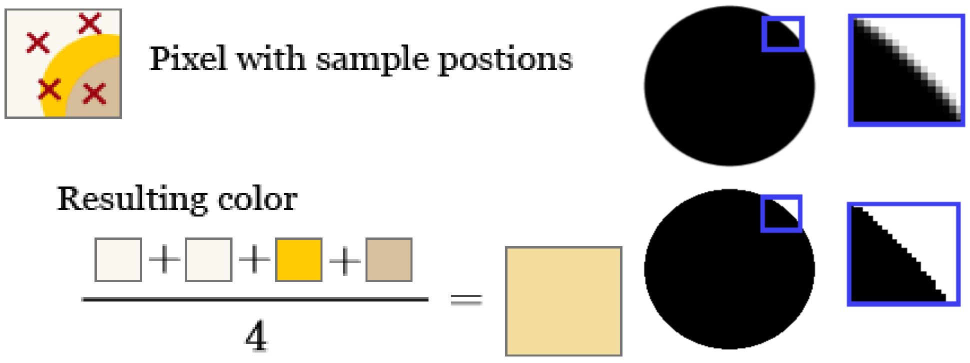


Figure 2-17 MSAA

Among the various techniques for reducing aliasing effects, multisampling is the one supported by Adreno 3xx. 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's time to compose the final image, the samples are resolved into the final pixel color, as depicted in Figure 2-17. Adreno 3xx supports two or four samples.

Read more on implementing MSAA by using the EGL function "eglChooseConfig" to set the frame buffer configuration at: http://www.khronos.org/opengles/documentation/opengles1_0/html/eglChooseConfig.html

Vertex Texture Access or Vertex Texture Fetch

With the advantage of having shared resources to process vertex and fragment shaders because of unified shader architecture as discussed earlier, in the Adreno GPUs the vertex shader has direct access to the texture cache. Consequently, it is simple to implement vertex texture algorithms for function definitions, displacement maps, or lighting LoD systems on Adreno GPUs. Vertex texture displacement is a very advanced technique that is used to render realistic water in games on a desktop and in consoles. The same could now be implemented in your applications running on Adreno GPUs.

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(….);
}
```

2.2.2 Adreno 3xx APIs

Adreno 3xx supports the following Khronos standard Khronos APIs including:

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

Along with the OpenGL ES APIs, the extensions to these APIs are also supported, as presented in chapter 5 .

In addition to the Khronos standard APIs, Adreno 3xx supports Microsoft's DirectX 11 API with Feature Level 9.3. Discussion of these APIs is outside the scope of this document.

3 Tools and Resources

3.1 Hardware Development Kit

The Snapdragon MDP (Mobile Development Platform), in smartphone and tablet form factors, is available to developers who want to get a head start developing and testing their applications on Snapdragon chipsets prior to commercialization. More information on purchasing these devices is available on our website: https://developer.qualcomm.com/develop/development-devices

Figure 3-1 Snapdragon mobile development platform

Snapdragon-powered commercial devices are ubiquitous. To learn more about them, and see a list: http://www.qualcomm.com/snapdragon/devices

3.2 Tools and Resources

This section describes Qualcomm's tools for graphics application development and analysis on Snapdragon platforms. The two primary tools are the Adreno SDK and the Adreno Profiler. They are available for free download on the Qualcomm Developer Network: http://developer.qualcomm.com. Please refer to the Adreno Graphics section of the site for more details.

3.2.1 Adreno SDK

Overview

The Adreno SDK includes support for emulation and other utilities that are important for graphics application development. The SDK is provided as a development environment for Qualcomm's Adreno graphics processors. It is intended for a broad spectrum of developers, from those who want to learn technologies like OpenGL ES, to those who want to utilize the more advanced features of Snapdragon's Adreno graphics solution.

The Adreno SDK can be downloaded by visiting the Adreno section on the Qualcomm Developer Network at https://developer.qualcomm.com/mobile-development/mobile-technologies/gaming-graphics-optimization-adreno/tools-and-resources.

Features

The Adreno SDK includes the following features:

- Emulation support in a desktop environment
- An SDK help system
- SDK browser
- OpenGL ES 2.0 and OpenVG 1.1 tutorials
- OpenGL ES 2.0 samples
- OpenVG 1.1 samples
- A suite of tools to help create content and code
- A Visual Studio project template to create new samples
- Advanced shader samples

Refer to the Adreno SDK documentation for more information.

Notable Components

Notable components of the Adreno SDK are discussed in detail in the following sections.

3.2.1.1 Adreno Texture Compression and Visualization Tool

Overview

The Adreno Texture Tool enables the compression of textures into formats supported by Adreno GPUs and the visual comparison of the compressed texture with a non-compressed one. It also enables setting up projects for batch process that allows compressing multiple textures in a single process. This can be achieved by embedding the command line batch tool in an art pipeline.

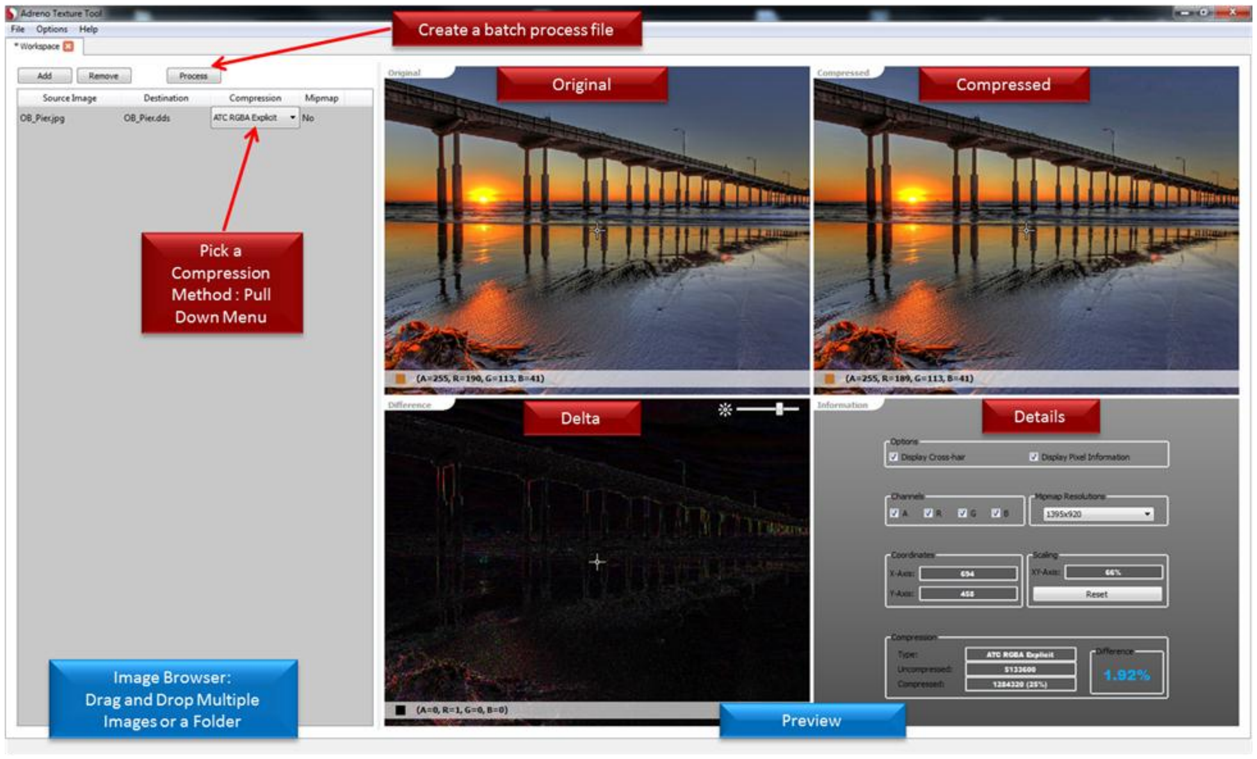


Figure 3-2 Adreno Texture Tool

Features

The Adreno Texture Tool enables the following:

- Visually compare different compression types
- Zoom into textures for a detailed view
- Run a ‘best fit’ compression analysis based on texture size or quality
- Utilize all standard texture compression formats
- 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 texture
- Save files out to open-standard .ktx and .dds formats
- Utilize the included Photoshop plug-in (32- and 64-bit) to load compressed texture formats
- Perform batch processing using the command line tool

3.2.1.2 Adreno Texture Converter

Overview

The Adreno Texture Converter API includes static and dynamic libraries and a corresponding header file to integrate texture compression into a content creation pipeline on the PC side. It can also be used to compress the textures dynamically within the application itself. This API enables maximum use of Adreno 3xx texture caching benefits after compressing the textures.

Features

The Adreno Texture Converter includes the following features:

- A simple interface with a single entry point. For example, Qconvert (<input image>,<output image>)
- Different image manipulations, such as image scaling and color channel swizzle and image compression
- The ability to generate normal maps for bump mapping
- Support for over 40 texture formats, including:
  - All OpenGL ES 1.1 texture formats, including ATC
  - All OpenGL ES 2.0 texture formats, including ETC, ETC2, EAC, and ATC
- A dynamic texture compression library that includes support for different operating systems, including Android, Linux and Windows
- Sample source code and a simple command line utility to convert TGA files into compressed mipmapped textures

For more details, please refer to the Texture Converter documentation in the Adreno SDK.

3.2.1.1 Qstrip (Adreno Geometry Optimizer)

Overview

The QStrip tool helps convert discrete triangles into triangle strips. This facilitates achieving better application performance on Adreno 3xx as resultant triangle strips from QStrip are optimized for the vertex reuse cache**.**

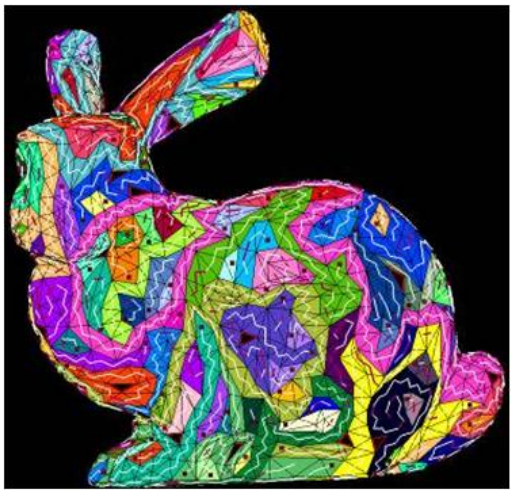


Figure 3-3 Qstrip: Adreno geometry optimizer

Features

QStrip is a simple library for the application-level optimization of vertex index data. The resulting index information may be used in subsequent calls to glDrawElements.

QStrip comes with two header files that expose two data structures.

- Qsplit – Takes sets of triangle indices as input and returns multiple triangle sets. The number of vertices can be predefined. These vertices are indexed by each contiguous set of triangles.
- Qstrip – Takes indexed triangles as input (three indices per triangle) and returns a single index list of triangle strips representing a vertex compression of the previous index data. The strip may be optimized for degenerates and for cache coherency to make the functionality more useful on a broader range of platforms.

For more details, please refer to the QStrip tool documentation within the Adreno SDK.

3.2.1.2 FBXModelConverter

Overview

This tool converts a Filmbox (FBX) file into an optimized model or animation file format that is also used by the Adreno SDK samples. Source code for the tool is provided so users may extend the tool to support additional source file formats and/or add additional commands for mesh processing. The destination mesh file format is designed to minimize the load time of meshes, requiring minimal run-time processing on an embedded device.

For more details, please refer to the FBXModelConverter documentation within the Adreno SDK.

3.2.2 Adreno Profiler

Figure 3-4 Adreno Profiler

Overview

The Adreno Profiler is a leading PC-based tool used by 3D content developers to test, debug, profile and optimize embedded 3D games and applications on commercial Snapdragon-based devices without having to make changes to the application.

The Adreno Profiler provides valuable, time-saving feedback that improves application performance and efficiency. This feedback includes:

- GPU and system-level performance metrics
- OpenGL ES API call tracing and emulation
- Real-time driver overrides
- Shader analysis functionality for estimating source shader complexity

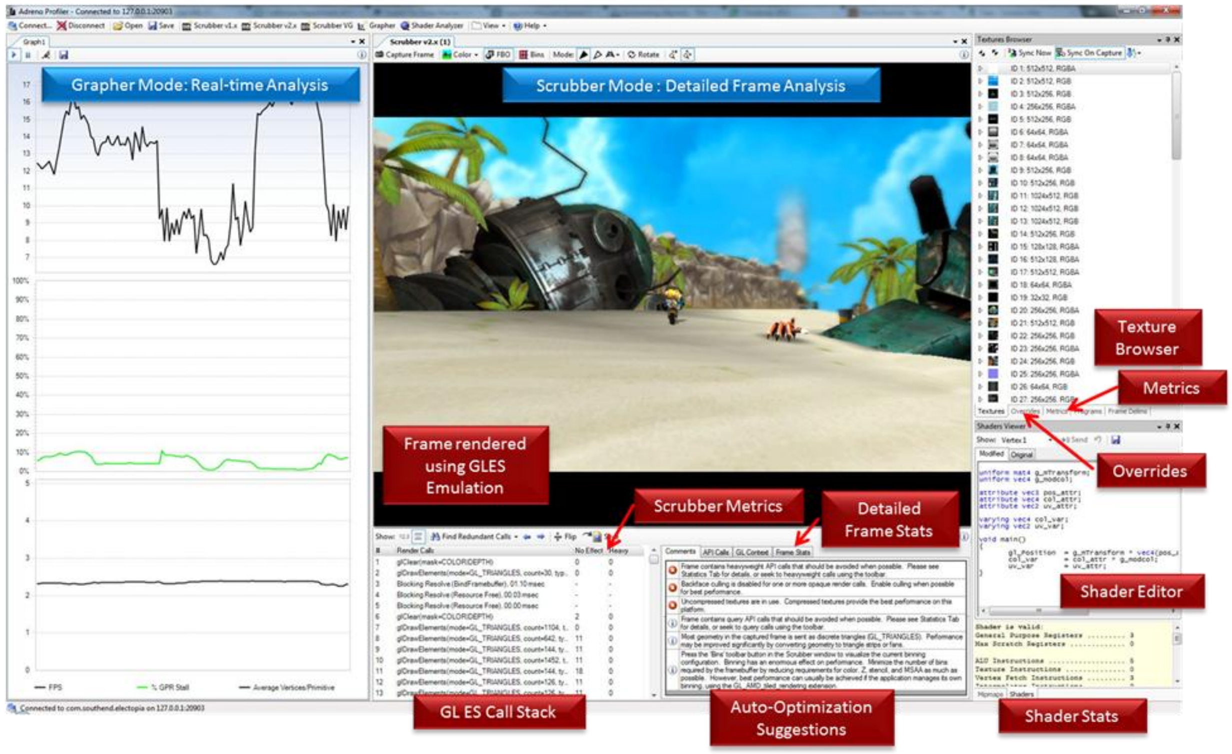

Grapher Mode: Real-time Analysis
Scrubber Mode : Detailed Frame Analysis
Texture Browser
Metrics
Overrides
Frame rendered using GLES Emulation
Scrubber Metrics
Detailed Frame Stats
Shader Editor
GL ES Call Stack
Auto-Optimization Suggestions
Shader Stats

Figure 3-5 Adreno Profiler features overview

The Adreno Profiler can be downloaded free by visiting the Adreno section on the Qualcomm Developer Network at https://developer.qualcomm.com/mobile-development/mobile-technologies/gaming-graphics-optimization-adreno/tools-and-resources.

Features

The Adreno Profiler has two main analysis modes: Scrubber and Grapher, as described in the following sections.

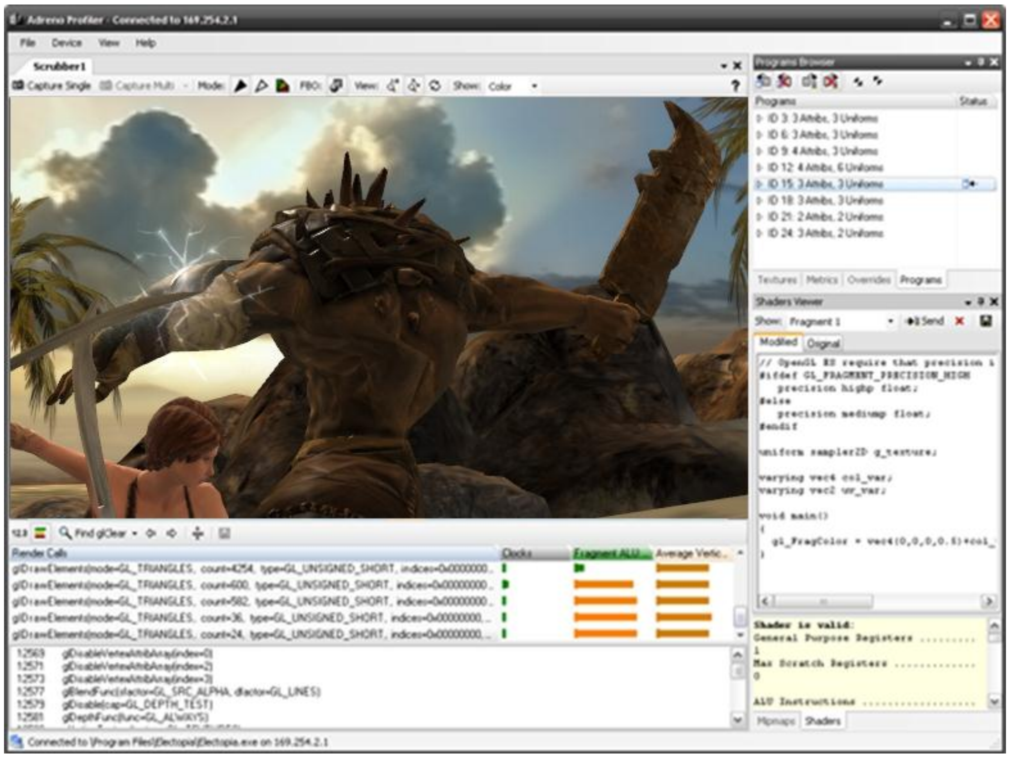
For more details, please refer to the documentation within the Adreno Profiler.

Scrubber Mode

This mode lets you capture frames of animation from the embedded application and inspect the call trace-in detail.

- Scrub the captured call trace through the integrated OpenGL ES emulator
- Search call trace for specific state settings
- Collect and display important performance statistics
- Modify any context state variable for a draw call and see the resulting change in the emulator
- Capture per-render call GPU metrics, such as clock cycles expended, vertex and fragment counts, and many more (applicable to OpenGL ES 2.0-based applications only)

- View, edit and override device shaders for immediate, real-time performance feedback for shader optimizations (applicable to OpenGL ES 2.0-based applications only)
- Download, inspect, and override textures


Adreno Profiler - Connected to 169.254.2.1
File Device View Help
Scrubber1
Capture Single Capture Multi Mode: FBO: View: Show: Color
Programs Browser
Programs Status
ID 3: 3 Attribs, 3 Uniforms
ID 6: 3 Attribs, 3 Uniforms
ID 9: 4 Attribs, 3 Uniforms
ID 12: 4 Attribs, 6 Uniforms
ID 15: 3 Attribs, 3 Uniforms
ID 18: 3 Attribs, 3 Uniforms
ID 21: 2 Attribs, 2 Uniforms
ID 24: 3 Attribs, 2 Uniforms
Textures Metrics Overrides Programs
Shaders Viewer
Show: Fragment 1 Send
Modified Original
Find glClear
Render Calls Clocks Fragment ALU Average Vertic...
Mipmaps Shaders

```
// OpenGL ES require that precision i
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif

uniform sampler2D g_texture;

varying vec4 col_var;
varying vec2 uv_var;

void main()
{
  gl_FragColor = vec4(0,0,0,0.5)*col_
}
```

```
Shader is valid:
General Purpose Registers ..........
1
Max Scratch Registers ..............
0

ALU Instructions ...................
```

```
glDrawElements(mode=GL_TRIANGLES, count=4254, type=GL_UNSIGNED_SHORT, indices=0x0000000...
glDrawElements(mode=GL_TRIANGLES, count=600, type=GL_UNSIGNED_SHORT, indices=0x00000000...
glDrawElements(mode=GL_TRIANGLES, count=582, type=GL_UNSIGNED_SHORT, indices=0x00000000...
glDrawElements(mode=GL_TRIANGLES, count=36, type=GL_UNSIGNED_SHORT, indices=0x00000000,...
glDrawElements(mode=GL_TRIANGLES, count=24, type=GL_UNSIGNED_SHORT, indices=0x00000000,...
```

```
12569   glDisableVertexAttribArray(index=0)
12571   glDisableVertexAttribArray(index=2)
12573   glDisableVertexAttribArray(index=3)
12577   glBlendFunc(sfactor=GL_SRC_ALPHA, dfactor=GL_LINES)
12579   glDisable(cap=GL_DEPTH_TEST)
12581   glDepthFunc(func=GL_ALWAYS)
```

Connected to \Program Files\Electopia\Electopia.exe on 169.254.2.1

Figure 3-6 Adreno Profiler: Scrubber mode

Grapher Mode

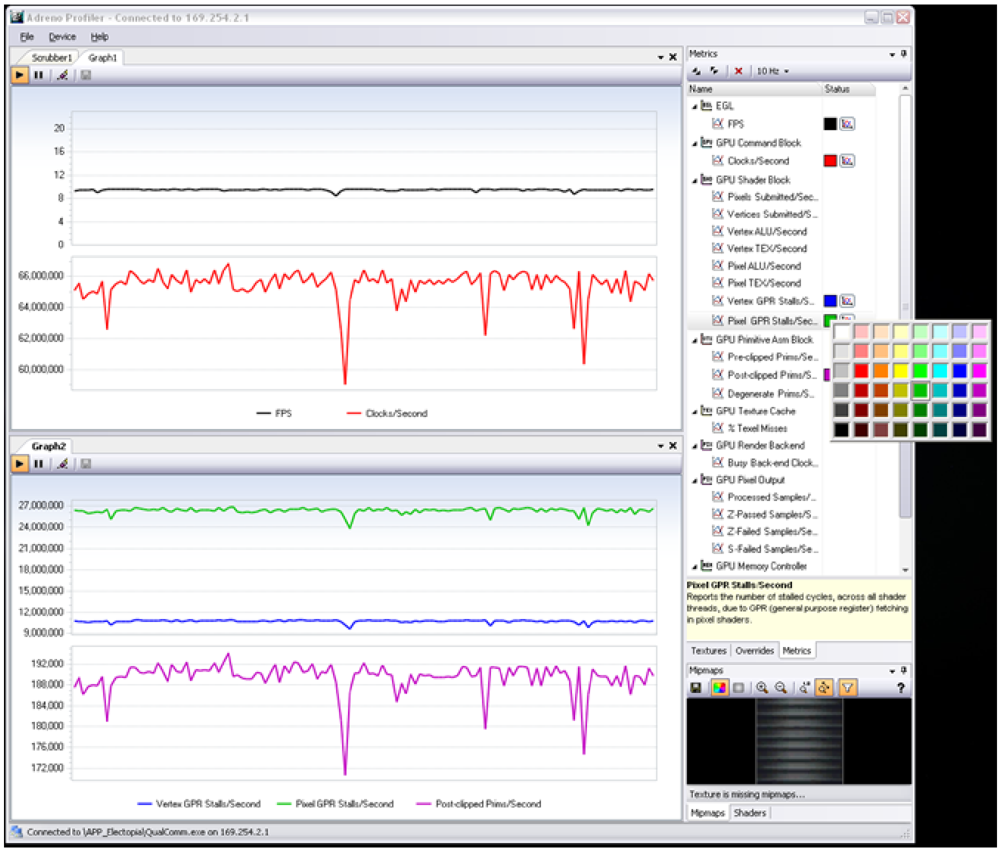
This mode plots real-time performance data streaming from the embedded graphics driver including:

- GPU:
  - Frames per second
  - Clock rate
  - Primitive and fragment throughput at each pipeline stage
  - Detailed instruction counts; e.g., ALU, shader branching, texture fetching, etc.
  - Stalls; e.g., texel fetch, General Purpose Register (GPR), etc.
  - Texture cache misses
  - Overdraw
  - Average triangle area

- System:
  - A feedback notification once there is an expensive driver operation, such as resolves (when blocking pipeline flushes)
  - Global and per-context GPU activity
  - Capture metrics that represent either GPU activity for the current context or Global GPU activity across all contexts

Note that while capturing performance metrics, you can override the operation of the embedded graphics driver in many ways, including setting fixed function state; e.g., switching blending ON/OFF and overriding the texturing parameters, as well as replacing shaders, etc.

Overrides are an important feature that can save significant development time. In the case of texture compression, the overrides feature can be used to simulate smaller textures, which essentially replicates the case when you are using compressed textures in your application. Compressing the textures and recompiling the art and application can take hours or days, depending on your application size, but the Adreno Profiler with its override feature can help test texture compression with the click of a button.


Adreno Profiler - Connected to 169.254.2.1

Pixel GPR Stalls/Second
Reports the number of stalled cycles, across all shader threads, due to GPR (general purpose register) fetching in pixel shaders.

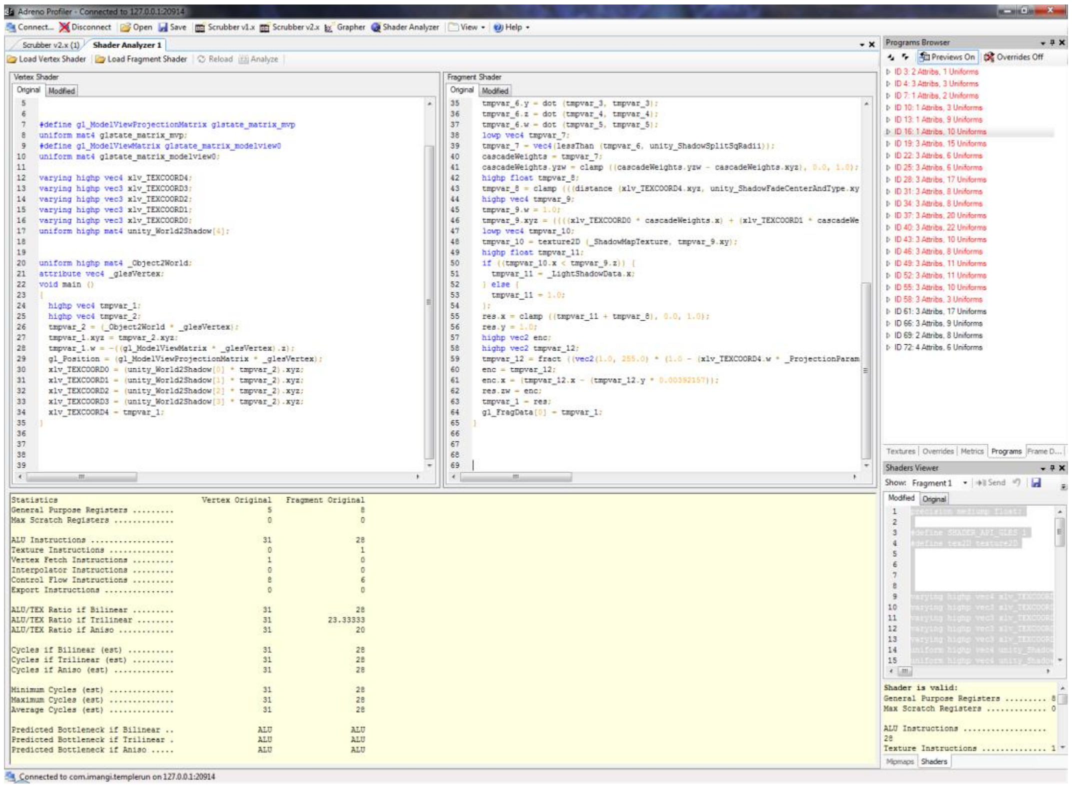
Texture is missing mipmaps...

Connected to \APP_Electopia\QualComm.exe on 169.254.2.1

Figure 3-7 Adreno Profiler: Grapher mode

Shader Analyzer Mode

This mode is particularly useful for analyzing vertex and fragment shaders offline. Any shader program code that needs to be analyzed can be copied and pasted into the respective window of the shader analyzer. The shader analyzer is able to provide an accurate number of GPRs (General Purpose Registers) and ALUs for those particular shaders.


Adreno Profiler - Connected to 127.0.0.1:20914
Connect... Disconnect Open Save Scrubber v1.x Scrubber v2.x Grapher Shader Analyzer View Help
Scrubber v2.x (1) Shader Analyzer 1
Load Vertex Shader Load Fragment Shader Reload Analyze
Vertex Shader
Original Modified
Fragment Shader
Original Modified
Programs Browser
Previews On Overrides Off
Textures Overrides Metrics Programs Frame D...
Shaders Viewer
Show: Fragment 1 Send
Modified Original
Mipmaps Shaders

```


#define gl_ModelViewProjectionMatrix glstate_matrix_mvp
uniform mat4 glstate_matrix_mvp;
#define gl_ModelViewMatrix glstate_matrix_modelview0
uniform mat4 glstate_matrix_modelview0;

varying highp vec4 xlv_TEXCOORD4;
varying highp vec3 xlv_TEXCOORD3;
varying highp vec3 xlv_TEXCOORD2;
varying highp vec3 xlv_TEXCOORD1;
varying highp vec3 xlv_TEXCOORD0;
uniform highp mat4 unity_World2Shadow[4];


uniform highp mat4 _Object2World;
attribute vec4 _glesVertex;
void main ()
{
  highp vec4 tmpvar_1;
  highp vec4 tmpvar_2;
  tmpvar_2 = (_Object2World * _glesVertex);
  tmpvar_1.xyz = tmpvar_2.xyz;
  tmpvar_1.w = -((gl_ModelViewMatrix * _glesVertex).z);
  gl_Position = (gl_ModelViewProjectionMatrix * _glesVertex);
  xlv_TEXCOORD0 = (unity_World2Shadow[0] * tmpvar_2).xyz;
  xlv_TEXCOORD1 = (unity_World2Shadow[1] * tmpvar_2).xyz;
  xlv_TEXCOORD2 = (unity_World2Shadow[2] * tmpvar_2).xyz;
  xlv_TEXCOORD3 = (unity_World2Shadow[3] * tmpvar_2).xyz;
  xlv_TEXCOORD4 = tmpvar_1;
}
```

```
  tmpvar_6.y = dot (tmpvar_3, tmpvar_3);
  tmpvar_6.z = dot (tmpvar_4, tmpvar_4);
  tmpvar_6.w = dot (tmpvar_5, tmpvar_5);
  lowp vec4 tmpvar_7;
  tmpvar_7 = vec4(lessThan (tmpvar_6, unity_ShadowSplitSqRadii));
  cascadeWeights = tmpvar_7;
  cascadeWeights.yzw = clamp ((cascadeWeights.yzw - cascadeWeights.xyz), 0.0, 1.0);
  highp float tmpvar_8;
  tmpvar_8 = clamp (((distance (xlv_TEXCOORD4.xyz, unity_ShadowFadeCenterAndType.xy
  highp vec4 tmpvar_9;
  tmpvar_9.w = 1.0;
  tmpvar_9.xyz = ((((xlv_TEXCOORD0 * cascadeWeights.x) + (xlv_TEXCOORD1 * cascadeWe
  lowp vec4 tmpvar_10;
  tmpvar_10 = texture2D (_ShadowMapTexture, tmpvar_9.xy);
  highp float tmpvar_11;
  if ((tmpvar_10.x < tmpvar_9.z)) {
    tmpvar_11 = _LightShadowData.x;
  } else {
    tmpvar_11 = 1.0;
  };
  res.x = clamp ((tmpvar_11 + tmpvar_8), 0.0, 1.0);
  res.y = 1.0;
  highp vec2 enc;
  highp vec2 tmpvar_12;
  tmpvar_12 = fract ((vec2(1.0, 255.0) * (1.0 - (xlv_TEXCOORD4.w * _ProjectionParam
  enc = tmpvar_12;
  enc.x = (tmpvar_12.x - (tmpvar_12.y * 0.00392157));
  res.zw = enc;
  tmpvar_1 = res;
  gl_FragData[0] = tmpvar_1;
}
```

```
Statistics                             Vertex Original   Fragment Original
General Purpose Registers .........                  5                   8
Max Scratch Registers .............                  0                   0

ALU Instructions ..................                 31                  28
Texture Instructions ..............                  0                   1
Vertex Fetch Instructions .........                  1                   0
Interpolator Instructions .........                  0                   0
Control Flow Instructions .........                  8                   6
Export Instructions ...............                  0                   0

ALU/TEX Ratio if Bilinear .........                 31                  28
ALU/TEX Ratio if Trilinear ........                 31            23.33333
ALU/TEX Ratio if Aniso ............                 31                  20

Cycles if Bilinear (est) ..........                 31                  28
Cycles if Trilinear (est) .........                 31                  28
Cycles if Aniso (est) .............                 31                  28

Minimum Cycles (est) ..............                 31                  28
Maximum Cycles (est) ..............                 31                  28
Average Cycles (est) ..............                 31                  28

Predicted Bottleneck if Bilinear ..                ALU                 ALU
Predicted Bottleneck if Trilinear .                ALU                 ALU
Predicted Bottleneck if Aniso .....                ALU                 ALU
```

ID 3: 2 Attribs, 1 Uniforms
ID 4: 3 Attribs, 3 Uniforms
ID 7: 1 Attribs, 2 Uniforms
ID 10: 1 Attribs, 3 Uniforms
ID 13: 1 Attribs, 9 Uniforms
ID 16: 1 Attribs, 10 Uniforms
ID 19: 3 Attribs, 15 Uniforms
ID 22: 3 Attribs, 6 Uniforms
ID 25: 3 Attribs, 6 Uniforms
ID 28: 3 Attribs, 17 Uniforms
ID 31: 3 Attribs, 8 Uniforms
ID 34: 3 Attribs, 8 Uniforms
ID 37: 3 Attribs, 20 Uniforms
ID 40: 3 Attribs, 22 Uniforms
ID 43: 3 Attribs, 10 Uniforms
ID 46: 3 Attribs, 8 Uniforms
ID 49: 3 Attribs, 11 Uniforms
ID 52: 3 Attribs, 11 Uniforms
ID 55: 3 Attribs, 10 Uniforms
ID 58: 3 Attribs, 3 Uniforms
ID 61: 3 Attribs, 17 Uniforms
ID 66: 3 Attribs, 9 Uniforms
ID 69: 2 Attribs, 8 Uniforms
ID 72: 4 Attribs, 6 Uniforms

```
precision mediump float;

#define SHADER_API_GLES 1
#define tex2D texture2D




varying highp vec4 xlv_TEXCOORD
varying highp vec3 xlv_TEXCOORD
varying highp vec3 xlv_TEXCOORD
varying highp vec3 xlv_TEXCOORD
varying highp vec3 xlv_TEXCOORD
uniform highp vec4 unity_Shado
uniform highp vec4 unity_Shado
```

```
Shader is valid:
General Purpose Registers ......... 8
Max Scratch Registers ............. 0

ALU Instructions ..................
28
Texture Instructions .............. 1
```

Connected to com.imangi.templerun on 127.0.0.1:20914

Figure 3-8 Adreno Profiler: Shader Analyzer mode

Further Details

For more details, please refer to the documentation within the Adreno Profiler.

3.2.3 Other SDKs from Qualcomm

Snapdragon SDK for Android

The Snapdragon SDK for Android is a collection of software components that can be downloaded to enhance applications especially when they're running on Snapdragon powered devices. This SDK is designed to make it easy for developers and device manufacturers to take advantage of a host of next-gen technologies, from low-power, always-on geo-fencing, to complex facial recognition and computer vision, to dual mic audio recording.

With the Snapdragon SDK for Android, existing applications can be enhanced for differentiated and optimized user experiences while maintaining a single code base.

For more information on the Snapdragon SDK for Android, please see the Snapdragon SDK for Android section of the Qualcomm Developer Network at: https://developer.qualcomm.com/mobile-development/mobile-technologies/snapdragon-sdk-android

Augmented Reality (Vuforia) SDK

Vuforia brings a new dimension to mobile experiences by utilizing augmented reality. Vuforia provides industry-leading technology and performance on a wide range of mobile devices. Vuforia's computer vision functionality recognizes a variety of 2D and 3D visual targets. With support for iOS, Android, and Unity 3D, Vuforia enables you to write a single native app that can reach over 400 models of smartphones and tablets.

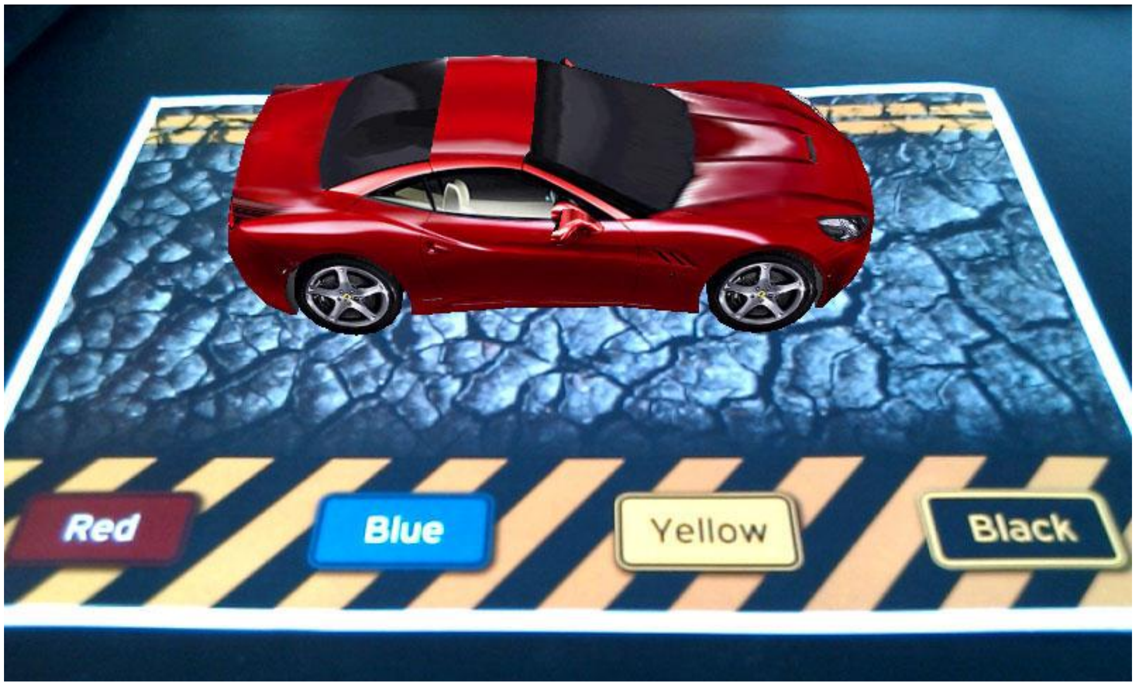


Figure 3-9 Vuforia SDK: augmented reality example

To learn more about the Vuforia SDK, see the Augmented Reality section of the Qualcomm Developer Network at: https://developer.qualcomm.com/develop/mobile-technologies/augmented-reality

Peer-to-Peer (AllJoyn™) SDK

Designed to solve many of the problems that exist in enabling peer-to-peer (P2P) communication, AllJoyn is an open-source software framework developed by Qualcomm Innovation Center Inc. (QuIC), that makes it easy to add ad-hoc, proximity-based P2P to applications without having to connect to cellular networks.

To learn more about the AllJoyn SDK, see the AllJoyn section of the Qualcomm Developer Network at: https://developer.qualcomm.com/mobile-development/mobile-technologies/peer-peer-alljoyn

4 Writing Shaders

4.1 Considerations Before Writing Shader Code

Lightweight / Heavyweight Resolves

A resolve is a memory copy operation. Copying data from one memory area to another is one of the most expensive operations. Resolves can be split into two types. The lightweight resolve copies a render surface from internal graphics memory to external memory. A heavyweight resolve copies data first from internal memory to external memory, and then copies it back.

While the typical example for a lightweight resolve is to display the backbuffer, the typical example for a heavyweight resolve is the use of glReadPixels. To read back the pixels, the results that were first saved out to the slow external memory need to be copied back into fast internal memory before the bin can finish rendering. Because of this double-copy, heavyweight resolves cost twice as much as lightweight resolves.

OpenGL ES 2.0 functions that can trigger heavyweight resolves under certain circumstances are listed as below.

eglSwapBuffers – This call triggers the driver to flush all of its buffered rendering to the hardware. By default, the swap command copies the content of the previous frame's depth buffer into each bin before starting the new frame. Starting the frame by clearing the depth buffer prevents the hardware from having to copy the depth 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 color buffer.

glTexImage2D, glTexSubImage2D, glBufferData, and ***glBufferSubData*** – These calls update memory buffers and can cause a heavyweight resolve. 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 and ***glCopyTexSubImage2D*** – These calls involve heavyweight resolves. Instead of these calls, use Frame Buffer Objects (FBO) to simulate the functionality.

glReadPixels – This call forces a complete flush of the GPU and idles the graphics pipeline. It is very expensive and thus best 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.

glBindFrameBuffer – If the bound buffers are not cleared, the driver assumes that the previous data in the 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.

A post-processing pipeline is a typical usage scenario in which the programmer needs to make sure only lightweight resolves are involved. For example, 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 into it.
2. Bind another smaller frame buffer object, clear it, and render a down-sampled and blurred version of the scene for future use as the out-of-focus scene.
3. Bind the back buffer, then combine the previous two buffer objects 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.

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);
…
```

4.2 Writing Efficient Shader Code

Tools of the Trade

The tools that can help in writing an efficient code are Qstrip, Adreno Texture Tool, Adreno Texture Converter and Shader Analyzer. While the first three are part of the Adreno SDK, which is explained in section 3.2.1 , Shader Analyzer is part of the Adreno Profiler. See section 3.2.2 for more information.

Shader Programming Tips

Achieving high performance is about removing bottlenecks and balancing the components of the system. Due to the mobile nature of Adreno GPUs, the tile-based architecture – as described above - prefers a lower-geometry density and scales better on the per-pixel operations. A simplified memory bandwidth analysis can demonstrate how high the geometry density can be.

Vertex and Pixel Shader Usage

The Adreno GPU supports:

- Fetching 256-bit words or 32 bytes from up to two different vertex streams at once
- Vectorized ALUs
- Bilinear filtered texture sampling – Four bilinear texture samples per clock
- Less expensive MSAA – 2x and 4x hardware multi-sample anti-aliasing
- Up to four render targets simultaneously with a depth/stencil buffer
- Up to eight threads in-flight at once (also called simultaneous contexts). Changing shaders or render states is inexpensive, since a new context can be started up easily. Threads work on units of four vertices or fragments at once.
- Pixel processing in 2x2 quads as shown in Figure 4-1.

Fetching Vertex Data

Indexed triangle strips are favored over indexed triangle lists on Adreno GPUs. The Qstrip tool helps reorder vertices so that they are optimally aligned in cache to reduce the cache misses.

Instancing

The major performance bottleneck in many graphical applications is the number of submitted geometry batches. While vertex throughput has substantially increased over the years, the number of batches or draw-primitives calls that can be rendered each frame have remained roughly the same. The number of batches that can be rendered per frame is directly tied to CPU performance, and any GPU performance improvements have little impact on this bottleneck.

To deal with this challenge, instanced geometry can be drawn in a single draw call. Instead of rendering many similar objects one at a time, the application can now specify the common instanced object data in one vertex stream and per-instance parameters (e.g., position, color, size, etc.) in another vertex stream; those streams are then combined in the vertex shader (GL_EXT_draw_instanced). The hardware will automatically replicate the object vertex data for each of the rendered objects while pairing it with per-instance data.

Use Intrinsics

Intrinsic functions as part of the GLSL language should be used whenever feasible, because it is unnecessary to write an entirely new function. There is a good chance that they are optimized for specific shader profiles. Consult the GLSL language guide for all the functions supported.

Use the Appropriate Data Type

By using the appropriate data type in the code, the compiler and the optimizer in the driver can optimize code and pair shader instructions. For example, using a float4 data type instead of a float data type prevents the compiler from arranging the output in a way that they can be co-issued on hardware. Small mistakes can sometime have a large impact on performance. For example, the following code should consume a single instruction slot.

```
int4 ResultOfA(int4 a)
{
  return a + 1;
}
```

```
The following code might consume 8 instruction slots because of 1.0.
```

```
int4 ResultOfA (int4 a)
{
  return a + 1.0;
}
```

The variable *a* will be converted to vec4. Then the addition will be done in floating point. Finally, the result will be converted back to the return type int4.

Reduce Type Casting

Reducing the number of type casts follows logic that is similar to the recommendation of using the appropriate data types. The following code might be sub-optimal:

```
vec3 Color = texture2D(Color, TexC);
Color *= diff + amb;
return Color;
```

Using the following GLSL shader can reduce the number of instructions by one:

```
vec4 Color = texture2D(Color, TexC);
Color *= diff + amb;
return Color;
```

Pack Scalar Constants

Packing scalar constants into vectors consisting of four channels substantially improves the fetch effectiveness of the hardware. For example, in the case of an animation system, this increases the number of available bones for skinning. The following code shows a simple way to achieve this:

```
float scale, bias;
vec4 a = Pos * scale + bias;
```

The following code might take one instruction less because the compiler can optimize the line to a more efficient instruction (mad).

```
Vec2 scaleNbias;
vec4 a = Pos * scaleNbias.x+ scaleNbias.y;
```

Keep Shader Length Reasonable

Extremely long shaders can be very inefficient. If there is a need to include a large number of instruction slots in a shader when compared to the texture fetches, the parts of the rendering algorithm can be stored into the texture and then fetched as a texture. A technique like this could also be expensive for memory bandwidth. But simple texture tricks like trilinear, anisotropic filtering, wide texture formats, volume and cubemap textures, texture projection, texture look-up with gradients or different LOD or gradients across a pixel quad may increase texture sampling time and have an overall benefit.

2x2 Pixel Processing

The dFdx() / dFdx() functions (supported with the GL_OES_standard_derivatives extension) are used to detect the rate of change of a given register across adjacent pixels in a horizontal (dFdx) and vertical direction (dFdx). In other words: how much does this value change from pixel to pixel, in either the screen-x or screen-y position? This is mostly used for texture LOD calculations, and typically for factors like choosing what mipmap level to sample if you are doing a complicated effect where the default mip level selection won't work as expected.

The scan converter “rough scans” each triangle into aligned 8x8 tiles as shown in Figure **4-1**.

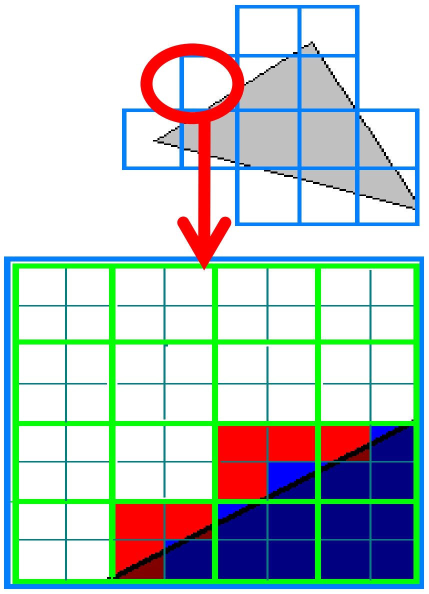


Figure 4-1 Scan converter “rough scans” each triangle

After this each tile is broken up into aligned 2x2 quads. Some of these quads might contain “dead” fragments as shown in red in Figure 4-1.
Very small triangles that are less than 2x2 fragments in size can waste at least 60% of the GPU’s processing power. Consider using geometry LODs that use only bigger triangles. So, for example, when implementing the particle system, using smaller triangles could reduce efficiency.

Occlusion Queries

Occlusion queries make it possible for an application to ask whether or not any pixels would be drawn if a particular object were rendered. With this feature, applications can check whether or not the bounding boxes of complex objects are visible; if the bounds are occluded, the application can skip drawing those objects. Hardware occlusion queries are appealing in today's games because they work in completely dynamic scenes. They are supported in OpenGL ES 3.0. The AMD_performance_monitor extension also provides a very similar functionality by showing if the depth test passes or fails.

Texture Sampling

The fact that the hardware works on 2x2 fragments at the same time offers a challenge on the fragment level. Four or more of those quads are grouped into a fragment vector (aka fragment thread). Optimizing meshes so that fragment vectors are nicely grouped is better for the texture cache.

Other strategies to avoid texture stalls are:

- Avoid random access
- Avoid volume textures
- Avoid fetching from 7 textures at once … more like 4
- Use compressed formats everywhere: much better memory usage, less stalls
- Use mipmaps when possible

In general, trilinear and anisotropic filtering is much more expensive than bilinear filtering, while there is no difference between point and bilinear filtering.

Texture filtering can influence the speed of texture sampling. A bilinear texture look-up 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. Because anisotropic filtering is adaptive, this hit is taken only on fragments that require anisotropic filtering that might end up being only a few fragments total. A rule of thumb for real-world cases is that anisotropic filtering will be less than twice the cost on average.

Different texture formats have a large impact on texture sampling performance. A 32-bit texture costs one cycle to fetch. So all 32-bit and smaller formats (including all the compressed formats) are single-cycle. A 64-bit format takes two cycles and a 128-bit format takes four cycles to fetch.

Cube maps and projected texture lookups do not incur any extra cost, while shader specific gradients – based on dFdx() / dFdx() - cost an extra cycle. That means a regular bilinear lookup that normally takes one cycle takes two with shader specific gradients. Please note that these shader-specific gradients cannot be stored across lookups, so if you do a texture lookup with the same gradients again in the same sampler, it will cost the one cycle hit again.

3D Textures

Use of 3D textures can severely impact memory and cache usage. Making effective use of texture memory is already a major task for real-time 3D content. Adding another dimension to a texture map increases its memory footprint significantly. There are four major techniques that help to improve 3D texture mapping performance:

- Keep textures small (< 32 texels in each direction)
- Repeat and mirror where appropriate. Volume detail textures can be repeated to great effect. Use mirroring address modes for symmetrical textures like volumetric light maps. The “mirror once” (GL_OES_texture_npot) addressing mode is especially useful with 3D light maps.
- Keep texture bit-depth as low as possible. Volumetric detail textures and light maps are often grayscale. For these, use a single-channel texture.
- Use compression. Developers should make the appropriate size/quality tradeoff for their application.

Threads in Flight / Dynamic Branching

Branching is quite crucial for the performance of the shader. Every time the branch encounters ‘divergence’ (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 any fragment shaders.

Pack Shader Interpolators

Shader interpolated values, or varyings, require a GPR to hold data being fed into a fragment shader. Therefore, you should minimize their use. Use constants where a value is uniform. All varyings have four components, whether you use them or not, so pack values together. Putting two vec2 texture coordinates into a single vec4 value is a common practice, but other strategies employ more creative packing.

Stream Out

The stream out functionality is a fixed-function unit that can store data from the vertex shader into a 1D buffer. This functionality is used for binning and might help with skinning and other cases when vertex data needs to be looped around the shader unit to do another pass.

Draw Skybox Last

Many games are still rendering the skybox first. This used to be the fastest way several hardware generations ago because it filled the color buffer directly without doing the depth test. However, on modern hardware the skybox should be rendered as late as possible in the frame. Only transparent objects blended against the skybox may be rendered after it. The reason is that the skybox itself tends to be occluded by large parts of the scene. If you render the skybox first a lot of pixels are being shaded that will only be overwritten later. This wastes pixel processing and bandwidth. By rendering the skybox at the end, only the necessary pixels will be shaded. When rendering the skybox last, you may want to peg it to the far clipping plane to avoid any issues with the skybox cutting into the scene. This can be accomplished easily in two ways. Copy the fourth row of the MVP matrix into the third so that Z and W evaluates to the same value and hence Z / W becomes 1.0. Alternately, copy the W value into Z in the vertex shader.

Level-of-Detail Shaders

To improve shader efficiency, similar to a geometric Level-of-Detail, a shader Level-of-Detail can be implemented. Based on the distance from the camera, the quality, cost and energy efficiency of shaders are exchanged. The farther the object is from the camera, the less the cost and quality will be, while increasing the energy efficiency.

Granularity of the LOD system can be based per frame or per object. The LOD value algorithm would follow the distance from the camera. Even a dynamic component in the form of the overall system workload can be included. For a lighting system, the LOD levels could be:

1. Normal-mapped per-pixel lighting with additional detailed maps attached
2. Normal-mapped per-pixel lighting without detailed map
3. Vertex lit with a color texture only
4. Vertex lit with diffuse-component only (the specular component of the lighting is dropped) and only one color texture.

Minimize Shader GPRs

Minimizing GPRs can be the most important means of optimizing performance. Inputting simpler shaders to the compiler helps guarantee optimal results. Sometimes, modifying GLSL to save even a single instruction can save a GPR. Not unrolling loops can also save GPRs, but that is up to the shader compiler. (Conversely, unrolled loops tend to naively lump texture fetches toward the top of the shader, resulting in a need for more GPRs to hold the multiple texture coordinates and fetched results simultaneously.)

Minimize Shader Instruction Count

The compiler optimizes specific instructions, but does not automatically do so in the most efficient way. Analyze shaders to save instructions wherever possible. Saving even a single instruction is worth the effort.

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 your own shaders and move those calculations off to the CPU. It may be much easier to identify math on shader constants in the post-compiled microcode.

Avoid Uber-Shaders

Uber-shaders combine multiple shaders into a single shader that uses static branching. Using them makes good sense if you are trying to reduce state changes and batch draw calls. However, this often comes at the expense of an increased GPR count, which has an impact on performance.

Avoid Killing Pixels (Discard) in the Fragment Shader

Some developers believe that manually killing (discarding) 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.

Break Up Draw Calls

If a shader is heavy on the GPRs and/or heavy on the texture cache demands, increased performance may result from breaking up the draw calls into multiple passes. Whether or not the results will be positive is very hard to predict, so using real-world measurements both ways is the best way 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; notably, the GMEM must be resolved for a previous pass to be used as input to a successive pass. Since resolves are not free, it is a performance cost that must be recouped elsewhere in the algorithm.

Using Mixed Precision

Even though it is recommended to use mediump and lowp as precisions for shaders, there may be situations when one has to use highp (high precision) for certain varyings (texture coordinate as an example) in shaders. Such situations can be handled with a conditional statement and a preprocessor-based MACRO definition, such as:

```
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
```

5 Using OpenGL ES Extensions

5.1 Overview

OpenGL ES is an extensible, low-level graphics API. Extensions provide OpenGL ES application developers with new rendering features above and beyond the features specified in the official OpenGL ES standard. OpenGL ES extensions keep the OpenGL ES API current with the latest innovations in graphics hardware and rendering algorithms.

This section describes the OpenGL ES extension mechanism and the set of OpenGL ES extensions supported on Adreno 3xx.

How OpenGL ES Extensions are Documented

An OpenGL ES extension is defined by its specification. The specifications are typically written as standard ASCII text files. A well-written OpenGL ES specification of an extension is documented to the level of detail needed by a hardware designer or OpenGL ES programmer. The specification does not cover why the extension exists or a tutorial on how it is implemented.

Where to Find OpenGL ES Extension Specifications

The latest public OpenGL ES specifications can be found on the Khronos website. Please note that extension specifications are updated periodically based on reviews and implementation feedback.

How to Read an OpenGL ES Extension Specification

When reading an OpenGL ES extension specification, it helps to be familiar with the original OpenGL ES specification. The original specification of OpenGL ES can be found on the Khronos website.

Each OpenGL ES Extension Specification follows the same pattern.

i. Name – the name of the extension; e.g., OES_depth_texture
ii. Name Strings - the name string used by OpenGL ES; e.g. GL_OES_depth_texture
iii. Contact – who to contact regarding this extension; usually an e-mail address
iv. Status – the status of the extension; e.g., Ratified by the Khronos BOP, March 20, 2008.
v. Version – the last date when it was modified; e.g., Last Modified Date: October 8, 2009
vi. Number – each OpenGL ES extension is assigned a unique number. Those numbers ensure that OpenGL extensions do not overlap in their use of enumerants or protocol tokens; e.g. 44

vii. Dependencies – often an extension specification builds on the functionality of pre-existing extensions or on a certain core version of OpenGL ES; e.g., OpenGL ES 2.0.

viii. Overview – this section provides a description from a bird's-eye view. The section answers the question as to what an extension is actually doing the fastest.

ix. Issues – this section describes open issues and states the resolution to resolved issues. These issues are often items of interest to the extension implementer, but can also help a programmer understand how the extension really works or show the restrictions of the extension; e.g., the advantages of supporting manual generation of mipmaps of a depth texture using GenerateMipmap.

x. New Procedures and Functions – this section lists the function prototypes for any new procedures and functions that the extension adds; e.g., boolean IsRenderbufferOES(uint renderbuffer).

xi. New Tokens – this section lists the tokens (also called enumerants) that the extension adds; e.g., FRAMEBUFFER_OES 0x8D40.

xii. Additions to Chapter of the OpenGL ES 1.x or 2.x Specification – this section documents how the core OpenGL ES specification should be amended to add the extension's functionality to the core OpenGL ES functionality; e.g., Additions to Chapter 2 of the OpenGL ES 2.0 Specification (OpenGL Operation).

xiii. Dependencies – this section describes how the extension depends on some other extension; e.g., Additions to Chapter 2 of the OpenGL ES 2.0 Specification (OpenGL Operation).

xiv. Errors – this section describes new error conditions particular to the extension; e.g., INVALID_ENUM is generated if RenderbufferStorageOES is called with an unsupported <internalformat>.

xv. New State – extensions typically add new state variables to OpenGL ES's state machine; e.g., The queries for NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS include 3DC_X_AMD and 3DC_XY_AMD.

xvi. Revision History – this section usually has a revision history that details the date and the person responsible for the revision, as well as a short note on what was revised; e.g.,

02/25/2005	Aaftab Munshi	First draft of extension
04/27/2005	Aaftab Munshi	Added additional limitations to simplify OES_framebuffer_object implementations
07/06/2005	Aaftab Munshi	Added GetRenderbufferStorageFormatsOES removed limitations that were added to OES version of RenderbufferStorage, and FramebufferTexture2DOES.

How to Access OpenGL ES Extensions

Access the extensions through the header files as below. These header files can be found as part of the Adreno SDK.

```
#include <GLES2/gl2ext.h> for OpenGL ES 2.0 extensions
#include <GLES2/glext.h> for OpenGL ES 1.0 extensions
```

5.2 Extensions Supported on Adreno 3xx

Adreno 3xx supports the following optional OpenGL ES 1.x extensions:

Extension	**Short Description**
GL_AMD_performance_monitor	Capture and reporting of performance monitors
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
GL_OES_compressed_paletted_texture	Enables direct support of palletized textures in OpenGL ES

Extension	Short Description
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

Extension	Short Description
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
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

OpenGL ES 2.0 Extensions Supported on Adreno 3xx

Adreno 3xx supports the following optional OpenGL ES 2.0 extensions:

Extension	Short Description
GL_AMD_compressed_ATC_texture	Enables support for ATC compressed texture formats
GL_AMD_performance_monitor	Enables the capture and reporting of performance monitors
GL_AMD_program_binary_Z400	Provides a program binary format, Z400_BINARY_AMD
GL_EXT_texture_filter_anisotropic	Permits the OpenGL application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering

Extension	Short Description
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 GL command completion than standard OpenGL
GL_OES_compressed_ETC1_RGB8_texture	Provides direct support of compressed textures in the Ericsson Texture Compression (ETC) formats 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_EGL_image	Provides a mechanism for creating texture and renderbuffer objects that share storage with specified EGLImage objects
GL_OES_EGL_image_external	Provides a mechanism for creating EGLImage texture targets from EGLImages
GL_OES_element_index_uint	Provides support for UNSIGNED_INT <type> values for DrawElements
GL_OES_fbo_render_mipmap	Enables rendering to any mip-level of texture that is attached to a framebuffer 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	Enables the application to use the GL itself 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 RGB8 and RGBA8 renderbuffer storage formats
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 16- (aka half float)

Extension	Short Description
	and 32-bit floating-point components
GL_OES_texture_half_float	Provides 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_npot	Provides support for the REPEAT and MIRRORED_REPEAT texture wrap modes and the minification filters supported for NPOT 2D textures, cubemaps and for 3D textures, if the OES_texture_3D extension is also supported
GL_OES_vertex_array_object	Introduces vertex array objects, which encapsulate VBOs
GL_OES_vertex_half_float	Adds a 16-bit floating point data type (aka half float) to vertex data specified using vertex arrays
GL_OES_vertex_type_10_10_10_2	Provides the 10:10:10:2 data format to vertex types
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 enables 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_perfmon_global_mode	A Qualcomm specific extension that enables performance monitoring
GL_QCOM_extended_get	A Qualcomm specific extension that enables instrumenting the driver for debugging of OpenGL ES applications
GL_QCOM_extended_get2	A Qualcomm specific extension that enables instrumenting the driver for debugging of OpenGL ES applications
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

6 Optimizing for the Platform

6.1 Overview

Optimizing a graphics application can be a long and tricky trial-and-error process. Fortunately, Adreno Profiler is able to point you to the specific part of the application that needs to be optimized. The Adreno SDK also has multiple utilities and resources that can help you optimize your application. See sections 3.2.1 and 3.2.2 for details on these tools.

Apart from these tools, the next section covers some of the high-level tips and tricks that one should know and use as a standard programming practice.

6.2 Optimization Guidelines

6.2.1 General

Minimize GL State Changes

To reduce command buffer space, applications should avoid redundantly setting a state. Constantly saving/restoring a state prior to rendering meshes and draw calls can cause needless bloat. Texture thrashing is costly, so sort your render calls by texture and then by state vector.

Minimize Number of Draw Calls

There is some overhead for each draw call and, on Android, each call to native code from Java. This can be reduced by minimizing the number of strips needed to describe any set of triangles that share the same state. Disjoint strips can be joined together by degenerate triangles. Using the Qstrip tool to optimize your geometry is suggested. Read section 3.2.1 to learn more about Qstrip tool.

Avoid Unnecessary Clear Calls

In the glClear() driver, calls are implemented as viewport covering quads to write the clear values into the frame buffer. Clears operate, therefore, at the normal pixel rate. For 24:8 depth buffers, the z-buffer and the stencil-buffer should always be cleared together. Redundant clears are costly, so they should be avoided.

Always clearing the depth-buffer at the start of a frame is imperative.

Avoid Unnecessary MakeCurrent Calls

Calling the eglMakeCurrent() call every frame within your application is not recommended.

Qualcomm's SIMD/FPU Co-Processor Usage

Snapdragon CPU cores support an additional VeNum a SIMD/FPU co-processor which is instructionally compliant with ARM's NEON™ instruction set. More information on VeNum and Neon is in section 2.1 of this document where we talk in detail about the CPU. This SIMD/FPU unit can be coded for jobs such as skinning, simulations, or physics to offload the calculations from the CPU or vertex shaders. 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.

Implement Multiple Threads

Snapdragon processors today support multiple CPU cores. Use multiple threads in your engine to take advantage of multiple CPU cores.

6.2.2 Rendering and Geometry

Use Front-to-Back Rendering

The 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 overkill, but, drawing the sky box last and terrain second-to-last, for example, is a sensible strategy.

Use Triangle Strips

Triangle strips are the geometry format for optimum throughput. We recommend using glDrawElement() with triangle strips. It is suggested that you use the Qstrip tool to optimize your geometry.

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. In other words, spend shader instructions on objects that matter. In practice, this means simplifying the scene as much as possible, like using baked-in lighting.

Use Back Face Culling

A simple, but sometimes overlooked, 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, non-planar objects, back faces should always be culled. Note that the deferred rendering algorithm obeys the GL cull mode, so back face culled polygons can improve vertex processing performance as well.

Use Vertex Buffer Objects

Using Vertex Buffer Objects (VBO) instead of client-side vertex arrays is the best way to use the hardware for performance. The use of client side vertex arrays comes at a significant cost to performance since the CPU is used to copy the vertex data into the GPU for each draw call, causing a tremendous drain on memory bandwidth and killing performance. Using VBOs avoids the additional overhead.

Use Direct Mode Rendering When Applicable

As explained in the FlexRender topic of section 2.2.1 , Adreno 3xx GPUs support both deferred and direct rendering. So if the application doesn't involve heavy overdraw (multiple layers of rendering) then it is recommended to use direct rendering with the extension as explained in section 2.2.1

Avoid Excessive Clipping

Avoid excessive clipping, especially for static geometry, by further dividing your geometry into more primitives. For example, in Figure 6-1, a static room is changed from 722 triangles to 785 triangles. More triangles were added to the overall geometry, but the performance is improved because there was no excessive clipping of the oversized triangles.

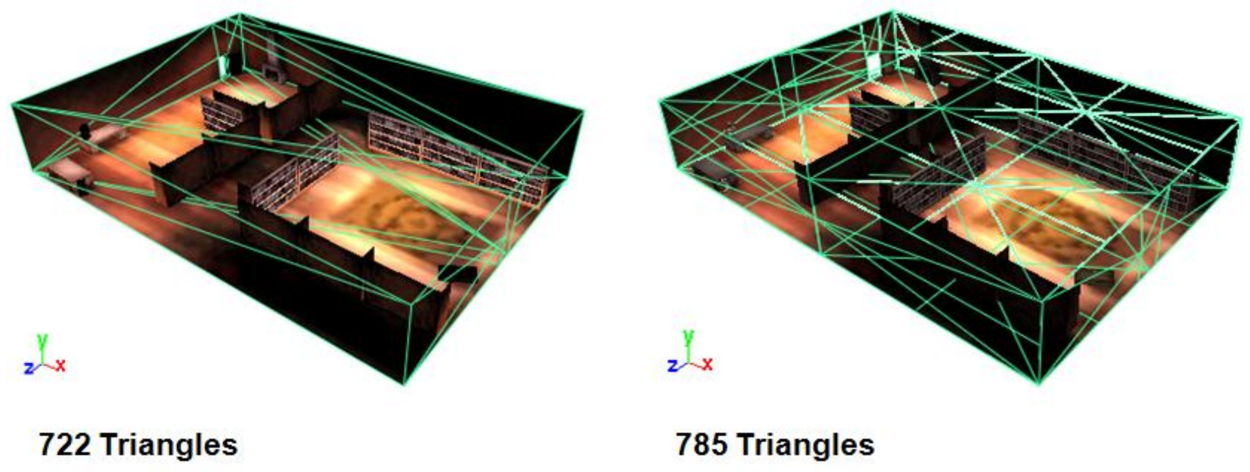


Figure 6-1 Geometry design to avoid clipping

Use Interleaved, Compressed Vertices

For vertex fetch performance, interleaved vertex attributes (xyz, uv, xyz, uv, … rather than xyz, xyz, xyz,…. uv, uv, uv, …) are best. The throughput is better with interleaved vertices, and compressing vertices is even better. Deferred rendering gives these optimizations a big advantage. There is support for half float coordinates in vertex buffers. These can be used for texture or normal coordinates where lower precision will not hurt the final imagery.

Use Level-of-Detail (LOD) in Effects

When we think of LoD systems, we think of the polygon count of meshes. That is good practice, and should be employed for mobile applications. In addition, consider having an LoD system for shader effects. For example, fade away specular highlights and normal mapping as an object recedes.

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 shadowmap. 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 still need to be run to determine if this will be as successful on the Adreno.

6.2.3 Textures

Compress Textures

Texture cache friendliness requires that textures be compressed whenever possible. Note that FBOs (render-to-texture) are not compressed. While CPU time could be spent to compress resolved FBO textures, the CPU-GPU synchronization and the actual compression time would make real-time rendering impossible. Use tools like Adreno Texture Converter in Adreno SDK to compress textures and normal maps to ATC format.

Use Mipmapping

Mipmaps are also important for texture cache performance. Note that OpenGL ES 1.x lacks an API to generate mipmaps at runtime (which would result in unacceptably long load times), so that mipmaps need to be generated offline for OpenGL ES 1.x. The SDK's ResourcePacker tool provides a source code example of how to do this. OpenGL ES 2.0 has an API (glGenerateMipmap) to generate such mipmaps. In addition, Adreno Texture Converter could be used for dynamic/offline mipmap generation along with compression.

Use Multi-Texturing

On the Adreno GPUs, multiple textures (as many as 16) can be used in a single render pass. Blending is an inexpensive effect, so use multiple textures for possible effects such as static lighting instead of dynamic lights.

Use Texture Stacks

Texture stacks are a unique GPU feature that can assign an array of textures to a single texture stage. (Cubemaps are actually just a stack of six textures.) The shader can simply use a third texture coordinate to select which texture it wishes to sample from. Using texture stacks can simplify state changes and increase the batching of draw calls. In other words, stacked textures can be used to minimize texture cache thrash for materials that have multiple textures. Each layer of textures in the stack can be a separate texture needed by the material. The fragment shaders would have to be written such that the texture z coordinate is chosen to select the appropriate texture.

6.2.4 Off-Screen Rendering

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 will dictate the ideal frame buffer resolution.

Use FBOs, also known as Render-to-Texture, 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 crisply at the native resolution. This approach is not uncommon in console games, in both previous and current generations. This provides the flexibility to choose a lower resolution, since the texture filter hardware smooths out the image.

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 below.

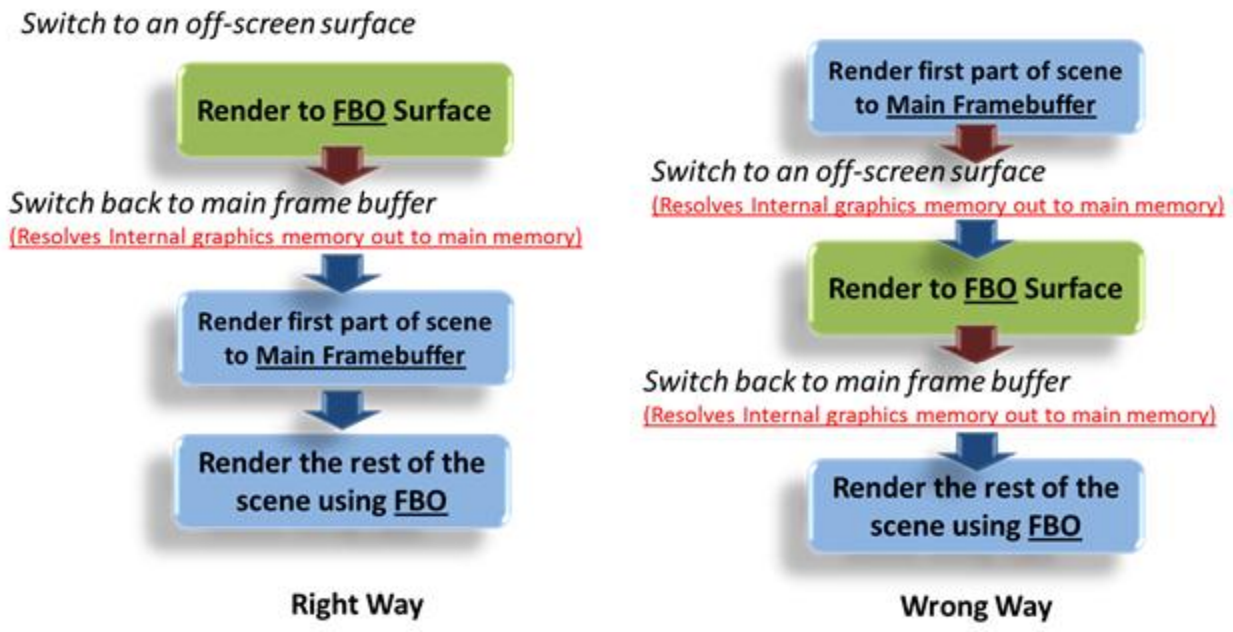

Switch to an off-screen surface
Render to FBO Surface
Switch back to main frame buffer
(Resolves Internal graphics memory out to main memory)
Render first part of scene to Main Framebuffer
Render the rest of the scene using FBO
Right Way
Render first part of scene to Main Framebuffer
Switch to an off-screen 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

6.2.5 Shaders

See chapter 4 for detailed consideration and optimizational guidelines on shaders.

6.2.6 Advanced Visual Effects

To implement advanced visual effects, experimentation and benchmarking are required to assess your 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.

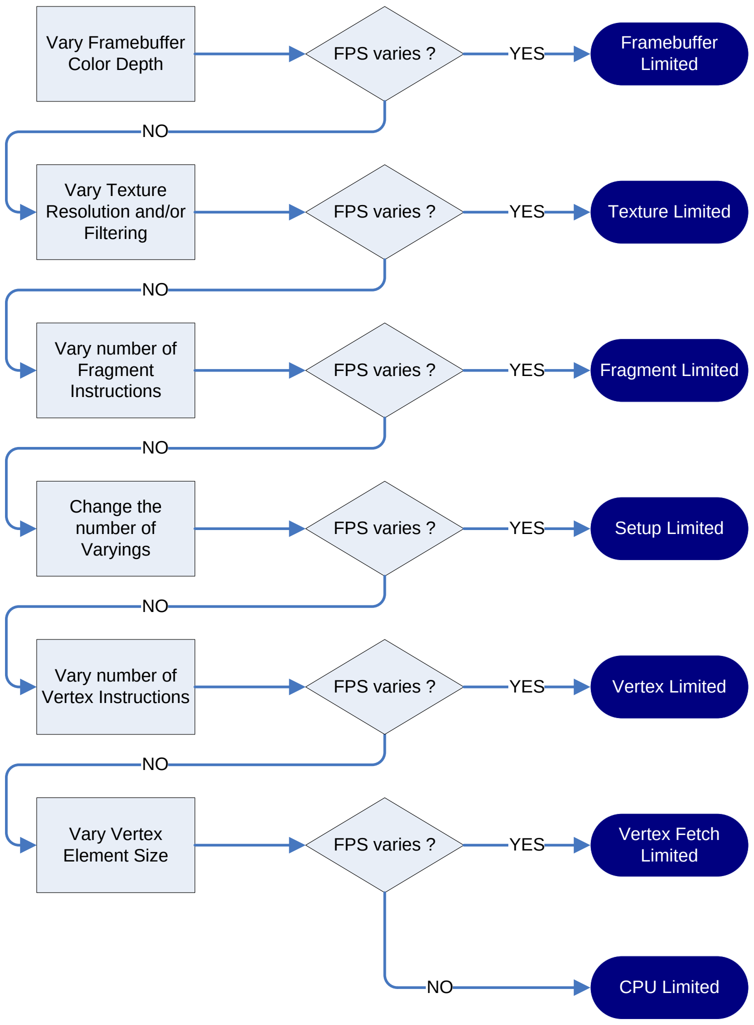


Figure 6-3 Identifying application bottlenecks in GPU

After characterizing the algorithm for a particular device, following are some tips for implementing advanced visual effects.

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 very 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 measurable difference. Vertex shaders can pre-fetch 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 full-size 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 not much 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 very strong blurred effect.

Advanced Lighting / Shadowing

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

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

Storing 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.

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. For example, a detailed shading may not be necessary on an object until the camera in the scene is not close to that particular object. So when the object is at a distance, a simpler shading can be used.

7 References

Reference documents, which may include Qualcomm standards and resource documents, are listed below. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Ref.	**Document**	
Standards		
S1	*Khronos' "The OpenGL Graphics System: A Specification"*	
S2	*Khronos' "The OpenGL Graphics System: A Specification"*	
Resources		
R1	*"OpenGL SuperBible Fourth Edition"*	Addison Wesley
R2	*"OpenGL ES 2.0 Programming Guide"*	Addison Wesley
R3	"GPU Gems 3"	Addison Wesley

8 Revision History

Revision	Date	Description
A	July 2012	Initial release