DirectX® Application Developers' Guide for Adreno™ GPUs

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

Contents

1 About this Document

This document is a guide for developing and optimizing DirectX applications for Windows RT on Snapdragon™ S4-based mobile platforms that include the Adreno™ series GPUs (Graphics Processing Units). The Adreno 225 GPU is embedded within Qualcomm's Snapdragon S4 (MSM8960) processor which is the first Snapdragon processor to support Microsoft's Windows RT platform.

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

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 its integrated Adreno GPU 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.

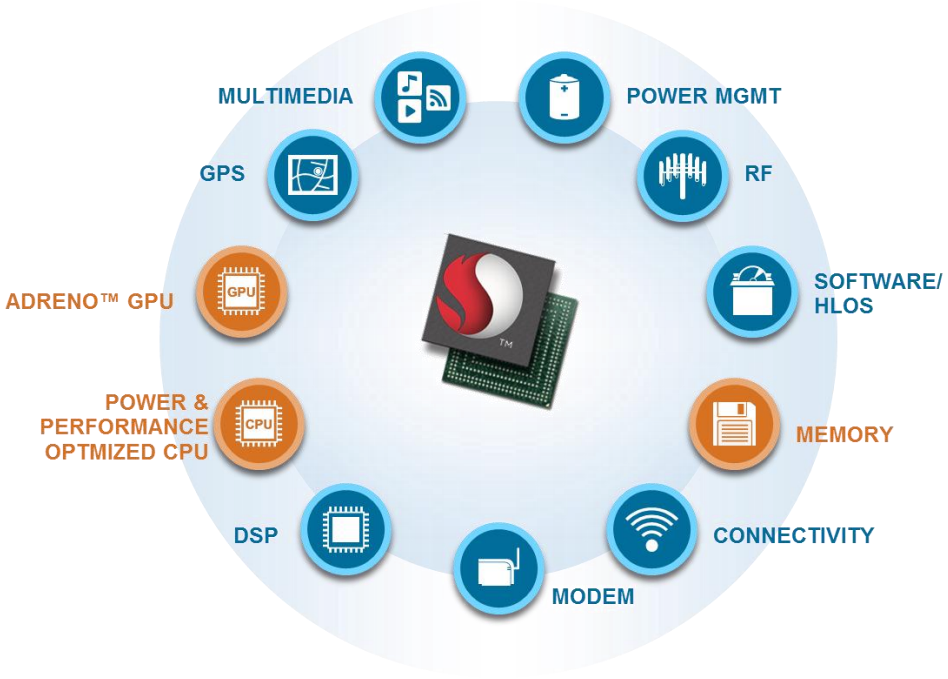


Figure 2-1 Components within Snapdragon system-on-chip processors

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 programming the GPU with Direct3D 11.1 on Windows RT and touches upon other related components like CPU and memory that are needed for graphics application development.

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.

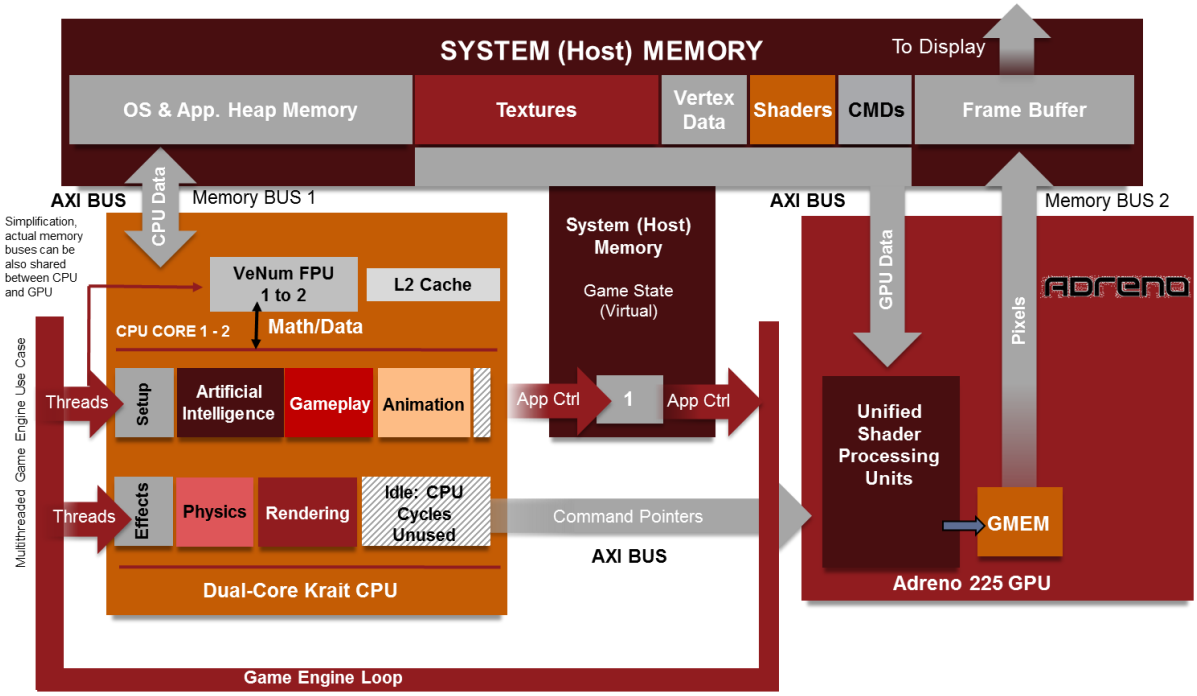


Figure 2-2 Overall system-level flow for graphics applications on Snapdragon S4 (MSM8960)

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 dual-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 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 onto 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 two components: GMEM (Internal Graphics Memory) sometimes also referred to as OCMEM (On Chip Memory), which is a tile buffer for storing color, depth and stencil values, and the shader processing unit.

Adreno GPUs have an internal local memory buffer that can store z, stencil and color values. The tile-based rendering architecture in Adreno GPUs allows for dividing the final frame buffer into smaller portions, called bins. Resolving these bins one at a time in this local buffer is called tile-based. A completely resolved final frame buffer is then stored in system memory as shown.

Another architectural highlight of Adreno GPUs is a unified shader architecture which allows for efficient shader processing. 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.

Display

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

2.2 Adreno Graphics Subsystem for DirectX

The Adreno GPU exceeds the requirements of Direct3D11.1 feature level 9_3. This section details the features of the Adreno GPU that are accessible through the Direct3D API.

Texturing Features

Texture Formats

The following table lists the texture formats supported by Direct3D11.1 feature level 9_3. For each format, the table specifies whether the format can be used for a 2D, 3D, or Cubemap texture. The *Filter* column specifies whether the format can be used with a sampler that has non-point min or mag filtering enabled. The *Mip* column specifies whether the format can be mipmapped, and the *GenMip* column specifies whether the driver can auto-generate mipmaps for the format. Finally, the *Render* column lists whether the format can be used as a render target.

Texture Format	3D	2D	Cube	Filter	Mip	GenMip	Render
DXGI_FORMAT_R32G32B32A32_FLOAT	X	X	X		X	X	X
DXGI_FORMAT_R16G16B16A16_FLOAT	X	X	X		X		X[2]
DXGI_FORMAT_R16G16B16A16_UNORM	X	X	X	X	X		X
DXGI_FORMAT_R32G32_FLOAT	X	X	X				
DXGI_FORMAT_R8G8B8A8_UNORM	X	X	X	X	X	X	X[12]
DXGI_FORMAT_R8G8B8A8_UNORM_SRGB	X	X	X	X	X	X	X[12]
DXGI_FORMAT_R8G8B8A8_SNORM		X	X	X	X		
DXGI_FORMAT_R16G16_FLOAT	X	X	X		X		X
DXGI_FORMAT_R16G16_UNORM	X	X	X	X	X	X	X
DXGI_FORMAT_R16G16_SNORM	X	X	X	X	X		
DXGI_FORMAT_R32_FLOAT	X	X	X		X		X

Texture Format	3D	2D	Cube	Filter	Mip	GenMip	Render
DXGI_FORMAT_R16_UNORM	X	X	X	X	X		
DXGI_FORMAT_R8G8_SNORM		X		X	X		
DXGI_FORMAT_R8_UNORM		X	X	X	X		
DXGI_FORMAT_BC1_UNORM	X	X	X	X	X		
DXGI_FORMAT_BC1_UNORM_SRGB		X	X	X	X		
DXGI_FORMAT_BC2_UNORM		X	X	X	X		
DXGI_FORMAT_BC2_UNORM_SRGB		X	X	X	X		
DXGI_FORMAT_BC3_UNORM		X	X	X	X		
DXGI_FORMAT_BC3_UNORM_SRGB		X	X	X	X		
DXGI_FORMAT_B8G8R8A8_UNORM	X	X	X	X	X	X	X[12]
DXGI_FORMAT_A8_UNORM	X	X	X	X			
DXGI_FORMAT_B5G6R5_UNORM		X	X	X	X	X	X[12]
DXGI_FORMAT_B5G5R5A1_UNORM		X	X	X	X		
DXGI_FORMAT_B4G4R4A4_UNORM		X	X	X	X		

[1] Formats tagged with [1] can be used as multisample render targets.

2 Formats tagged with [2] are render targets that support blending.

Texture Compression

Compressing textures can significantly improve the performance and load time for graphics applications since it reduces texture memory and bus bandwidth use. Important compression texture formats that are supported by Adreno GPUin Direct3D 11.1 are:

- **BC1** – DXT1 texture compression format (4-bit per pixel for RGB with 0 or 1-bit of alpha)
- **BC2** – DXT2 texture compression format (8-bit per pixel for RGBA). In BC2, the alpha is stored as an uncompressed 4-bit value.
- **BC3** – DXT5 texture compression format (8-bit per pixel for RGBA). In BC3, the alpha is compressed using a block compression technique.

In previous versions of Direct3D, the D3DX library provided useful utility methods for loading compressed textures from DDS files. However, Microsoft deprecated D3DX and did not include it in the Windows 8 SDK. However, Qualcomm provides support for compressing textures to the BC1/BC2/BC3 format in the *Adreno Texture Converter* tool as part of Adreno SDK.

Large Texture Size

Adreno GPU supports texture sizes as big as 4096 x 4096 texels.

Multiple Render Targets

Adreno GPU supports up to four render targets simultaneously together with a depth/stencil buffer. The following depth/stencil formats are exposed through Direct3D11.1 feature level 9_3.

Depth/Stencil Format	Description
DXGI_FORMAT_D24_UNORM_S8_UINT	24-bit depth interleaved with 8-bit stencil
DXGI_FORMAT_D16_UNORM	16-bit depth-only

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. Adreno GPU supports sRGB color space for render targets as well as textures, making this color-correct viewing experience possible.

sRGB color space is supported in Direct3D11 using the texture formats and render targets with format DXGI_FORMAT_R8G8B8A8_UNORM_SRGB, DXGI_FORMAT_BC1_UNORM_SRGB, DXGI_FORMAT_BC2_UNORM_SRGB, and DXGI_FORMAT_BC3_UNORM_SRGB.

Geometry Features

Vertex Formats

The following table lists the formats that can be used for vertex data in Direct3D 11.1 Feature Level 9.3.

Vertex Format	Description
DXGI_FORMAT_R32G32B32_FLOAT	3-component 32-bit float
DXGI_FORMAT_R32G32B32A32_FLOAT	4-component 32-bit float
DXGI_FORMAT_R16G16B16A16_FLOAT	4-component 16-bit half float
DXGI_FORMAT_R16G16B16A16_SNORM	4-component 16-bit signed normalized integer [-1.0, 1.0]
DXGI_FORMAT_R16G16B16A16_SINT	4-component 16-bit signed integer [-32768, 32767]
DXGI_FORMAT_R32G32_FLOAT	2-component 32-bit float
DXGI_FORMAT_R8G8B8A8_UNORM	4-component 8-bit unsigned normalized integer [0, 1.0]
DXGI_FORMAT_R8G8B8A8_UINT	4-component 8-bit unsigned integer [0, 255]
DXGI_FORMAT_R16G16_FLOAT	2-component 16-bit half float
DXGI_FORMAT_R16G16_SNORM	2-component 16-bit signed normalized integer [-1.0, 1.0]
DXGI_FORMAT_R32_FLOAT	1-component 32-bit float

Index Formats

Index buffers can be stored as 16-bit USHORT or 32-bit UINT using the formats DXGI_FORMAT_R16_UINT and DXGI_FORMAT_R32_UINT.

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 2-3 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 2-3 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.

Geometry instancing can be implemented using *ID3D11DeviceContext::DrawIndexedInstanced()* in Direct3D11 Feature Level 9_3.

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 2-4. 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.

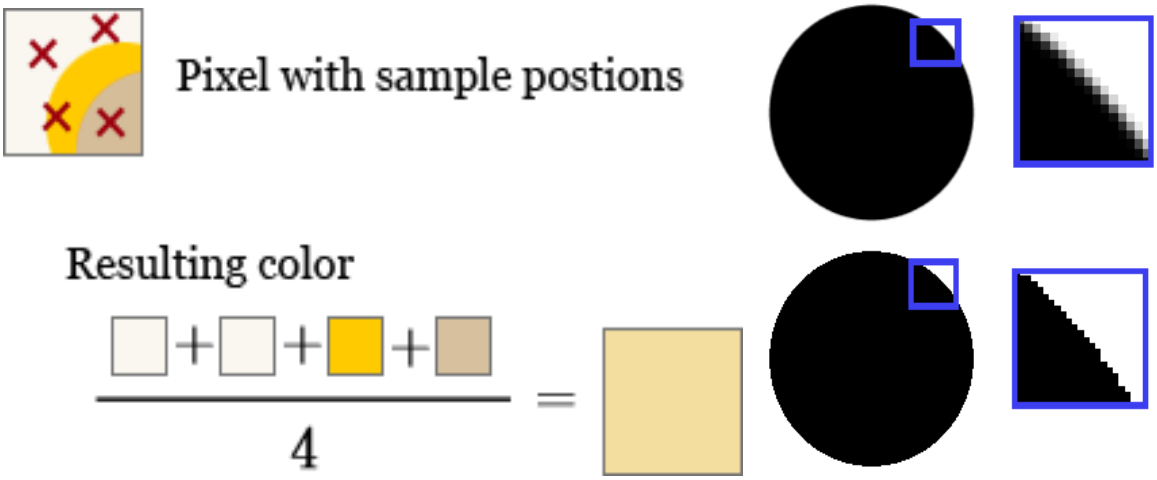


Figure 2-4 MSAA

Among the various techniques for reducing aliasing effects, multisampling is the one supported by Adreno GPU. 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 2-4. Adreno GPU supports two or four samples.

In Metro applications, MSAA cannot be performed on the main back buffer. Rather, the application must render to a multisampled offscreen render target, resolve it to a non-multisampled render target using *ID3D11DeviceContext::ResolveSubresource()*, and then render the non-multisample version to the backbuffer as a texture bound to a fullscreen quad. This technique is demonstrated in the MSAA example in the Adreno SDK for DirectX.

3 Tools and Resources

This section describes developer tools from Microsoft as well as from Qualcomm that are available for graphics application development and analysis on Windows RT platforms powered by Snapdragon. Microsoft's tools are available on Microsoft's MSDN website http://msdn.microsoft.com while Qualcomm-provided tools are available for free download on the Qualcomm Developer Network: http://developer.qualcomm.com.

3.1 Microsoft Direct3D11.1 Tools

In the past, Microsoft created a separate installer for the Microsoft DirectX SDK. However, starting with Windows 8, Microsoft now includes Direct3D as part of the standard Windows SDK. Microsoft has also made many improvements in providing integrated Direct3D tools in Visual Studio 2012.

HLSL in Visual Studio 2012

Visual Studio 2012 provides syntax highlighting for developing shaders in HLSL. Further, the HLSL compiler is directly integrated into the Visual Studio build system such that the shaders will be compiled using *fxc* (the Microsoft HLSL compiler) at build time. The build rules for HLSL files contain options including setting the *Shader Type* and *Shader Model*. For example, an HLSL pixel shader compiled for Adreno GPU would be set to *Shader Type* of "Pixel Shader (/ps)" and *Shader Model* "Shader Model 4 Level 9_3 (/4_0_level_9_3) as shown in Figure 3-1.

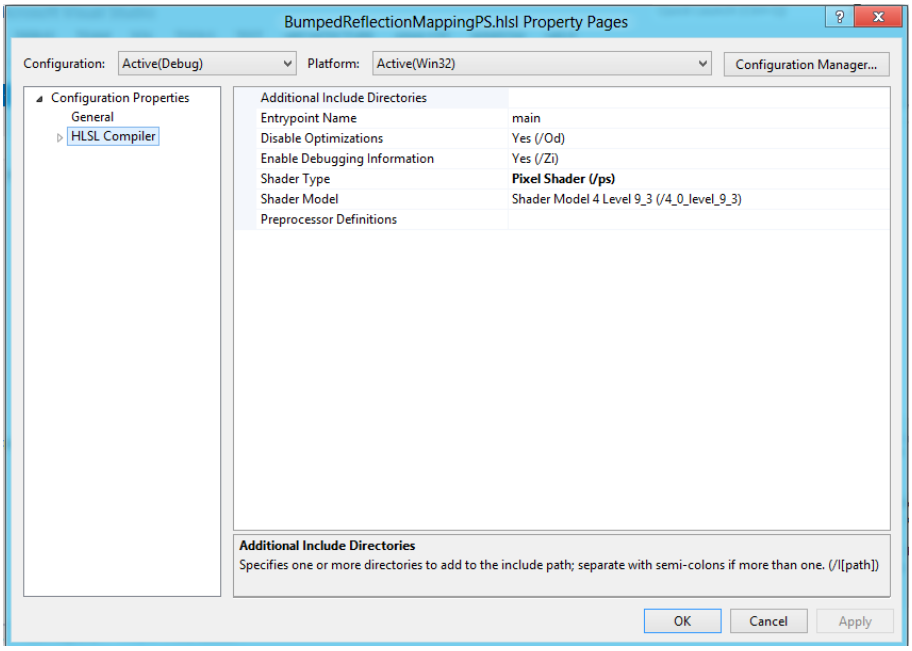


Figure 3-1 HLSL in Visual Studio 2012

In previous versions of Direct3D, it was possible for the application to compile the HLSL shader code at runtime using D3DX. However, in Metro applications, the HLSL must be compiled ahead of time using *fxc*. The build process highlighted above will produce a binary shader file (usually named '.cso' for Compiled Shader Object) that can be loaded at runtime. Microsoft does provide a debugging D3DCompiler API that can be used in debug builds of an application to compile the HLSL code at runtime, but this API cannot be used by released Metro applications.

Visual Studio Graphics Debugger

Visual Studio 2012 also contains a built-in Graphics Debugger (formerly PIX for Windows) that includes performance and debugging functionality for Direct3D 11.1. The Graphics Debugger can be invoked in Visual Studio 2012 from the *Debug * Graphics* menu by clicking on *Start Diagnostics (ALT+F5)* as shown in Figure 3-2.

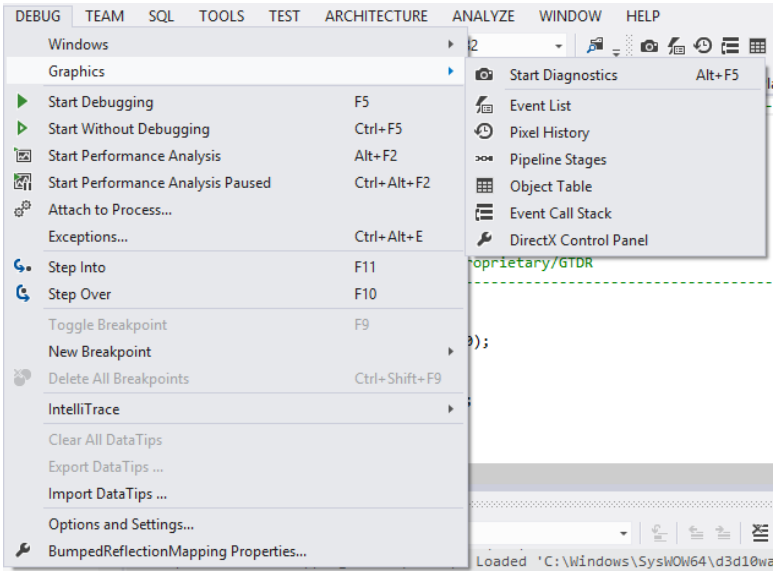


Figure 3-2 Visual Studio Graphics Debugger

Microsoft provides a complete guide to using its Graphics Debugger at http://msdn.microsoft.com/en-us/library/hh315751(v=vs.110).aspx.

DirectX Debug Layer

The Microsoft Windows SDK includes a debugging layer for DirectX that can be used to identify many types of Direct3D problems. All of the Adreno SDK for DirectX examples enable the Debug SDK layer for _DEBUG builds of applications. It is highly recommended that developers enable the Debug SDK layer during development to catch Direct3D issues. The code in section 3.1 demonstrates how to enable the debug layer by setting the D3D11_CREATE_DEVICE_DEBUG flag.

DirectX Control Panel

Additional debug settings for the DirectX Debug Layer can be set via the Microsoft DirectX Control Panel. This tool can be used to force the debugger to break when certain DirectX errors, warnings, corruption, info, or messages are encountered. The

DirectX Control Panel is also available directly from Visual Studio 2012 under the *Debug * Graphics* menu.

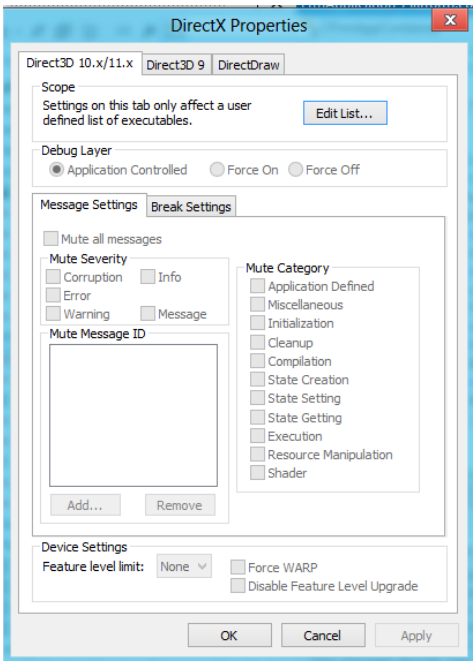


Figure 3-3 DirectX Control Panel

3.2 Qualcomm 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 DirectX, 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-a nd-resources.

Features

The Adreno SDK includes the following features:

- Emulation support in a desktop environment
- An SDK help system
- SDK browser

- Advaced samples for DirectX11 in Metro style
- A Visual Studio project template to create new samples

Refer to the Adreno SDK documentation for more information.

3.3 Qualcomm Adreno Profiler

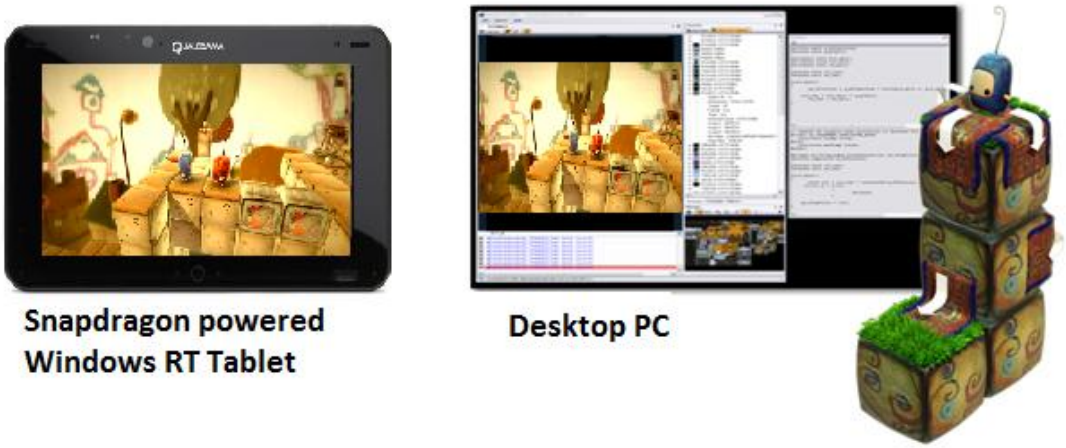


Figure 3-4 Adreno Profiler for DirectX Development

Overview

The Adreno Profiler is a leading PC-based tool used by 3D content developers to test, analyze, 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
- DirectX11 API call tracing and emulation
- Real-time driver overrides
- Shader analysis functionality for estimating source shader complexity

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.

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

4 Programming for the Platform

4.1 Direct3D 11.1 Feature Level 9_3 Overview

The Microsoft Direct3D 11.1 API introduces the notion of feature levels. A feature level defines a set of functionality that a GPU must minimally support. A range of different GPUs can be supported through Direct3D 11.1 via different feature levels. The Adreno exceeds the requirements of Direct3D 11.1 feature level 9_3. Application developers should target their applications to support feature level 9_3 in order to assure full compatibility with the Adreno GPU. The list of capabilities supported by each feature level in Direct3D 11.1 is provided by Microsoft at http://msdn.m icrosoft.com/en-us/library/windows/desktop/ff476876(v=vs.85).aspx.

The selection of a Direct3D feature level is done at device creation time by using *D3D11CreateDevice()*. When creating the device, the application passes an array of feature levels to attempt to create. In order to be compatible with the Adreno GPU, the application should request a *D3D_FEATURE_LEVEL_9_3* device (feature level 9_2 and 9_1 are supported as well). The following block of code demonstrates creating a *D3D11Device* for the Adreno GPU.

```
// Specify Direct3D feature level 9_3
D3D_FEATURE_LEVEL featureLevels[] =
{
    D3D_FEATURE_LEVEL_9_3
};

Microsoft::WRL::ComPtr<ID3D11Device> d3dDevice;
Microsoft::WRL::ComPtr<ID3D11DeviceContext> d3dDeviceContext;

UINT creationFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;
#if defined(_DEBUG)
creationFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif

if (FAILED(
            D3D11CreateDevice(
                nullptr,
                D3D_DRIVER_TYPE_HARDWARE,
                nullptr,
                creationFlags,
                featureLevels,
                ARRAYSIZE(featureLevels),
                D3D11_SDK_VERSION,
                &d3dDevice,
                nullptr,
                &d3dDeviceContext
                ) )
{
            // error..
}
```

4.1.1 WinRT and C++ Component Extensions (C++/CX)

Metro style or Modern UI style applications written to target the Adreno GPU with Direct3D 11.1 in Windows RT (Windows 8 for ARM) must be written against the WinRT API. WinRT is the new Windows Runtime in Windows RT as well as Windows 8. WinRT replaces the Win32 API from previous versions of Windows and forms the foundation that Windows RT (and Windows 8) Metro applications are built upon. WinRT is a native API and currently Microsoft does not expose Direct3D to managed (e.g., C#/VB) applications. While there are projects providing managed wrappers for Direct3D under Metro, Microsoft recommends that Direct3D Metro applications be written in C++.

In order to gain access to the WinRT API in C++, Microsoft introduced new language extensions called C++ Component Extensions (C++/CX). Reviewing the new language features is outside the scope of this document; however an excellent introduction to C++/CX and WinRT can be found at http://www.codeproject.com/Articles/262151/Visual-Cplusplus-and-WinRT-Metro-Some-fundamentals.

Apart from native C++ applications, Windows RT also lets developers build applications using a variety of programming languages and tools. One can program your applications using C#, C++, or Visual Basic, while using XAML to declaratively describe the user interface. Or you can build apps using web technologies like HTML5, CSS3, and JavaScript. More information on those can be found at: http://msdn.microsoft.com/en-US/windows/apps/br229512

But developers looking for the best possible performance on Windows 8 should use Microsoft DirectX 11.1 with C++.

4.1.2 Checking Feature Support

Direct3D 11 feature levels define a minimum set of functionality that a GPU must support along with several optional features. An application can check for optional functionality at runtime using the *ID3D11Device::CheckFeatureSupport()* method. For example, an application can determine whether the driver runs on top of a tile-based rendering GPU. The following snippet of code checks for a tile-based rendering.

```
D3D11_FEATURE_DATA_ARCHITECTURE_INFO info;
d3dDevice->CheckFeatureSupport(D3D11_FEATURE_ARCHITECTURE_INFO,
                               &info, sizeof(info));
if ( info.TileBasedDeferredRenderer )
{
    // TBR present
}
```

The Adreno graphics driver will set *TileBasedDeferredRenderer* to TRUE. Many other features can be queried at runtime using *ID3D11Device::CheckFeatureSupport()*. For the full list of available queries, please see http://msdn.microsoft.com/en-us/library/windows/desktop/ff476497(v=vs.85).aspx.

5 Optimizing for the Platform

5.1 Hardware Features to Consider

Adreno GPUs also support certain hardware features that are important to know before diving into the optimization guidelines. Below are some of those hardware features.

Unified Shader Model

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 GPU 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 5-1.

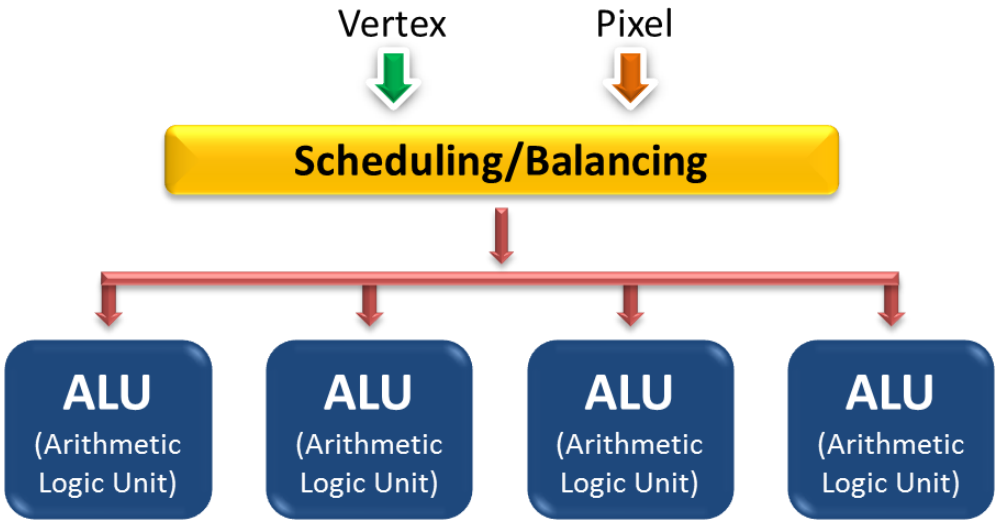


Figure 5-1 Unified Shader Architecture

Figure 5-1 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 5-2. This allows for greater flexibility of pixel/vertex load balances.

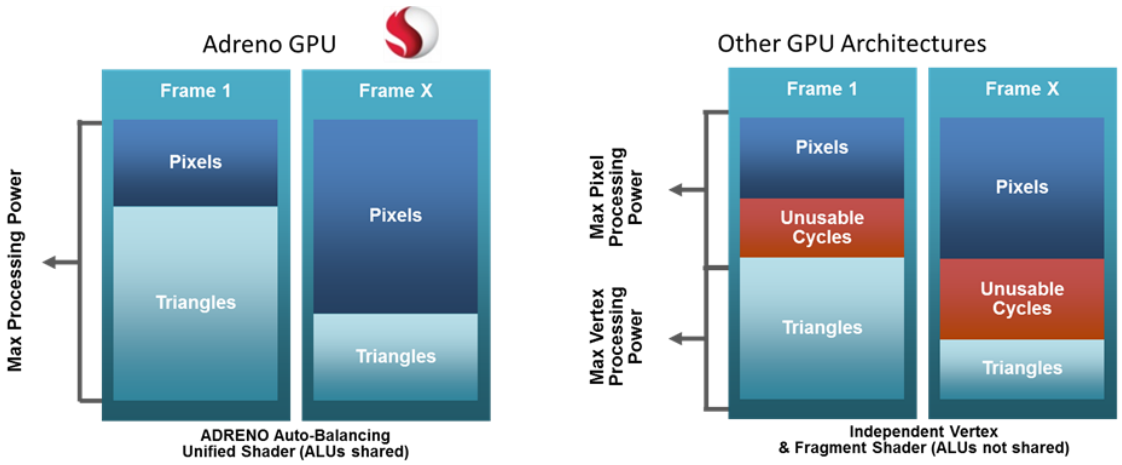


Figure 5-2 Flexibility in shader resources: Unified Shader Architecture

The Adreno 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.

Early Z Rejection

One of the important hardware features in Adreno GPUs, 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 5-3 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.

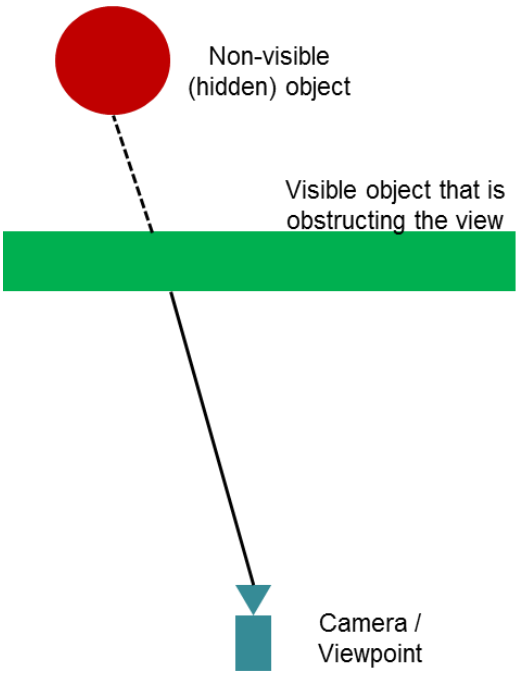


Figure 5-3 Early Z rejection: hidden object

Consider the example in Figure 5-4, 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 GPU can reject occluded pixels at up to four times the drawn pixel fill rate.

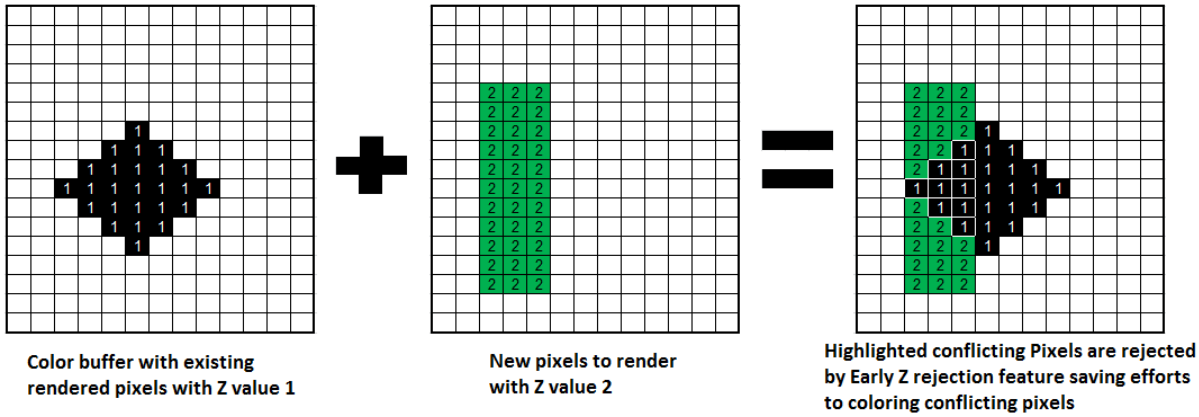


Figure 5-4 Early Z rejection

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.

Tile Based Rendering (TBR) Optimization

Another important architectural highlight of Adreno GPU is The tile-based rendering which is sometimes also referred to as binning. TBR is a mechanism that breaks the scene frame buffer into small regions for rendering, thereby optimizing the overall rendering. The tile-based 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 tile or bin is rendered to the GMEM. Then, each bin is resolved to the render surface in memory. The tile-based rendering mechanism is shown in further detail in Figure 5-5.

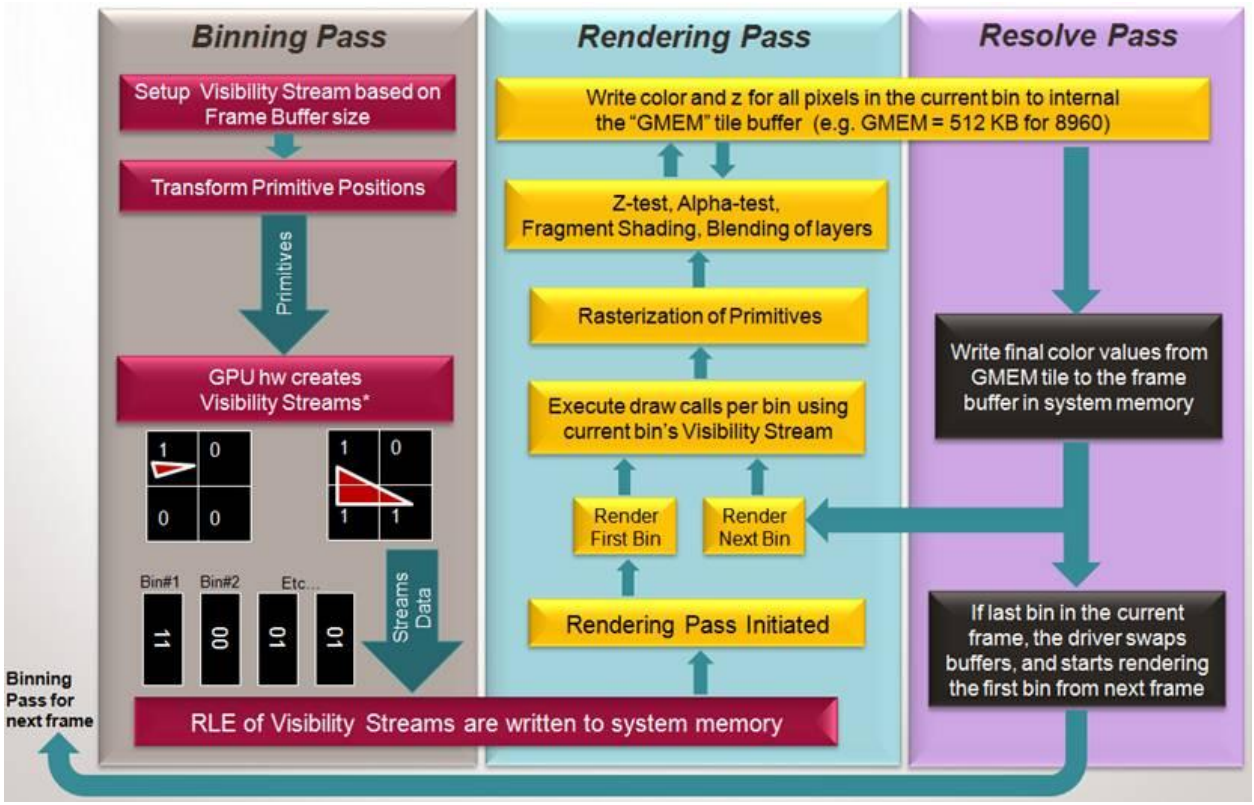


Figure 5-5 Tile based rendering or binning with Adreno GPUs

In the traditional desktop style rendering method, which is also referred to as direct rendering, the intermediate step of processing and rendering the scene per bin is eliminated. The geometry is directly rendered into the final frame buffer, which is in-system memory.

Tile-based rendering is very useful for cases with considerable depth complexity or overdraw with alpha blending. Tile-based rendering 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 Adreno GPU's tile-based rendering algorithm is apparent to the end developer, but there are certain considerations that you can make in your game design to take advantage of this.

The guidelines below include ways to take advantage of these hardware features.

5.2 Considerations for Tile-Based Rendering

On tile-based rendering architectures, it is important to minimize the loading and storing of data from GMEM. As described in previous sections, all data rendered into GMEM must be copied back out into system memory before it can be used further. The process of copying data from GMEM into system memory is called a GMEM store. In some cases, when the driver cannot determine if a render target will have every pixel (within the rendering scissor) overwritten with an opaque pixel, it must load the previous render target data into GMEM prior to rendering. The process of copying from system memory into GMEM is called a GMEM load. Copying data from one memory area to another is one of the most expensive operations, and should be minimized when possible.

A GMEM store is necessary when switching render targets, or prior to presenting the frame. Minimizing render target switches is key to achieving good performance on any tile-based renderer. GMEM loads are necessary only when the render target is bound, and not cleared or discarded prior to drawing to it. A flush is treated the same as binding a render target, and to avoid a GMEM load, must be cleared/discarded prior to drawing. A typical scenario that requires a GMEM load would be to copy from a render target into a new buffer for postprocessing effects. A typical scenario that does not require a GMEM load, but is often overlooked by application developers, is rendering to an offscreen render target, then switching to the swap chain and rendering the scene, without calling clear/discard on either render target.

There are a number of operations in Direct3D11.1 that can cause GMEM load. Some examples are listed below:

- *IDirect3D11DeviceContext::CopyResource* or *IDirect3D11DeviceContext::CopySubresourceRegion* where the source is a render target
- *IDirect3D11DeviceContext::CopyResource* or *IDirect3D11DeviceContext::CopySubresourceRegion* where the destination was already used as a texture for rendering to the current render target

- Binding a new render target using *IDirect3D11DeviceContext::OMSetRenderTarget* and drawing to it
- Presenting the framebuffer using *IDXGISwapChain1::Present*
- Explicitly calling flush using *IDirect3D11DeviceContext::Flush*

Application developers can take a number of important steps to avoid unnecessary resolves and get optimal performance out of the Adreno GPU. What follows is set of guidelines that should be followed to minimize unnecessary resolves.

Use *DiscardView* and/or *DiscardResource*

Direct3D11.1 provides two functions that are very important for achieving optimal performance out of a TBR: *IDirect3D11Context::DiscardView* and *IDirect3D11Context::DiscardResource*. These two functions inform the driver that the content in a resource view or resource is no longer needed. The driver can use this information to avoid restoring GMEM when rendering to the discarded target.

For example, an application might know that it will write an opaque value to each pixel of the framebuffer. The application therefore does not clear the contents of the screen before rendering a frame. While this behavior is correct, if *DiscardView* is not called, the driver will be forced to restore the contents of the render target before rendering (a heavyweight resolve). The driver otherwise has no way of knowing that the application intends to overwrite each pixel value. By calling *DiscardView*, the driver can avoid doing the heavyweight resolve because it knows the application is not depending on the contents being restored.

Minimize Render Target Switches

Every time a render target is switched, a performance penalty is incurred. Where possible, applications should issue all rendering to a single render target before switching to a new render target. Switching target at a minimum, causes a GMEM store…and if you are not clearing the contents between render target changes, causes a GMEM load.

Set Up the Scissor Rectangle Appropriately

If your application needs to switch frequently between render targets, you can see significant performance improvements by setting up the scissor rectangle to match the area being updated (using *ID3D11DeviceContext::RSSetScissorRects*). This allows the driver to track the areas that have been updated and minimize the traffic through GMEM to only the sections needed for the rendering. The driver will track scissors individually and apply the union of all scissors when it processes the scene. It is not recommended to change the scissor to tightly match the bounds of the object in each draw call. Applications should experiment to see how many scissors are necessary to eliminate unnecessary processing of unchanged pixels. As a rule of thumb, an application should not set more than 16 different scissors per render target.

Avoid Explicit Flushes

Avoid explicitly calling *IDirect3D11DeviceContext::Flush.* The exception to this is if no more rendering will be issued to the currently bound render target.

Defer Reading Render Target Data

Copying data from the current render target may cause a flush. If the data is not immediately needed, consider deferring the copy from the render target until all rendering commands for that render target have been issued. If the data is immediately needed, and the render target is not being changed (e.g., a snapshot of the render target is used as a texture for subsequent rendering commands), this can cause a flush and a GMEM load operation.

There are a few options available to minimize the cost of the GMEM load operation:

- If the render target data will be completely overwritten, call *DiscardResource*/*DiscardView* prior to calling any rendering commands.
- If the subsequent rendering does not cover the entire screen, set up the scissor rectangle to match the portion of the screen that will be updated.
- Use data from a previous render target.

Avoid Mapping Resources

CPU access to resources is not pipelined and thus caution should be exercised in performance-sensitive code. If a resource has been used to texture or blt into the current render target, then a lock is issued on that resource, and the current work must be flushed out before allowing the CPU to access the resource. The flush will cause a GMEM store, and if the render target continues to be used, it will cause a GMEM load prior to the rendering commands following the lock. If an application is mapping a resource for writing, a flush would be avoided by using D3D11_MAP_WRITE_DISCARD or D3D11_MAP_WRITE_NO_OVERWRITE

5.3 Adreno GPU Performance Recommendations

Aside from the tile-based rendering recommendations provided above, there are other important performance considerations when trying to optimally utilize the Adreno GPU. What follows in this section is a series of performance recommendations for Direct3D11.1 applications targeting Adreno GPUs.

5.3.1 Writing Efficient Shader Code

In order to take full advantage of Adreno shader resources, it is important to understand some parts of the architecure.

There are a few tips on how to make your shaders perform even better.

Shader compilation is expensive

Qualcomm’s optimizing compiler trades off CPU processing time during compilation for faster execution time when the application is using the shader. Shaders should be created only at non-critical times (e.g., application creation, loading screens, showing simple menus).

Use Intrinsics

Intrinsic functions as part of the HLSL language should be used whenever feasible, because you don't need to reinvent the wheel by writing your own function, and there is a good chance that they are optimized for specific shader profiles. Consult the HLSL Intrinsic Functions list for all the functions supported: http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx.

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 sometimes have a large impact on performance. For example, the following code should consume a single instruction slot.

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

Whereas the following code might consume 8 instruction slots because of the 1.0.

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

The variable a will be converted to float4 and then the addition will be done in floating point and finally the result will be converted back to the return type int4.

Pack Scalar Constants

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

```
float scale, bias;
float4 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).

```
float2 scaleNbias;
float4 a = Pos * scaleNbias.x + scaleNbias.y;
```

2x2 Pixel Processing

The scan converter "rough scans" each triangle into aligned 8x8 tiles as shown in Figure 5-6.

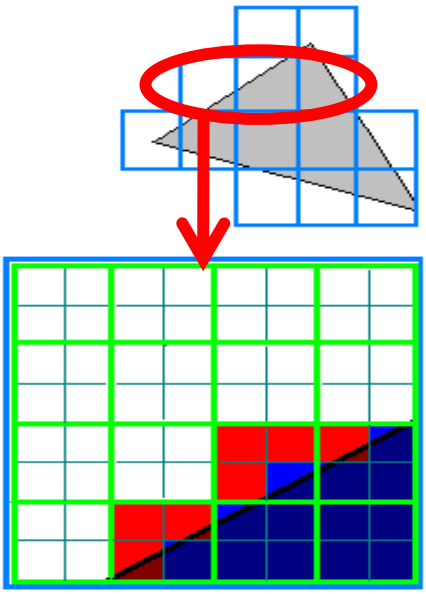


Figure 5-6 Scan converter "rough scans" each triangle

After this each tile is broken up into aligned 2x2 quads. Some of these quads might contain "dead" pixels as shown in red in

Figure 5-6. Very small triangles that are less than 2x2 pixels 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 a particle system, there could be reduced efficiency when it is implemented using smaller triangles.

Texture Sampling

The fact that the hardware works on 2x2 pixels at the same time offers a challenge on the pixel level. Four or more of those quads are grouped into a pixel vector (aka pixel thread). Optimizing meshes so that pixel vectors are nicely grouped together makes more efficient use of the texture cache.

Other strategies to avoid texture stalls are:

- Avoid random access
- Avoid 3D volume textures
- Avoid fetching from 7 textures in one pixel shader; a more appropriate number is 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 pixels that require anisotropic filtering that might end up being only a few pixels 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 1 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 look-ups do not incur any extra cost, while shader-specific gradients –based on ddx() / ddy()- 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 have a severe impact on memory and cache usage. Making effective use of texture memory is already a major task for real-time 3D content. Adding an additional 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 (D3D11_TEXTURE_ADDRESS_MIRROR) modes for symmetrical textures like volumetric light maps. The mirror once (D3D11_TEXTURE_ADDRESS_MIRROR_ONCE) 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.

Branching

Static branching performs well, but at the risk of extra GPRs. Using dynamic branching in a shader has a certain, non-constant overhead that depends on the exact shader code. Therefore, using dynamic branching is not always a performance win. Since multiple pixels are processed as one, in the event that some pixels take a branch while others do not, the GPU must execute both paths (all instructions really do execute, but there's a masking bit that controls the output so that only appropriate pixels are affected). In this case, the performance will appear worse than if each pixel was processed individually. If all pixels take the same path, the GPU is capable of really taking the branch, which is good for performance.

Pack Shader Interpolators

Shader-interpolated values (values passed between the vertex and pixel shader) require a GPR to hold data being fed into a pixel shader. Therefore, you should minimize their use. Use constants where a value is uniform. All interpolaters have four components, whether you use them or not, so pack values together. Putting two float2 texture coordinates into a single float4 value is a common practice, but other strategies employ more creative packing.

Level-of-Detail Shaders

To improve the efficiency of shaders, 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 further the object is from the camera, the less expensive 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. 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

Minimize Shader GPRs

Minimizing GPRs can be the most important performance optimization method. Inputting simpler shaders to the compiler helps guarantee optimal results. Sometimes, modifying HLSL 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 simultaneously the multiple texture coordinates and fetched results.)

Avoid Math on Shader Constants

Almost every shipped game since the advent of shaders has spent instructions performing needless 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 (clip) in the Pixel Shader

Some developers believe that manually killing pixels (*clip()* in HLSL) in the pixel shader boosts performance. The rules are not simple for two reasons:

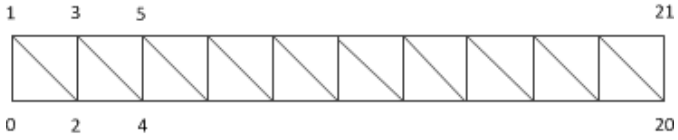
- If some pixels in a thread are killed, and others are not, the shader still executes.
- If clip is used, Early-Z must be disabled by the driver. The reason for this is because if Early-Z is enabled and a pixel is killed in the pixel shader, the depth buffer value would be incorrectly updated for that pixel. The driver must therefore disable Early-Z in the case when a shader uses the clip() function.

It is recommended to only use the *clip()* instruction in the pixel shader when it is absolutely necessary to achieve an effect like alpha test. Otherwise, do not use *clip()* as a performance optimization because it may result in reduced performance.

5.3.2 Rendering and Geometry

Vertex reuse

The Adreno GPU has a post-transform vertex shader cache that stores limited number of vertices. This means that after a vertex shader is executed, the post-transform result of executing the vertex shader is stored in the cache. If the same vertex is referenced again in the index buffer before it is evicted from the cache, vertex shader execution can be skipped. The behavior of the post-transform vertex cache is illustrated by example in Figure 5-7.



	Indexed rendering		Non-indexed rendering
TriList	VS Processed	VS in cache (reused)	VS processed
0,1,2	0,1,2	none	0,1,2
1,2,3	3	0-2	1,2,3
2,3,4	4	0-3	2,3,4
3,4,5	5	0-4	3,4,5
4,5,6	6	0-5	4,5,6
5,6,7	7	0-6	5,6,7
6,7,8	8	0-7	6,7,8
7,8,9	9	0-8	7,8,9
8,9,10	10	0-9	8,9,10
9,10,11	11	0-10	9,10,11
10,11,12	12	0-11	10,11,12
11,12,13	13	0-12	11,12,13
12,13,14	14	0-13	12,13,14
13,14,15	15	0-14	13,14,15
14,15,16	16	1-15	14,15,16
15,16,17	17	2-16	15,16,17
16,17,18	18	3-17	16,17,18
17,18,19	19	4-18	17,18,19
18,19,20	20	5-19	18,19,20
19,20,21	21	6-20	19,20,21

Figure 5-7 Benefits of indexed rendering

Applications should always use indexed rendering and create index buffers for holding the indices. If an application chooses not to use indexed rendering, it will not benefit from the vertex reuse described in Figure 5-2. The best way to achieve optimal reuse of the post-transform vertex cache is to order the index buffers using either triangle strips or strip-ordered indexed triangle lists (both should achieve roughly equivalent performance).

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 tile-based rendering algorithm obeys the D3D cull mode, so back face culled polygons can improve vertex processing performance as well.

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.

Avoid Excessive Clipping

Avoid excessive clipping, especially for a static geometry, by further dividing your geometry into more primitives. For example, in Figure 5-8, 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.

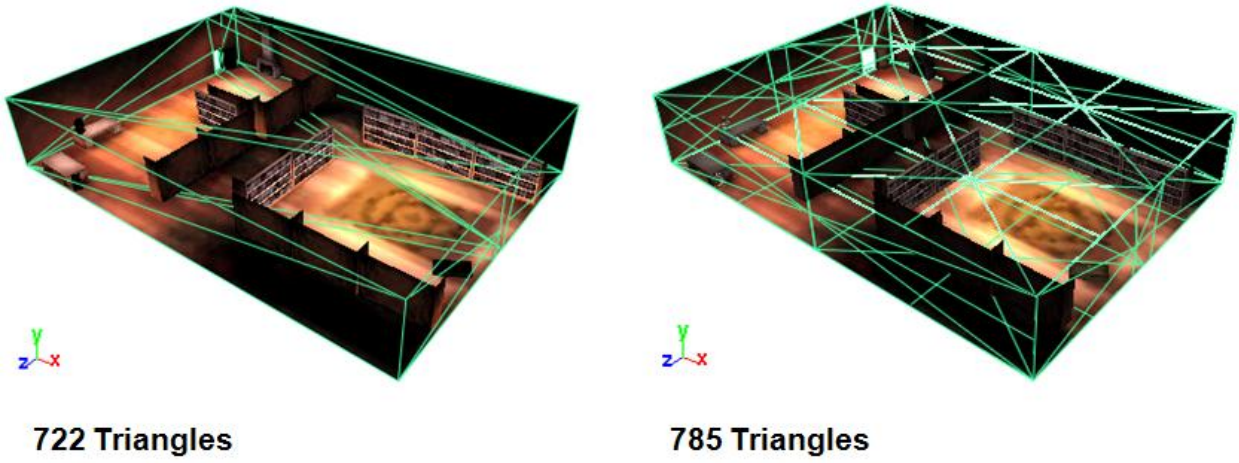


Figure 5-8 Geometry design to avoid clipping

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. Tile based 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.

Draw Skybox Last

Many games are still rendering the skybox first. This used to be 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 for this is that the skybox itself tends to be occluded by large parts of the scene. If you render the skybox first you're shading a lot of pixels that will be overwritten later on. This wastes pixel processing and bandwidth. By rendering the skybox in 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 easily be accomplished either by copying 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. Alternatively, the W value can be copied into Z in the vertex shader.

Prefer Lazy Constant updates

For Direct3D11.1 Level 9 applications, the D3D runtime will batch up all constant updates and broadcast them to the driver on demand. Applications should consider lazy updates of constant state in order to minimize the amount of work done by the Direct3D runtime.

Minimize State Changes

For tile-based rendering architectures, every state update is replayed in each bin. Applications should be careful to set only whatever state is absolutely necessary for a given draw operation. That being said, make sure the state represents how you intend to render the scene. For example, do not set or leave the blend enable state to TRUE if the rendering is not relying on blending for the final image. The driver enables optimizations based on states, and leaving the state incorrectly set could have a negative effect on performance.

Warmup Phase for Shader Combinations

The Adreno GPU requires the driver to merge all shaders together prior to submitting them to the GPU. At draw time, if there is a unique combination of layout state, vertex shader and pixel shader, the driver will link all of the shaders and cache the resulting shader in memory. If possible, it is beneficial for an application to draw dummy primitives with the expected combination of layout state, vertex and pixel shaders to allow the driver to pre-cache the combinations during a loading screen.

Occlusion Queries

When in tile-based rendering mode, occlusion queries are batched along with the rendering for a given render target. Since rendering for a given render target is not processed until a flush condition is encountered, it is possible that the GPU hasn't processed the query by the time Present is called, even if ID3D11DeviceContext::Begin and ID3D11DeviceContext::End were issued near the start of the frame. The most efficient way to use occlusion queries is to issue the query on a previous frame, when possible. Applications can call Flush or GetData to give the driver a hint that the application needs the pending query, but this may incur an additional GMEM load and GMEM store (see section 5.2).

If the occlusion data must be used in the same frame as the query, if Begin and End are issued for rendering on a given render target, it is best to wait until issuing rendering commands on another render target before calling GetData. This way the natural flow of the driver will process the render target, and the query, and there will not be any additional GMEM loads or GMEM stores.

Never spin in a tight loop to wait on a query to be satisfied. While this will work functionally, it does not allow the application, or driver, to continue to feed commands to the GPU, and will affect performance. Applications should have a fallback path to take in case a query is not ready when GetData is called.

Updating Data

The ideal way to modify vertex and/or index buffer data is for an application to manage its own buffers. The following application pseudo code will run optimally in the driver.

1. Create a buffer of a known size, preferably large enough to hold at least one frame of data.
2. Keep a count of how much data has been added to that buffer,
3. If there is space in the buffer for your data, map the buffer using D3D11_MAP_WRITE_NO_OVERWRITE to get a pointer to the surface. Write the data to that location and increment the data count.
4. If there is not enough space in the buffer for your data, map the buffer using D3D11_MAP_WRITE_DISCARD and reset the data count to zero.
5. Copy the new data into the mapped buffer, then unmap it.

The path described above is fairly commonly used by applications, and is optimized in the driver.

5.3.3 Textures

Compress Textures

Texture cache friendliness requires that textures be compressed whenever possible. Note that render targets are not compressed. While CPU time could be spent to compress resolved render targets, the CPU-GPU synchronization and the actual compression time would make real-time rendering impossible. Use external tools like Adreno Texture Converter to compress textures and normal maps.

Use Mipmaps

Mipmaps have a two-fold benefit when lower LOD mips can be used. When they are fetched from system memory, they are smaller and consume less bandwidth. Also, since lower LOD mips are significantly smaller, they are more likely to reside in the texture cache, which eliminates any need to fetch data over the system memory bus.

If an application decides to use GenerateMips, it should be done when loading a scene to avoid any potential hitches during the rendering the end user sees.

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.

Resource bind flags

Set only the necessary bind flags for a given resource. For example, do not mark a read-only texture as a render target. The driver uses these flags to optimize several things, and with incorrect information, an application will be taking a less-desirable path through the driver and GPU.

Texture updates

When running a Direct3D11.1 Level 9 application, applications should consider avoiding UpdateSubresource calls. The Direct3D runtime will create a temporary resource and copy the system memory data into it prior to calling the driver to blt the data. For more direct control over how the update is handled, applications should create and manage

their own staging resource, and use CopyResource/CopySubresourceRegion for updating textures.

6 Migrating from OpenGL ES to Direct3D11.1

This chapter is a guide for developers migrating applications from OpenGL ES 2.0/3.0 to Direct3D11.1 feature level 9_3. While providing similar functionality to OpenGL ES, Direct3D11.1 is a substantially different API. The following sections detail how various parts of the OpenGL ES API map to Direct3D11.1.

6.1 Feature Comparison of OpenGL ES 2.0/3.0 and Direct3D11.1 Feature Level 9_3

The Adreno GPUs support OpenGL ES 2.0, OpenGL ES 3.0, and Direct3D11.1 Feature Level 9_3. The following table lists the features exposed in each API for the Adreno. For features that are not part of core OpenGL ES 2.0/3.0, the extension through which the feature is supported is listed. This table provides the developer a high-level guide as to whether features of their OpenGL ES 2.0 or OpenGL ES 3.0 application are possible in Direct3D11.1 Feature Level 9_3.

Feature	**OpenGL ES 2.0**	**OpenGL ES 3.0**	**Direct3D11.1 Feature Level 9_3**
Triangles, Triangle Strips, Triangle Fans, Lines, Points	X	X	X
Line Width	X	X	
32-bit Vertex Indices	GL_OES_element_index_uint	X	X
Point Sprites	X	X	
Polygon Offset (Depth Bias)	X	X	X
Vertex Textures	Optional (supported on Adreno 3xx)	X	
Alpha To Coverage	X	X	
Non-Power-of-2 Textures (no mipmapping, limited wrap modes)	X	X	X
3D Textures	GL_OES_texture_3D	X	X
Multisample Anti-Aliasing (MSAA)	Optional (supported on Adreno 3xx)	X	X
Floating Point 16-bit Textures	GL_OES_texture_float	X	X
Floating Point 32-bit Textures	GL_OES_texture_half_float	X	X
BC1/BC2/BC3 Texture Compression			X

Feature	OpenGL ES 2.0	OpenGL ES 3.0	Direct3D11.1 Feature Level 9_3
ATC Texture Compression	GL_AMD_compressed_ATC_texture	GL_AMD_compressed_ATC_texture	
ETC Texture Compression	GL_OES_compressed_ETC1_RGB8_ texture	X	
Depth Textures	GL_OES_depth_texture	X	
Anisotropic Texture Filtering	GL_EXT_texture_filter_anisotropic	GL_EXT_texture_filter_anisotropic	X
Shader Derivatives	GL_OES_standard_derivatives	X	X
2D Texture Arrays		X	
Multiple Render Targets		X	X
Occlusion Queries		X	X
10/10/10/2 Vertex Data	GL_OES_vertex_type_10_10_10_2	X	
16-bit Floating Point Vertex Data	GL_OES_vertex_half_float	X	X
10/10/10/2 Textures	GL_EXT_texture_type_2_10_10_10_ REV	X	
11/11/10 Floating Point Textures		X	
Geometry Instancing		X	X
sRGB Textures and Render Targets		X	X
Transform Feedback		X	
Shader Bitwise Integer operations		X	
RGB5E9 shared-exponent textures		X	

6.2 Mapping OpenGL ES Code to Direct3D11

Developers porting applications from OpenGL ES to Direct3D11 will encounter a large number of API differences between OpenGL ES and Direct3D11. This section provides a guide for porting OpenGL ES functionality to Direct3D11.

6.2.1 Vertex and Index Buffer Objects

Most OpenGL ES applications have static vertex data stored in vertex buffer objects (VBOs). Some OpenGL ES applications also rely on client-side vertex arrays to transfer data from the CPU to GPU at draw time. There is no equivalent of client-side vertex arrays in Direct3D11; all vertex data must be stored in buffers (*ID3D11Buffer*).

The following OpenGL ES code shows a typical example of creating a vertex buffer object and loading it with data:

```
GLuint hBufferHandle;
glGenBuffers( 1, &hBufferHandle );
```

```
glBindBuffer( GL_ARRAY_BUFFER, hBufferHandle );
glBufferData( GL_ARRAY_BUFFER, nNumVertices*nVertexSize, pSrcVertices,
              GL_STATIC_DRAW );
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

An *ID3D11Buffer* used for string the same vertex data could be created with the following code in Direct3D11:

```
D3D11_BUFFER_DESC bdesc = CD3D11_BUFFER_DESC( nVertexSize * nNumVertices,
                                              D3D11_BIND_VERTEX_BUFFER );
D3D11_SUBRESOURCE_DATA vertexBufferData;
vertexBufferData.pSysMem = pSrcVertices;
vertexBufferData.SysMemPitch = 0;
vvertexBufferData.SysMemSlicePitch = 0;

ComPtr<ID3D11Buffer> buffer;
if ( FAILED( pD3DDevice->CreateBuffer( &bdesc, &vertexBufferData, &buffer) ) )
{
        return FALSE;
}
```

A similar mapping of code exists between creating an OpenGL ES index buffer object (IBO) and an *ID3D11Buffer* in Direct3D11 for storing indices. In OpenGL ES, VBOs are bound for rendering using *glBindBuffer()* and then bound to vertex attributes using *glBindVertexAttribPointer()* and *glEnableVertexAttribArray()*. In Direct3D11, vertex and index buffers are bound to the Input Assembler (IA) using code such as the following:

```
UINT32 offset = 0;
pD3DDeviceContext->IASetVertexBuffers(
            0,                                  // StartSlot
            1,                                  // NumBuffers
            pVertexBuffer.GetAddressOf(),       // VertexBuffers
            &m_nVertexSize,                     // Strides
            &offset);                           // Offsets

DXGI_FORMAT format = (m_nIndexSize == 2) ? DXGI_FORMAT_R16_UINT :
                                           DXGI_FORMAT_R32_UINT;
pD3DDeviceContext->IASetIndexBuffer(
            pIndexBuffer.Get(),     // IndexBuffer
            format,                 // Format
            0);                     // Offset
```

One big difference between OpenGL ES and Direct3D11 is the way in which vertex data gets bound as inputs to the vertex shader. In Direct3D11, an *ID3D11InputLayout* object must be created with a reference to the vertex shader byte code. The *ID3D11InputLayout* object defines how the vertex data is laid out in memory and also how it binds to the HLSL semantics provided in the vertex shader.

For example, let's say that there is a vertex shader that declares the following set of inputs:

```
struct VertexShaderInput
{
    float2 vVertexPos : POSITION;
    float4 vVertexColor: COLOR0;
    float2 vVertexTex : TEXCOORD0;
};
```

The layout is described using the following code:

```
D3D11_INPUT_ELEMENT_DESC inputDesc[] =
{
  {"POSITION", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA,
    0},

  {"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,
    D3D11_INPUT_PER_VERTEX_DATA, 0},

  {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT),
    D3D11_INPUT_PER_VERTEX_DATA, 0},
};
```

The *ID3D11InputLayout* object can then be created using the D3D11_INPUT_ELEMENT_DESC array and the vertex shader byte code as follows:

```
ComPtr<ID3D11InputLayout> inputLayout;
result = !FAILED(
            pD3DDevice->CreateInputLayout(
                inputDesc,
                3,
                vertexShaderByteCode,
                vertexShaderByteCodeSize,
                &inputLayout) );
```

At render time, the vertex layout must be set to the input assembler as follows:

```
  pD3DDeviceContext->IASetInputLayout(inputLayout.Get());
```

6.2.2 Textures

In OpenGL ES 2.0, texture and sampler state were bound together in a single object called a texture object. OpenGL ES 3.0 introduces sampler objects that separate sampler state from texture data. In Direct3D11.1, much like OpenGL ES 3.0, textures and sampler state are separate. Additionally, in Direct3D11.1 an *ID3D11ShaderResourceView* is needed to be able to access a texture in a shader. The Direct3D11.1 separation of texture data, sampler state, and shader resource view allow for increased flexibility (especially at higher feature levels), but also require more setup code than OpenGL ES.

The following example of code creates a texture object and loads it with data (a single mip level) and sampler state in OpenGL ES 2.0.

```
glGenTextures( 1, (GLuint*)pTexId );
glBindTexture( GL_TEXTURE_2D, *pTexId );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,     GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,     GL_REPEAT );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, nMipWidth, nMipHeight,
                              nBorder, GL_RGBA, GL_UNSIGNED_BYTE,
                              pInitialData );
```

In order to perform the equivalent in Direct3D11, we must create an *ID3D11Texture2D, ID3D11SamplerState,* and *ID3D11ShaderResourceView* as shown below.

```
D3D11_TEXTURE2D_DESC textureDesc = {0};
textureDesc.Width = nWidth;
textureDesc.Height = nHeight;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.Usage = D3D11_USAGE_DEFAULT;
textureDesc.CPUAccessFlags = 0;
textureDesc.MiscFlags = 0;
textureDesc.MipLevels = 1;
textureDesc.ArraySize = 1;
textureDesc.SampleDesc.Count = 1;
textureDesc.SampleDesc.Quality = 0;
textureDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;

D3D11_SUBRESOURCE_DATA textureSubresourceData[1];
ZeroMemory(&textureSubresourceData[0], sizeof(D3D11_SUBRESOURCE_DATA));
textureSubresourceData[0].pSysMem = pInitialData;
textureSubresourceData[0].SysMemPitch = nMipWidth * nBPP / 8;
textureSubresourceData[0].SysMemSlicePitch = 0;

ComPtr<ID3D11Texture2D> texture;
ComPtr<ID3D11ShaderResourceView> textureView;
ComPtr<ID3D11SamplerState> sampler;

if ( FAILED(
        pD3DDevice->CreateTexture2D(
            &textureDesc,
            &textureSubresourceData[0],
            &texture ) ) )
{
    return FALSE;
}

// Create the shader resource view for the texture
D3D11_SHADER_RESOURCE_VIEW_DESC textureViewDesc;
ZeroMemory(&textureViewDesc, sizeof(textureViewDesc));
textureViewDesc.Format = textureDesc.Format;
textureViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
textureViewDesc.Texture2D.MipLevels = textureDesc.MipLevels;
textureViewDesc.Texture2D.MostDetailedMip = 0;

if ( FAILED( pD3DDevice->CreateShaderResourceView(
                texture.Get(),
                &textureViewDesc,
                &textureView
                ) ) )
{
   return FALSE;
}
// Create a sampler
D3D11_SAMPLER_DESC samplerDesc;
ZeroMemory(&samplerDesc, sizeof(samplerDesc));
samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
samplerDesc.MaxAnisotropy = 0;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;
```

```
samplerDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
samplerDesc.BorderColor[0] = 0.0f;
samplerDesc.BorderColor[1] = 0.0f;
samplerDesc.BorderColor[2] = 0.0f;
samplerDesc.BorderColor[3] = 0.0f;

if ( FAILED( pD3DDevice->CreateSamplerState( &samplerDesc, &sampler ) ) )
{
    return FALSE;
}
```

While it is clearly significantly more code to create a texture in Direct3D11.1 than OpenGL ES 2.0, the separation of sampler state from texture data can provide a simplification and performance advantage. Rather than having to create a sampler for each texture, an application can create a sampler state object and share it for any textures that will be fetched using equivalent sampler settings in the shader.

Binding the sampler and shader resource view to the pixel shader is straightforward and can be accomplished using code such as shown below.

```
VOID Bind( UINT32 nTextureUnit )
{
    pD3DDeviceContext->PSSetShaderResources(
            nTextureUnit,
            1,
            m_pShaderResourceView.GetAddressOf());

    pD3DDeviceContext->PSSetSamplers(
            nTextureUnit,
            1,
            m_pSamplerState.GetAddressOf());
}
```

6.2.3 Render State

In OpenGL ES, most render state is specified one function call at a time using routines such as *glEnable(), glBlendFunc(), glDepthFunc(), etc.* The mapping of render state to a single function provides a significant ease-of-use advantage to the developer: render state can be specified as known and as needed. However, a common performance problem in OpenGL ES applications is redundant state setting. Further, there is function overhead associated with calling a routine for each individual render state.

The designers of Direct3D11 decided to make the API more performance-friendly in order to reduce the overhead and frequency of state changes. In Direct3D11, all render state is set via *render state blocks*. Render state is grouped together into blocks of state that are commonly set together for a particular pipeline stage. While this state model comes at a performance advantage, it is also a bit less convenient for developers used to using the OpenGL state model.

For example, in OpenGL ES, one can set up the depth test, backface culling, and blending with the following simple function calls:

```
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );
glEnable( GL_DEPTH_TEST );
glDepthFunc( GL_LEQUAL );
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Setting the same set of render state in Direct3D11 requires creating several render state objects and setting them to be current as shown next.

```
ComPtr<ID3D11RasterizerState> rasterizerState;
D3D11_RASTERIZER_DESC rdesc = CD3D11_RASTERIZER_DESC(D3D11_DEFAULT);
rdesc.CullMode = D3D11_CULL_BACK;
rdesc.FrontCounterClockwise = TRUE;
pD3DDevice->CreateRasterizerState(&rdesc, &rasterizerState);

ComPtr<ID3D11DepthStencilState> depthStencilState;
D3D11_DEPTH_STENCIL_DESC dsdesc = CD3D11_DEPTH_STENCIL_DESC(D3D11_DEFAULT);
dsdesc.DepthFunc = D3D11_COMPARISON_LESS_EQUAL;
dsdesc.DepthEnable = TRUE;
pD3DDevice->CreateDepthStencilState(&dsdesc, &depthStencilState);

ComPtr<ID3D11BlendState> blendState;
D3D11_BLEND_DESC bdesc = CD3D11_BLEND_DESC(D3D11_DEFAULT);
bdesc.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
bdesc.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
bdesc.RenderTarget[0].DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
bdesc.RenderTarget[0].BlendEnable = TRUE;
pD3DDevice->CreateBlendState(&bdesc, &blendState);

// Set the render states
pD3DDeviceContext->RSSetState( rasterizerState.Get() );
pD3DDeviceContext-> OMSetDepthStencilState ( depthStencilState.Get() );
float blendFactor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
UINT32 sampleMask = 0xffffffff;
pD3DDeviceContext->OMSetBlendState(blendState.Get(), blendFactor, sampleMask);
```

It is important to understand that in order to make the most efficient use of render state objects in Direct3D11, the application should minimize the number of times the objects are set. Further, the render state objects should ideally be created at load/startup time rather than dynamically. At runtime, the application should only be setting render state objects current rather than creating new ones. This is not always possible, but ideally this method should be employed.

6.2.4 Uniforms/Constants

In OpenGL ES 2.0, all shader constants are declared as uniform variables in GLSL and set with values using the *glUniform*()* APIs. Loading uniform data is a frequent bottleneck in OpenGL ES 2.0 applications, especially when dealing with large volumes of data such as in matrix palette skinning. Much like with the render state API, the Direct3D11 designers took an approach that aimed to achieve efficiency. All constant data loaded to shaders in Direct3D11 is done through the user of constant buffers. A constant buffer is basically a grouping of constants that will be updated at similar frequency. For example, the following shows a constant buffer declaration in HLSL:

```
cbuffer BumpedReflectionConstantBuffer : register(c0)
{
    float4x4 MatModelViewProj;
    float4x4 MatModelView;
    float4   LightPos;
    float4   EyePos;
    float    FresnelPower;
    float    SpecularPower;
};
```

A constant buffer is loaded with data using an *ID3D11Buffer*. Typically, an application will declare a C/C++ structure representing the same data that is stored in the constant buffer in the shader. For example, the following C structure represents the constant buffer data above.

```
struct BumpedReflectionConstantBuffer
{
    FRMMATRIX4X4 MatModelViewProj;
    FRMMATRIX4X4 MatModelView;
    FRMVECTOR4   LightPos;
    FRMVECTOR4   EyePos;
    FLOAT32      FresnelPower;
    FLOAT32      SpecularPower;
    FRMVECTOR2   Padding; // Pad to multiple of 16-bytes
};
```

The constant buffer is created using the following block code:

```
D3D11_SUBRESOURCE_DATA constantBufferData;
constantBufferData.pSysMem = pSrcConstants;
constantBufferData.SysMemPitch = 0;
constantBufferData.SysMemSlicePitch = 0;

ComPtr<ID3D11Buffer> buffer;
if ( FAILED(
        pD3DDevice->CreateBuffer(
            &CD3D11_BUFFER_DESC(nBufferSize, D3D11_BIND_CONSTANT_BUFFER),
            &constantBufferData,
            &buffer) ) )
{
    return FALSE;
}
```

The constant buffer can be updated with data using *ID3D11DeviceContext::UpdateSubresource()* when the constant data changes. Finally, the constant buffer can be bound to the vertex (and/or pixel shader) as follows:

```
pD3DDeviceContext->VSSetConstantBuffers( bufferIndex,1,
                m_pBuffer.GetAddressOf());

pD3DDeviceContext->PSSetConstantBuffers( bufferIndex, 1,
                m_pBuffer.GetAddressOf());
```

6.2.5 Framebuffer Objects

In OpenGL ES, render targets are created using Framebuffer Objects (FBOs). FBOs have attachment points for color buffer(s) (more than one if using multiple render targets) and a depth/stencil buffer (depth and stencil can be separate or combined in an interleaved buffer). The following block of code demonstrates the creation of an FBO with a color attachment that can be used as a texture and depth/stencil attachment that can be fetched from as a texture.

```
glGenTextures( 1, &m_hTexture );
glBindTexture( GL_TEXTURE_2D, m_hTexture );
glTexImage2D( GL_TEXTURE_2D, 0, nInternalFormat, nWidth, nHeight, 0, nFormat,
              nType, NULL );
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                    m_hTexture, 0 );

glGenTextures( 1, &m_hDepthTexture);
```

```
glBindTexture( GL_TEXTURE_2D, m_hDepthTexture);
glTexImage2D( GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, nWidth, nHeight, 0,
              GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL );
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
                        m_hDepthTexture, 0);
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_STENCIL_ATTACHMENT, GL_TEXTURE_2D,
                        m_hDepthTexture, 0);
```

To create a render target that can be used as a texture in Direct3D11, we need to create a texture with both a shader resource view and a render target view. The following block of code demonstrates creating a render target that has a texture color buffer and a depth/stencil buffer.

```
D3D11_TEXTURE2D_DESC textureDesc;
ZeroMemory( &textureDesc, sizeof(textureDesc) );
textureDesc.Width = nWidth;
textureDesc.Height = nHeight;
textureDesc.MipLevels = 1;
textureDesc.ArraySize = 1;
textureDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
textureDesc.SampleDesc.Count = 1;
textureDesc.Usage = D3D11_USAGE_DEFAULT;
textureDesc.BindFlags = D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE;
textureDesc.CPUAccessFlags = 0;
textureDesc.MiscFlags = 0;
// Render target view
D3D11_RENDER_TARGET_VIEW_DESC renderTargetViewDesc;
ZeroMemory(&renderTargetViewDesc, sizeof(renderTargetViewDesc));
renderTargetViewDesc.Format = textureDesc.Format;
renderTargetViewDesc.ViewDimension = D3D11_RTV_DIMENSION_TEXTURE2D;
renderTargetViewDesc.Texture2D.MipSlice = 0;

// Shader resource view
D3D11_SHADER_RESOURCE_VIEW_DESC shaderResourceViewDesc;
ZeroMemory(&shaderResourceViewDesc, sizeof(shaderResourceViewDesc));
shaderResourceViewDesc.Format = textureDesc.Format;
shaderResourceViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
shaderResourceViewDesc.Texture2D.MipLevels = 1;
shaderResourceViewDesc.Texture2D.MostDetailedMip = 0;

// Sampler description
D3D11_SAMPLER_DESC samplerDesc;
ZeroMemory(&samplerDesc, sizeof(samplerDesc));
samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_POINT;
samplerDesc.MaxAnisotropy = 0;
samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_CLAMP;
samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_CLAMP;
samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_CLAMP;
samplerDesc.MipLODBias = 0.0f;
samplerDesc.MinLOD = 0;
samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;
samplerDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
samplerDesc.BorderColor[0] = 0.0f;
samplerDesc.BorderColor[1] = 0.0f;
samplerDesc.BorderColor[2] = 0.0f;
samplerDesc.BorderColor[3] = 0.0f;

// Create Texture
ComPtr<ID3D11Texture2D> pTexture;
if ( FAILED(pD3DDevice->CreateTexture2D( &textureDesc, NULL, &pTexture) ) )
{
    return FALSE;
}
```

```
// Create Render Target View
ComPtr<ID3D11RenderTargetView> pTextureRenderTargetView;
if ( FAILED(pD3DDevice->CreateRenderTargetView( pTexture.Get(),
                                                &renderTargetViewDesc,
                                                &pTextureRenderTargetView ) ) )
{
    return FALSE;
}

// Create Shader Resource View
ComPtr<ID3D11ShaderResourceView> pTextureShaderResourceView;
if ( FAILED(pD3DDevice->CreateShaderResourceView( pTexture.Get(),
                                                  &shaderResourceViewDesc,
                                                  &pTextureShaderResourceView
                                                  ) ) )
{
    return FALSE;
}

// Create Sampler State
ComPtr<ID3D11SamplerState> pTextureSampler;
if ( FAILED(pD3DDevice->CreateSamplerState( &samplerDesc, &pTextureSampler ) )
)
{
    return FALSE;
}
// Create a depth/stencil texture for the depth attachment
ComPtr<ID3D11Texture2D> pDepthTexture;
textureDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
textureDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
if ( FAILED(pD3DDevice->CreateTexture2D( &textureDesc, NULL, &pDepthTexture )))
    return FALSE;

ComPtr<ID3D11DepthStencilView> pDepthStencilView;
D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc;
ZeroMemory(&depthStencilViewDesc, sizeof(depthStencilViewDesc));
depthStencilViewDesc.Format = textureDesc.Format;
depthStencilViewDesc.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
depthStencilViewDesc.Texture2D.MipSlice = 0;
if ( FAILED(pD3DDevice ->CreateDepthStencilView( pDepthTexture.Get(),
                                                 &depthStencilViewDesc,
                                                 &pDepthStencilView ) ) )
    return FALSE;
```

The *ID3D11ShaderResourceView* can be used to bind the texture to a pixel shader. To *render to the render target, the ID3D11RenderTargetView and ID3D11DepthStencilView are used as* follows:

```
ID3D11RenderTargetView* pRTViews[1] = { pTexture0RenderTargetView.Get() };
pD3DDeviceContext->OMSetRenderTargets( 1, pRTViews, pDepthStencilView.Get() );
```

Depth Textures

A common use of render targets is to render to a depth texture for an effect such as shadow mapping. The Adreno GPU supports depth textures and this functionality is exposed in OpenGL ES 2.0 via the GL_OES_depth_texture and is part of the OpenGL ES 3.0 core functionality. However, Microsoft chose not to expose depth textures in Direct3D11.1 feature level 9_3. As a consequence, the easiest way to achieve the equivalent effect is to render to a single-channel 32-bit color texture

(DXGI_FORMAT_R32_FLOAT) and write a shader that computes the depth value and writes it to the color buffer. In order to get the best use of precision, it is recommended to write the value 1.0 – (z/w) to each pixel. The following block of HLSL code writes the depth value to an interpolator that is sent to the pixel shader:

```
vso.Position = mul(MatModelViewProj,  Position);
// Store depth for pixel shader
// Depth is Z/W.
// Use 1 - z/w to get better precision
vso.Depth = (1.0 - vso.Position.z / vso.Position.w);
```

Several examples in the DirectX Adreno SDK make use of this functionality including the DepthOfField, ShadowMap, and CascadedShadowMap examples. Please see the source code for these examples for more details on how to emulate depth textures.

6.2.6 Vertex/Pixel Shaders

Porting between GLSL and HLSL vertex/pixel shaders is mostly straightforward. Several types have different names (e.g., vec4 in GLSL is float4 in HLSL) and these changes can usually be done via simple find-and-replace. Most of the functionality in GLSL has an equivalent syntax in HLSL as detailed below.

Vertex Attributes and Varyings

In GLSL, all vertex shader inputs are declared as attributes. In HLSL, the vertex shader inputs are provided as arguments to the vertex shader main function and given semantics. As described in section 6.2.1 , all of the vertex shader inputs are bound to semantics and the *ID3D11InputLayout* object describes how the vertex data maps to these semantics. In GLSL, variables are passed between the vertex and pixel shader using interpolators called varyings. In HLSL, the vertex shader main function returns the interpolators and the same return types are used as the argument to the pixel shader. Further, all of the interpolators are explicitly assigned to semantics in HLSL.

The following vertex shader demonstrates the use of attributes and varyings in GLSL:

```
uniform mat4  g_MatModelViewProj;
uniform mat4  g_MatModel;
uniform vec3  g_LightPos;
uniform vec3  g_EyePos;

attribute vec4 g_PositionIn;
attribute vec2 g_TexCoordIn;
attribute vec3 g_TangentIn;
attribute vec3 g_BinormalIn;
attribute vec3 g_NormalIn;

varying vec2 g_TexCoord;
varying vec4 g_LightVec;
varying vec3 g_ViewVec;
varying vec3 g_Normal;

void main()
{
    vec4 Position  = g_PositionIn;
    vec2 TexCoord  = g_TexCoordIn;
    vec3 Tangent   = g_TangentIn;
    vec3 Binormal  = g_BinormalIn;
    vec3 Normal    = g_NormalIn;

    gl_Position = g_MatModelViewProj * Position;
```

```
    g_TexCoord = TexCoord;
    g_Normal   = Normal;

    vec4 WorldPos = g_MatModel * Position;
    vec3 lightVec = g_LightPos - WorldPos.xyz;
    vec3 viewVec  = g_EyePos   - WorldPos.xyz;

    // Transform vectors into tangent space
    g_LightVec.x = dot( lightVec, Tangent);
    g_LightVec.y = dot( lightVec, Binormal);
    g_LightVec.z = dot( lightVec, Normal);
    g_LightVec.w = saturate(1.0 - 0.03 * dot(g_LightVec.xyz, g_LightVec.xyz));
    g_LightVec.xyz = normalize(g_LightVec.xyz);

    g_ViewVec.x  = dot( viewVec, Tangent );
    g_ViewVec.y  = dot( viewVec, Binormal );
    g_ViewVec.z  = dot( viewVec, Normal );
    g_ViewVec = normalize(g_ViewVec);
}
```

This same vertex shader in HLSL using input and output semantics is shown below.

```
cbuffer MaterialConstantBuffer
{
    float4x4 MatModelViewProj;
    float4x4 MatModel;
    float4   LightPos;
    float4   EyePos;
};

struct VertexShaderInput
{
    float4 Position : POSITION;
    float2 TexCoord : TEXCOORD0;
    float3 Tangent : TANGENT;
    float3 Binormal : BINORMAL;
    float3 Normal : NORMAL;
};

struct PixelShaderInput
{
    float4 Position : SV_POSITION;
    float2 TexCoord : TEXCOORD0;
    float4 LightVec : TEXCOORD1;
    float3 ViewVec : TEXCOORD2;
    float3 Normal : TEXCOORD3;
};

PixelShaderInput main(VertexShaderInput input)
{
    PixelShaderInput vso;

    float4 Position  = input.Position;
    float2 TexCoord  = input.TexCoord;
    float3 Tangent   = input.Tangent;
    float3 Binormal  = input.Binormal;
    float3 Normal    = input.Normal;

    vso.Position = mul(MatModelViewProj,  Position);
    vso.TexCoord = TexCoord;
    vso.Normal   = Normal;

    float4 WorldPos = mul(MatModel, Position);
```

```
    float3 lightVec = LightPos.xyz - WorldPos.xyz;
    float3 viewVec  = EyePos.xyz   - WorldPos.xyz;

    // Transform vectors into tangent space
    vso.LightVec.x = dot( lightVec, Tangent);
    vso.LightVec.y = dot( lightVec, Binormal);
    vso.LightVec.z = dot( lightVec, Normal);
    vso.LightVec.w = saturate( 1.0 - 0.03 * dot( vso.LightVec.xyz,
                               vso.LightVec.xyz ) );
    vso.LightVec.xyz = normalize(vso.LightVec.xyz);

    vso.ViewVec.x  = dot( viewVec, Tangent );
    vso.ViewVec.y  = dot( viewVec, Binormal );
    vso.ViewVec.z  = dot( viewVec, Normal );
    vso.ViewVec = normalize(vso.ViewVec);

    return vso;
}
```

Another difference you will notice between these two vertex shaders is the use of the *mul()* intrinsic function for multiplying the *MatModelViewProj* matrix times the *Position*. In GLSL, the * operator is used for multiplication by matrices, but in HLSL it must be done explicitly using the *mul()* intrinsic function.

Textures

As mentioned in section 6.2.2 , Direct3D11 separates sampler state from texture state. In HLSL, you must declare both a Texture and SamplerState in order to be able to fetch from a texture. The following block of code shows fetching from a texture in GLSL in OpenGL ES 2.0:

```
uniform sampler2D g_DiffuseTexture;
varying vec2 g_TexCoord;
void main()
{
    vec4 vColor = texture2D( g_DiffuseTexture, g_TexCoord );
    // ...
}
```

This same code is written in HLSL as follows:

```
Texture2D tDiffuseTexture : register(t0);
SamplerState sDiffuseTexture : register(s0);
// ...
PixelShaderOutput main(PixelShaderInput input)
{
    float4 vColor = tDiffuseTexture.Sample(sDiffuseTexture, input.TexCoord );
    // ...
}
```

Note the use of the register(##) keyword. This is used to bind the texture and sampler to a specific unit. In GLSL, this is done via setting the value of the uniform, but in HLSL it is done explicitly in the shader code.

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	
Resources		
R1	*Programming Guide for Direct3D11* http://msdn.microsoft.com/en-us/library/windows/desktop/ff476345(v=vs.85).aspx	Microsoft
R2	*Visual C++ and WinRT/Metro - Some fundamentals* by Nish Sivakumur http://www.codeproject.com/Articles/262151/Visual-Cplusplus-and-WinRT-Metro-Some-fundamentals	Codeproject
R3	*Programming Windows®, 6th Edition* by Charles Petzold	Microsoft Press

8 Revision History

Revision	Date	Description
A	October 2012	Initial release
B	August 2013	Minor updates