Mali™ GPU OpenGL ES

Application Development Guide

Copyright © 2007-2009 ARM. All rights reserved.
ARM DUI 0363D (ID111709)

Mali GPU OpenGL ES
Application Development Guide

Copyright © 2007-2009 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
06 December 2007	A	Non-Confidential	First Release
29 May 2008	B	Non-Confidential	Second Release
30 September 2008	C	Non-Confidential	Update for OpenGL ES extensions support
17 November 2009	D	Non-Confidential	Update for Mali Developer Portal

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
Mali GPU OpenGL ES Application Development Guide

Preface

About this book vii
Feedback ix

Chapter 1 Introduction

1.1 Mali system overview 1-2
1.2 Graphics standards 1-3
1.3 Mali GPU Developer Tools 1-4

Chapter 2 Developing OpenGL ES Applications

2.1 Developing applications 2-2
2.2 Mali rendering strategy 2-3
2.3 OpenGL ES limitations 2-4
2.4 Supported OpenGL ES extensions 2-8
2.5 Optimizing application speed 2-9
2.6 Recommendations and best practices 2-10
2.7 Identifying problems in applications 2-19

Appendix A OpenGL ES 2.0 Limit Values and Optional Language Features

A.1 Optional language features A-2
A.2 Limit values A-3

Glossary

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

List of Tables
Mali GPU OpenGL ES Application Development Guide

	Change history	ii
Table 2-1	Relative costs of common shader program operations	2-15
Table 2-2	Application problems and suggested solutions	2-19
Table A-1	Mali GPU implementation values	A-3

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

List of Figures
Mali GPU OpenGL ES Application Development Guide

Figure 2-1 Translation matrix with code 2-5
Figure 2-2 Graphics application and shader program communication 2-6

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Preface

This preface introduces the *Mali GPU OpenGL ES Application Development Guide*. It contains the following sections:

- *About this book* on page vii
- *Feedback* on page ix.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

About this book

This is the *OpenGL ES Application Development Guide* for the *Mali GPU*. It provides guidelines for using the OpenGL ES 1.1 and OpenGL ES 2.0 APIs to develop applications for a Mali GPU.

This document applies to the Mali GPU range, that is Mali-55, Mali-200, and Mali-400 MP. Any differences for particular GPUs are clearly indicated. The document describes how to achieve optimal use of the hardware and software.

Use this guide in conjunction with the other Mali GPU documentation. See *Additional reading* on page viii for a list of the other available documentation.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this book, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This guide is written for programmers who are programming a *System-on-Chip* (SoC) that uses a Mali GPU. It assumes that you have experience in software development, and are familiar with graphics software terminology.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the Mali GPU, the software architecture, and the Mali GPU developer tools you can use to develop OpenGL ES applications on the Mali GPU.

Chapter 2 *Developing OpenGL ES Applications*

Read this chapter for information about optimizing application performance and identifying problems when developing OpenGL ES application software.

Appendix A *OpenGL ES 2.0 Limit Values and Optional Language Features*

Read this chapter for information about language features that you can use in OpenGL ES 2.0 implementations, and also about the sizes of various shader resources.

Glossary Read this for definitions of terms used in this book.

Conventions

The typographical conventions that this book can use are:

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace` Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

monospace	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
monospace italic	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *Mali GPU OpenVG Application Development Guide* (ARM DUI 0380)
- *Mali GPU Developer Tools Technical Overview* (ARM DUI 0501)
- *Mali GPU Performance Analysis Tool User Guide* (ARM DUI 0502)
- *Mali GPU Texture Compression Tool User Guide* (ARM DUI 0503)
- *Mali GPU Shader Development Studio User Guide* (ARM DUI 0504)
- *Mali GPU Shader Library User Guide* (ARM DUI 0510)
- *Mali GPU Offline Shader Compiler User Guide* (ARM DUI 0513)
- *Mali GPU Demo Engine User Guide* (ARM DUI 0505)
- *Mali GPU OpenGL ES 2.0 Emulator User Guide* (ARM DUI 0511)
- *Mali GPU OpenGL ES 1.1 Emulator User Guide* (ARM DUI 0506)
- *Mali GPU Binary Asset Exporter User Guide* (ARM DUI 050).

Other publications

This section lists relevant documents published by third parties:

- *OpenGL 2.1 Specification*, `http://www.opengl.org`
- *OpenGL ES Common/Common-Lite Profile Specification Version 1.1*, `http://www.khronos.org`
- *OpenGL ES Common Profile Specification Version 2.0*, `http://www.khronos.org`
- *OpenGL ES Common Profile Specification Version 1.1*, `http://www.khronos.org`
- *OpenGL ES Shading Language (GLSL ES)*, `http://www.khronos.org`
- *EGL 1.4 Specification*, `http://www.khronos.org`
- *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2* (5th Edition, 2005), Addison-Wesley Professional. ISBN 0-321-33573-2
- *OpenGL Shading Language* (2nd Edition, 2006), Addison-Wesley Professional. ISBN 0-321-33489-2.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product then contact malidevelopers@arm.com and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- the title
- the number, ARM DUI 0363D
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 1
Introduction

This chapter introduces the *Mali GPU OpenGL ES Application Development Guide*. It contains the following sections:

- *Mali system overview* on page 1-2
- *Graphics standards* on page 1-3
- *Mali GPU Developer Tools* on page 1-4.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

1.1 Mali system overview

The Mali *Graphics Processing Unit* (GPU) forms the basis of a high performance graphics processing solution. When implemented as part of a *System-on-Chip* (SoC) device, the GPU forms an integral part of the graphics solution.

Programmable hardware, such as the Mali-200 GPU or Mali-400 *Multi Processor* (MP) GPU, consists of programmable processors, one or more pixel processors, and a geometry processor. The geometry processor performs all geometric and vertex processing and passes this information, as data structures, to the pixel processors. The pixel processors perform rendering to produce the final image.

Entry-level, fixed-function hardware might not have a dedicated hardware geometry processor. For example, the Mali-55 GPU consists of a pixel processor that performs rasterization according to the OpenGL ES 1.1 standard. The geometric operations for the Mali-55 GPU are performed by software running on the CPU.

Note

- The Mali-200 and Mali-400 MP GPUs support both the OpenGL ES 1.1 and OpenGL ES 2.0 graphics standards.
- Fixed-function hardware such as the Mali-55 GPU only supports OpenGL ES 1.1.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

1.2 Graphics standards

The Mali GPUs support the OpenGL ES APIs, these are subsets of the full OpenGL APIs. OpenGL ES contains 2D and 3D graphics functionality specifically for embedded system applications on mobile handsets, *Personal Digital Assistants* (PDAs), and other handheld devices. The Mali GPUs use OpenGL ES to provide a low-level interface between graphics software and hardware graphics acceleration.

The Mali GPUs also support the OpenVG API. OpenVG contains 2D functionality for hardware accelerated vector and raster graphics. See the *Mali GPU OpenVG Application Development Guide*.

Specifically, fixed-function Mali GPUs, such as the Mali-55, support:

OpenGL ES 1.1

OpenGL ES 1.1 is a subset of the OpenGL 1.5 standard that implements a fixed-function pipeline.

EGL 1.3

EGL 1.3 specifies how OpenGL ES drivers are integrated with a platform-specific windowing system.

In addition to OpenGL ES 1.1, programmable hardware GPUs, such as the Mali-200 and Mali-400 MP, also support:

OpenGL ES 2.0

OpenGL ES 2.0 is a subset of the OpenGL 2.0 standard that implements a pipeline with application-programmable vertex and fragment processing. You use the OpenGL *ES Shading Language* (ESSL) to specify vertex and fragment shader programs.

EGL 1.4

EGL 1.4 specifies how OpenGL ES drivers are integrated with a platform-specific windowing system.

The OpenGL ES drivers are implementations of these standards that control the graphics hardware. The appropriate drivers are included with the Mali GPUs.

See `http://www.khronos.org` for more information about these graphics standards and the OpenGL ES drivers.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

1.3 Mali GPU Developer Tools

The Mali GPU Developer Tools consist of the following development tools to help you optimize OpenGL ES application development:

OpenGL ES 1.1 Emulator

The OpenGL ES 1.1 Emulator is a library that maps OpenGL ES 1.1 API calls to OpenGL 2.0 API calls to be executed on a graphics card in a standard PC. The graphics card must support at least OpenGL 2.0 for the emulator to work. There are emulators for Windows and Linux.

OpenGL ES 2.0 Emulator

The OpenGL ES 2.0 Emulator is a library that maps OpenGL ES 2.0 API calls to OpenGL 2.0 API calls to be executed on a graphics card in a standard PC. The graphics card must support OpenGL 2.0 for the emulator to work. There are emulators for Windows and Linux.

Mali GPU Performance Analysis Tool

The Performance Analysis Tool helps you analyze graphics application performance by studying hardware and software performance counters produced by the Mali GPU and OpenGL ES Instrumented Drivers respectively. The Performance Analysis Tool can display any of a set of performance counter values for each frame.

Mali GPU Texture Compression Tool

You can run the Mali GPU Texture Compression Tool on your computer to encode texture images into formats that take less memory than the original. This enables you to reduce the amount of memory bandwidth required to read texture data. The reduced bandwidth results in superior performance and reduced power consumption. The Mali GPU supports *Ericsson Texture Compression* (ETC). ETC is the compression scheme recommended by the OpenGL ES Working Group. The Texture Compression tool has both a graphical and command-line interface.

Mali OpenGL ES Instrumented Drivers

The Mali OpenGL ES Instrumented drivers are a special build of the Mali OpenGL ES drivers with added debug functionality. These drivers contain functionality for sampling of hardware and software performance counters, additional debug output, dumping of driver state, and rendering overrides for visual debugging.

Mali GPU Demo Engine

The Mali GPU Demo Engine is a library of C++ functions that aid building OpenGL ES applications for an ARM platform with Mali GPU acceleration. You can run the Mali GPU Demo Engine on a Microsoft Windows workstation, a Linux workstation, and on an ARM platform with a Mali GPU. You can use the Mali GPU Demo Engine to create new applications, deliver training, and explore implementation possibilities.

Mali GPU Binary Asset Exporter

The Mali GPU Binary Asset Exporter is a program that runs on your desktop computer and converts graphic assets from the COLLADA format to the Mali Binary Asset format. These graphic assets include such elements as:

- geometry data
- textures

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

- lighting
- movements for animation.

The assets can be imported in COLLADA format that can be generated by commercial 3D modelling tools such as Google SketchUpPro and Autodesk 3ds Max.

Note

The Mali GPU Demo Engine requires assets in Mali Binary Asset format so that any COLLADA assets must be converted to Mali Binary Asset format before the assets are used in the Demo Engine.

Mali GPU Shader Development Studio

The Mali GPU Shader Development Studio is an Eclipse plug-in that extends the functionality of the Eclipse platform and enables editing of OpenGL ES shaders. You can use it to start developing shaders, or to work on existing shaders and shader effects.

You can preview shaders as they are being developed by rendering them on remote OpenGL ES hardware or on local or remote emulations. You can modify various shader variables and use the Mali GPU Shader Development Studio to view the corresponding changes in the effects.

Note

The following shader tools are not applicable to fixed-function hardware such as the Mali-55 GPU:

- Mali GPU Shader Development Studio
- Mali GPU Shader Library
- Mali GPU Offline Shader Compiler.

Mali GPU Shader Library

The Mali GPU Shader Library is a collection of example shader programs. These examples contain the ESSL vertex and fragment shader source files for shader programs and other information to help you start developing shader programs for the Mali GPU. You can use the shader programs as they are provided, modify them to suit your requirements, or use them to learn and develop your own programs.

Mali GPU Offline Shader Compiler

The Mali GPU Offline Shader Compiler translates vertex shaders and fragment shaders written in the OpenGL *ES Shading Language* (ESSL) into binary vertex and fragment shaders.

The OpenGL ES Emulator and Mali GPU Shader Development Studio use the Mali GPU Offline Shader Compiler to check the syntax of shaders before they are sent for rendering. Each shader is compiled in the background and any warnings or errors that are generated are collected.

See *Mali GPU Developer Tools Overview* for more information about the Mali GPU developer tools.

You can use the Mali GPU developer tools together with standard ARM development tools such as *RealView® Developer Suite* (RVDS). See `http://www.arm.com` for a complete list of ARM software development tools. Many of these tools are semihosted, so that you can operate them from your desktop computer, using a tool such as RealView ICE® for communication with the development platform.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Chapter 2
Developing OpenGL ES Applications

This chapter describes how to develop applications for the Mali GPU, using OpenGL ES. It contains information about different development approaches, how to optimize your applications, and how to locate problems in your applications.

This chapter contains the following sections:

- *Developing applications* on page 2-2
- *Mali rendering strategy* on page 2-3
- *OpenGL ES limitations* on page 2-4
- *Supported OpenGL ES extensions* on page 2-8
- *Optimizing application speed* on page 2-9
- *Recommendations and best practices* on page 2-10
- *Identifying problems in applications* on page 2-19.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.1 Developing applications

The different approaches involved in developing a graphics application for embedded devices use various hardware and software tools, such as compilers, graphics drivers, debuggers, and communications facilities. You can use any of the following approaches to develop OpenGL ES applications:

- If you are targeting a specific device that contains a Mali GPU, obtain a development kit for that device. Contact the device manufacturer for information about obtaining a development kit.
- Obtain a hardware development platform for the Mali GPU from the Mali Development Center to help you to evaluate the Mali GPU and design graphics software for different operating systems. See `http://www.malideveloper.com` for more information.
- Obtain the OpenGL ES 1.1 or 2.0 Emulator from the Mali Developer Center `http://www.malideveloper.com` to help you getting started developing your OpenGL ES application on your desktop PC.

  Note

  GPUs for desktop computers implement the full OpenGL standard, and not OpenGL ES. Ensure that your application uses only the functions available in the OpenGL ES subset. Also, if you are developing OpenGL ES 2.0 content, ensure that the GPU in your desktop PC supports programmable Fragment and Vertex shaders with Shader Model 3.0 or higher.

If you are using programmable graphics hardware, ensure that your shaders are compatible with both the *OpenGL ES Shading Language* (ESSL). Use the Offline Shader Compiler to check that your shaders are valid ESSL and to establish the number of execution cycles that your shaders require. See *Shader programs* on page 2-13 for more information.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.2 Mali rendering strategy

The Mali GPUs use *tile-based immediate-mode* rendering.

For this type of rendering, the framebuffer is divided into tiles of size 16 by 16 pixels. The *Polygon List Builder* (PLB) organizes input data from the application into polygon lists. There is a polygon list for each tile. When a primitive covers part of a tile, an entry, called a polygon list command, is added to the polygon list for the tile.

The pixel processor takes the polygon list for one tile and computes values for all pixels in that tile before starting work on the next tile. Because this tile-based approach uses a fast, on-chip tile buffer, the GPU only writes the tile buffer contents to the framebuffer in main memory at the end of each tile. Non-tiled-based, immediate-mode renderers generally require many more framebuffer accesses. The tile-based method therefore consumes less memory bandwidth, and supports operations such as depth testing, blending and anti-aliasing efficiently.

Note

The pixel processors in the Mali-200 and Mali-400 MP GPUs support early-Z rejection of fragments when depth testing is enabled. If depth testing is set to *less than or equal* for example, when a fragment is generated, the hardware tests whether the fragment depth is greater than that of the same pixel in previously-generated fragments. If so, the new fragment is discarded before it reaches the fragment shader.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.3 OpenGL ES limitations

This section describes:

- *Differences between OpenGL and OpenGL ES*
- *Running code on OpenGL and OpenGL ES*.
- *Viewing transforms* on page 2-5
- *Shader programming* on page 2-5.

2.3.1 Differences between OpenGL and OpenGL ES

OpenGL ES 1.1 and OpenGL ES 2.0 are subsets of the full OpenGL standard. When using the OpenGL ES API, there are limitations that you must be aware of when developing your applications.

For example, the following OpenGL functionality is not present in either OpenGL ES 1.1 or OpenGL ES 2.0:

- There is no support for `glBegin` or `glEnd`. Use vertex arrays and vertex buffer objects instead.
- The only supported rasterization primitives are points, lines and triangles. Quads are not supported.
- There is no polynomial function evaluation stage.
- You cannot send blocks of fragments directly to individual fragment operations.
- There is no support for display lists.

In addition, the following OpenGL functionality is not present in OpenGL ES 2.0:

- There is no support for the fixed-function graphics pipeline. You must use your own vertex and fragment shader programs.
- There is no support for viewing transforms such as `glFrustumf`. You must compute your own transformation matrix, pass it to the vertex shader as a uniform variable, and perform the matrix multiplication in the shader.
- There is no support for specialized functions such as `glVertexPointer` and `glNormalPointer`. Use `glVertexAttribPointer` instead.

Note

In OpenGL ES 2.0, perspective divide, frustum clipping and scissoring functionality are handled by fixed-function steps.

See the *OpenGL ES 1.1 Specification* and *OpenGL ES 2.0 Specification* for more information about the differences between OpenGL and OpenGL ES.

2.3.2 Running code on OpenGL and OpenGL ES

If you are writing code to run on both OpenGL 2.0 and OpenGL ES 2.0, be aware of slight differences between the shader languages used in the two APIs. For example, OpenGL ES 2.0 requires a precision declaration in the fragment shader code, while OpenGL 2.0 does not support this.

To enable your code to run on both OpenGL 2.0 and OpenGL ES 2.0, start every fragment shader with:

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

```
#ifdef GL_ES
 precision mediump float;
#endif
```

2.3.3 Viewing transforms

When computing your own viewing matrix, ensure you understand how OpenGL ES handles matrices.

When you multiply the position vector with several matrices to achieve transformations such as translation, rotation, scaling or projection, the effect when applying the resulting matrix on a position vector is as if the individual matrix multiplication operations were performed from right to left. This means that the usual sequence of matrices is, from left to right:

1. Projection matrix
2. View matrix
3. Any number of model transformation matrices, from most global to most local.

You must then multiply the resulting matrix by the vertex position with the matrix on the left.

It is common practice to present transformation matrices in row major order. However, when storing one or two-dimensional arrays in the C programming language, OpenGL ES assumes matrices are stored in column major order. This means that when you write matrices as constants, they appear transposed. This applies to constants in your application program in addition to shader programs. Figure 2-1 shows an example of a translation matrix.

```
float move_matrix[]={
   1.0f, 0.0f, 0.0f, 0.0f,          1  0  0  X
   0.0f, 1.0f, 0.0f, 0.0f,     =    0  1  0  Y
   0.0f, 0.0f, 1.0f, 0.0f,          0  0  1  Z
   x,    y,    z,    1.0f           0  0  0  1
};
```

Figure 2-1 Translation matrix with code

2.3.4 Shader programming

When using OpenGL ES 2.0, you must always provide a vertex shader and a fragment shader program. The vertex shader program is executed once for each vertex that is passed to the API calls. The fragment shader program is executed once for each resulting fragment.

Note

Fixed-function hardware such as the Mali-55 GPU does not support shader programs.

The simple example application in Figure 2-2 on page 2-6 illustrates communication between the application and shader programs.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

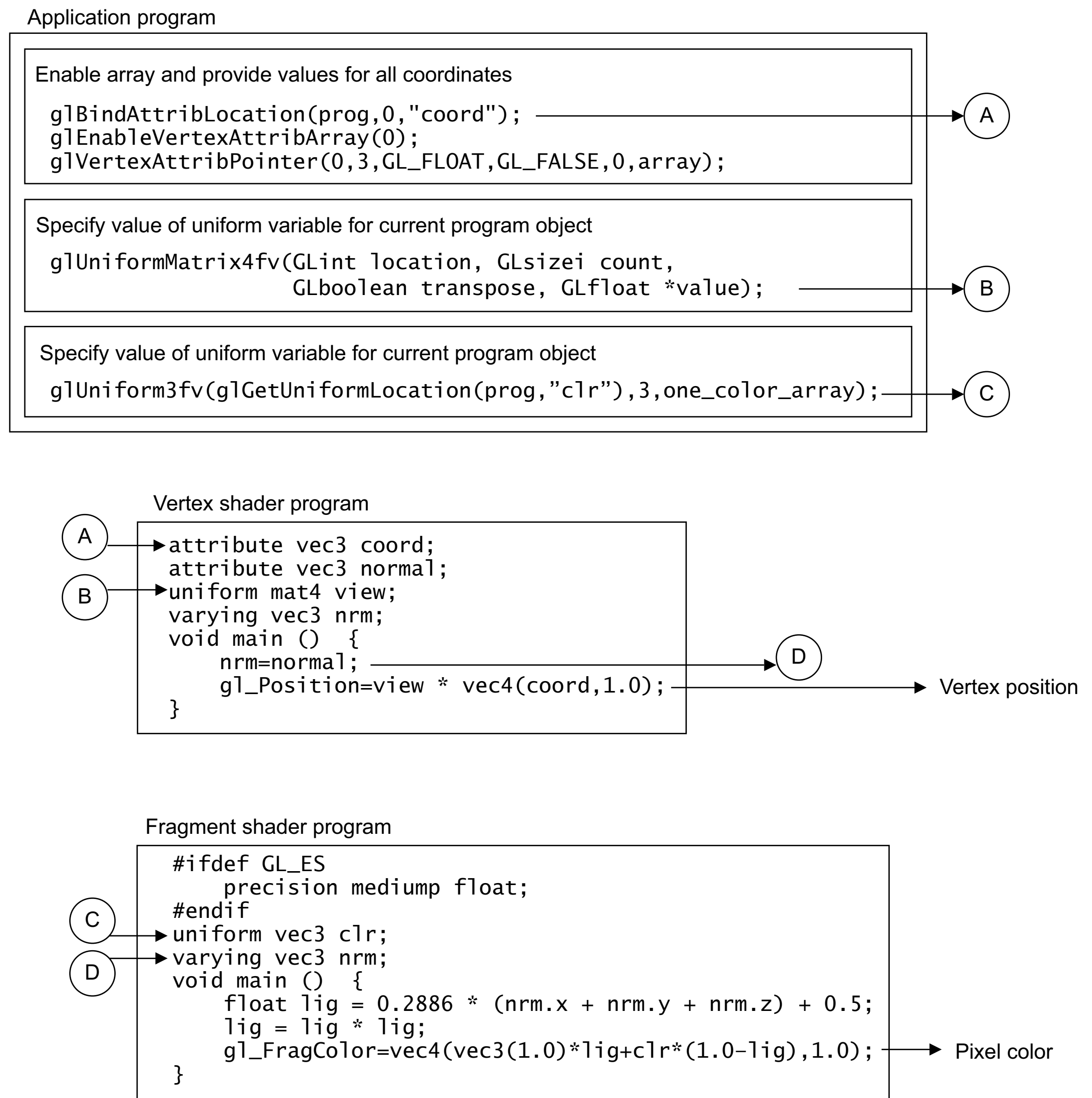

Figure 2-2 Graphics application and shader program communication

The communication flow shown in Figure 2-2 is:

1. To specify vertex coordinates, your application calls `glBindAttribLocation()`. The `coord` attribute specifies the vertex coordinates `x`, `y`, `z`, and `w`.

   Note

   To have any effect, the application must call `glBindAttribLocation()` before the program is linked.

   The application then enables the array using `glEnableVertexAttribArray`.

   The function `glVertexAttribPointer()` provides values for the vertex coordinates. In the example, the value 3 indicates that there are three components for each vertex `x`, `y`, and `z`. The `array` is the array of vertex coordinates. The vertex shader is run once for each vertex. Each time the program runs, it starts with its set of such vertex attributes, as values of attribute variables.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2. The application can similarly supply uniform values to the shader program. When calling one of the `glUniform` functions, your application must already have called `glGetUniformLocation` to find the location value to use. The call to the `glUniform` function specifies the value that appears in that uniform variable in the shader programs. The uniform variable value remains the same for all elements drawn, until your application makes another call to `glUniform`, with the same location.

3. The homogenous coordinate is output from the vertex shader program by assigning the value to the standard, built-in variable `gl_Position`.

4. Other values are output from the vertex shader by assigning them to varying variables. These are special variables that store interpolated values. Varying values from the vertex shader program become the values of the varying variables in the fragment shader program. More precisely, the varying variables for the three corner vertices are interpolated to produce values for all the fragments of each triangle.

5. Uniform variables specified by the application appear with values in the fragment shader in the same way as in the vertex shader.

6. The color result from the fragment shader is generated by assigning an RGBA value to the standard variable `gl_FragColor`.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.4 Supported OpenGL ES extensions

OpenGL ES supports extensions that can provide additional functionality.

Note

- The extensions supported depend on:
  - the version of the Mali OpenGL ES drivers release
  - the extensions exposed by the SoC manufacturer.
- See the Mali OpenGL ES drivers documentation which you can obtain from the SoC manufacturer for information on the extensions supported in your target.

For general background on extensions, see the *OpenGL ES 1.1 Specification* and *OpenGL ES 2.0 Specification* at `http://www.khronos.org`.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.5 Optimizing application speed

The execution speed of an Open GL ES application is, in general, limited by a bottleneck somewhere in the processing pipeline. The most common places for bottlenecks are:

- in the application code, for example, in collision detection
- during the transfer of data to and from the main memory
- during vertex processing on the geometry processor
- during fragment processing on the pixel processor.

You can locate bottlenecks using the Mali Performance Analysis Tool, or another performance analysis tool. These tools enable you to observe the load on the different processing stages. See the *Mali GPU Performance Analysis Tool User Guide*.

See the remaining sections of this chapter for information about how to produce efficient OpenGL ES applications.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.6 Recommendations and best practices

This section contains recommendations for obtaining the best performance from your applications, when running on a Mali GPU.

This section contains information about:

- *Textures*
- *Antialiasing* on page 2-11
- *Draw mode* on page 2-11
- *Vertex buffer objects* on page 2-11
- *Data precision* on page 2-12
- *Volume of data processed* on page 2-12
- *Render targets* on page 2-12
- *Processing pipeline* on page 2-13
- *Shader programs* on page 2-13
- *Shader arithmetic* on page 2-16
- *Other recommendations* on page 2-16
- *Additional recommendations for OpenGL ES 1.1* on page 2-17.

2.6.1 Textures

High-resolution textures might use large amounts of memory, and represent a major load on the Mali GPU. You can achieve more efficient texture cache utilization by observing the following recommendations:

- Do not use larger textures than are necessary.
- Always enable mipmapping for textures that might sometimes be rendered scaled down. This is usually the case for textures on 3D models.
- Where possible, order triangles so that triangles covering adjacent regions of the texture are close to each other in the drawing order.

Texture compression

In general, use compressed textures in applications. This limits the amount of memory occupied, and also the memory read and write bandwidth.

The Mali-55, Mali-200, and Mali-400 MP GPUs support the ETC texture compression format which reduces the size of a texture to 4 *bits-per-pixel* (bpp).

Texture compression offers benefits such as improved performance, lower memory requirements, and better cache utilization, but can result in reduced image quality. However, for textures typically used in game applications, the quality of the compressed textures is acceptable. Use uncompressed textures only when the quality of compressed textures is unacceptable.

Note

ETC does not support an alpha component for pixel colors.

ARM provides the tools required to create ETC format textures. See the *Mali GPU Texture Compression Tool User Guide*.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

In general, use as low precision as possible, especially when compression cannot be used. See *Data precision* on page 2-12 for more information about data precision. Also, use RGB rather than RGBA, unless you require the alpha component. For OpenGL ES 2.0, if your texture is monochrome, use an 8-bit gray scale texture and add color using a uniform variable.

2.6.2 Antialiasing

The GPU supports 4x multisampled *Full Scene Anti-Aliasing* (FSAA), or *Multisampled Anti-Aliasing* (MSAA), with negligible performance loss. When activated, anti-aliasing is performed by recording the polygon that is visible for each of 4 sub-pixels for each pixel, and then blending the color of these to form the final pixel. You can enable 4x FSAA by specifying an EGL configuration setting when the graphics context and rendering surfaces are created.

In addition to 4x FSAA, the Mali GPU also partially supports 16x FSAA resulting at an approximate four-times performance cost in pixel processing compared to 4x.

2.6.3 Draw mode

For large meshes, each vertex is typically included in several triangles. The number of times that such a vertex is processed depends on the function that performs the draw operation.

Using the API function `glDrawElements`, the attributes of one vertex have to be processed only once. The alternative function, `glDrawArrays`, causes each set of vertex data to be transferred and processed each time it is used in a triangle. For this reason, the most efficient way to pass geometry data when using typical meshes, is usually with indexed mode, using `glDrawElements`.

Alternatively, restructure the mesh into triangle fans or triangle strips, as this reduces the number of times each vertex occurs in the vertex array. You can also use triangle fans or triangle strips to reduce the number of indices transferred when `glDrawElements` is used.

——— Note ———

Drawing a mesh using triangle fans or strips requires more draw calls than when using individual triangles. Because each draw call has some amount of processing overhead, splitting the mesh could easily move the processing bottleneck from vertex processing to draw call overhead, eliminating the advantage of performing less vertex processing. See *Identifying problems in applications* on page 2-19 for more information about bottlenecks.

———————

Store vertex data tightly and in the order that it is to be used. This improves the effectiveness of the vertex cache, by minimizing the amount of data transferred from RAM to the vertex cache.

2.6.4 Vertex buffer objects

Use vertex buffer objects whenever possible, preferably with the usage parameter set to `GL_STATIC_DRAW`. This limits memory bandwidth usage and permits extra optimizations in the driver.

If there is more than one vertex attribute array with the same lifetime, it is particularly effective to store these arrays interleaved in one buffer object. This improves the effect of prefetch and caching in the vertex loader.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.6.5 Data precision

You can save both memory space and bandwidth by minimizing the amount of data that is passed through the OpenGL ES API. Use shorter, lower precision data when possible, and avoid using floats and other 32-bit data types unless absolutely necessary. For example, where possible:

- define vertex positions using `GL_SHORT`
- define surface normals using `GL_BYTE`
- define colors using `GL_UNSIGNED_BYTE`.

Similarly, avoid using full-size integer and floating point values for texture components, if one byte is sufficient.

2.6.6 Volume of data processed

The volume of data that the Mali GPU processes is an important efficiency consideration. In general, aim to minimize the amount of data processed by observing the following recommendations:

Only draw geometry that is visible

Avoid defining geometry that is not visible in the current frame. You can achieve this by using clipping or frustum culling in your application.

Use textures efficiently

See recommendations in *Textures* on page 2-10.

Sort your geometry according to depth

Maximize the effectiveness of the Mali-200 and Mali-400 GPU early-Z rejection feature, by sorting the geometry in an approximate front-to-back order. Sorting your draw calls according to depth is usually sufficient.

Note

- In general, you must avoid defining geometry that is not visible in the current frame. You can achieve this by using clipping or frustum culling in your application.
- Be aware however that extensive analysis to determine which features are visible might take more time than rendering the invisible geometry. Concentrate on coarse-grain culling and let the GPU do fine-grain culling.

2.6.7 Render targets

The following efficiency considerations relate to render targets:

Only render to one render target at a time

Ensure that you finish all calls for one render target before moving onto the next. Render all the textures in a cause-and-effect order:

- render to textures before the textures are used
- render the back buffer last.

Plan your use of textures

Do not modify a texture after you have passed it as a parameter to an API function, until that frame is rendered. Such modification generally forces the driver to make copies of the texture. Instead, set up all the textures that you require for a frame, before you start making API calls that reference any of them.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Avoid using glReadPixels

Avoid using `glReadPixels`. This slows down operations severely, even if only a small number of pixels are read. In many cases you can achieve the required effect more efficiently by generating the result in two stages:

1. Render to a texture, that is to a frame buffer object.
2. When using this texture, access the pixels computed in step 1.

2.6.8 Processing pipeline

The following efficiency considerations relate to the graphics processing pipeline:

Use `eglSwapBuffers`

If your application displays animations, ensure your application terminates the specification of a frame, and starts processing it, by calling `eglSwapBuffers()`. The application then produces the next frame. This process ensures that a stable image remains on the display while the next frame is being computed.

Limit number of vertices that `glDrawElements` **processes**

After a call to `glDrawElements()`, polygon list building cannot start until the preceding step, vertex shading or transform and lighting, has been completed. To enable these units to work in parallel, ensure that no single call to `glDrawElements()` comprises more than approximately one fifth of the total number of vertices. This is especially important for calls immediately before or immediately after a call to `glDrawArrays()`.

——— **Note** ———

- On desktop OpenGL systems, using `glDrawArrays` instead of `glDrawElements` can sometimes reduce the latency between the API calls and the final image appearing on the display. However, the Mali GPU uses deferred rendering. This means that rendering does not start until all geometry for a frame has been specified.
- Deferred processing and the pipeline nature of the Mali GPU implementation results in some latency. Typically, three frames are in various stages of processing while a fourth is displayed. At rates below 20 frames per second, you might notice a delay between a button being pushed and observing a reaction in the image. So, if you want a very fast-moving game, either simplify your scene, or optimize processing to get a higher frame rate.

2.6.9 Shader programs

This section contains information about recommended practices for shader programs. It also introduces the concept of costs within programs and your program structure affects execution speed.

Shader program general recommendations

For best performance when using shaders, observe the following recommendations:

Perform shading language compiler calls first

Ensure you make all calls to the shading language compiler during application startup, before you start to supply geometry or texture data to the driver.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

The compiler uses main memory for its internal data. This memory space can be reused for data such as user application data, vertex attributes, textures, and polygon lists, after the compiler has finished.

Use custom shader programs

In general, aim to use a large number of shader programs tailored to the requirements of each surface, rather than fewer general purpose shader programs with optional features that are controlled by uniform values. Specialized shaders generally run faster.

Consider program size

You can use the stand-alone version of the Offline Shader Compiler to check the size of programs. You can also use the compiler to experiment with programs and to see how your changes affect the number of instructions.

Note

The sizes reported by the compiler relate to the native size of instruction words in the hardware. Each instruction word can contain a number of ESSL operations.

Looping and conditional branching

Do not unroll your loops manually. Instead, organize your data in arrays and process these with a `for` statement where possible. Also, use `if` statements when doing so is natural. If it is beneficial to do so, the compiler unfolds the `if` statement to execute both branches and select between the results.

Avoid using too many varyings when using ESSL

When programming shaders in ESSL, economize on the number of varyings used in the fragment shader program. Varyings consume memory bandwidth when they are transferred between the geometry processor and memory, and between memory and the pixel processor.

Avoid using too many matrix multiplications

Multiplying a 4x4 matrix with a 4x vector involves 16 multiplications and 12 additions. This is therefore expensive, particularly so on the vertex shader which has such multiplications as one of its main tasks. Avoid multiplying two matrices in a shader program, because this is even more expensive. Instead, ensure that your application program computes the complete transformation matrix to be applied to each vertex attribute, and transfers it as a uniform.

For example, avoid separating the viewing transformations into rotation on one hand and the translation, scaling, and projection on the other, with the intent of applying the rotation to the normal vectors, rotation and then the others to the positions. If you must multiply a vector with multiple matrices inside the shader, multiply them onto the vector one at a time, rather than multiplying the matrices together first.

Estimating program costs

The way you write your application ultimately has some impact on the execution speed. Because packing individual operations into the instruction words is a complex combinatorial job, it is not possible to give simple numbers for the cost of any single programming construct. However, it is possible to define a relative *cost* for program constructs, where the following terms are used:

Free Operation has little or no impact on program execution speed.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Low Simple and fast operations that have low impact on execution speed.

Medium These operations have an intermediate impact on execution speed, and are likely to cost between 2-5 times the cost of a low-cost operation.

High These operations have the highest impact on execution speed, and are likely to cost between 5-20 times the cost of a low-cost operation.

Table 2-1 defines the likely relative costs of various program constructs. Consider these costs when developing your applications.

Table 2-1 Relative costs of common shader program operations

Operation	Example	Geometry processor	Pixel processor
swizzle	`.yx`	Free[a]	Free[a]
negative	`-x`	Free	Free[a]
absolute	`abs`	Low	Free[a]
`clamp` to [0,1]	`clamp(x,0.0,1.0)`	Low	Free[a]
other clamps	`clamp(x,-1.0,1.0)`	Low	Low
arithmetic operators	`+`, `-`, `*`	Low	Low[a]
minimum, maximum	min, max	Low	Low[a]
comparison		Medium	Low[a]
access local variable		Free[a]	Free[a]
access uniform		Low[a]	Low[a]
access varying		Low[a]	Medium[a]
divide	`a/b`	Medium	Medium
square root	`sqrt`	Medium	Medium
reciprocal	`1/x`	Medium	Medium
exponential, logarithm	`exp`, `log`	Medium	Medium
trigonometric	`cos`, `sin`	High	Medium
power	`pow`	High	Medium
array indexing		Medium	Medium[a]
vector indexed with variable		High	Medium
conditional statements	`if`, `for`	Medium	Low

a. Operations that the corresponding processor can do on all four components of one vector in one sub-instruction.

Note

Although Table 2-1 indicates the relative costs of various programming constructs, use the Offline Shader Compiler to obtain a more accurate idea of the likely cost of your programs.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.6.10 Shader arithmetic

Internally, the geometry processor works on 32-bit floating point values that follow the IEEE 754 standard with the exception that de-normalized, small values are taken as 0.0. The vertex shader represents integers using floating-point values.

You can transfer vertex data, from the geometry processor to the pixel processor, in several different formats. Values are converted by hardware from and to the float forms used in the shaders, at no extra cost.

The OpenGL ES 2.0 standard implies that, by default, data must be transferred as 32-bit floating point values. However, this produces much memory traffic, and the full 32-bit precision is often not necessary. To avoid using 32-bit values, set the precision for the output varying values from the vertex shader program to either `mediump` or `lowp`.

The fragment shader uses 16-bit floating point format, that consists of:

- Sign.
- 5-bit exponent, with offset 15.
- 10-bit mantissa, with an implied most significant 1-bit.

This format corresponds to approximately three decimal digits of precision.

The arithmetic for 16-bit floating point values deviates slightly from the IEEE 754 standard. For example:

- The fragment shader treats *Not A Number* (NaN) values and de-normalized values as 0.0 when they are used as input to operations.
- Converting +INF and -INF to integers yields, respectively, the largest and smallest representable values. This can occur on the output from the two shading units.

In many cases, this treatment of 16-bit floating point values results in reasonable pixel colors rather than NaN values. For example, division by zero or taking a square root of a negative argument always yields NaN. If the fragment shader uses these results as color components, they are dark. When used as texture coordinates, it has the same effect as 0.0.

If you blend the resulting color with other contributions, the error might not be noticeable. In other words, it might not be necessary to include `if` statements in your shader program to guard against division by zero or bad arguments to built-in functions. Avoid taking such precautions until you notice bad pixels on the display.

2.6.11 Other recommendations

Other recommendations for improving the efficiency of your applications are:

Use point sprites

The pixel processors in the Mali-200 and Mali-400 MP support point sprites. Therefore, use point sprites rather than triangles or quads for entities such as particles and billboards.

Use appropriate triangle dimensions

Avoid setting up long, thin triangles. The pixel processor always runs groups of four nearest neighbor fragments. Therefore, a strip that is one pixel wide takes as much time to process as a strip that is two pixels wide.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Using state changes

Avoid making redundant state changes, or changing state and then changing it back between draw calls. It can also be beneficial to group geometry with similar state together to reduce the number of state changes required.

Note

Minimizing state changes must be balanced against the advantages of front-to-back ordering, as recommended in *Volume of data processed* on page 2-12.

Clear the entire framebuffer

Always clear the entire framebuffer by calling `glClear`. Because of the way tiling works, using `glClear` is essentially a free operation. Clearing only a part of the buffer is inefficient both in terms of time and power consumption.

For example, partial clearing of the framebuffer might occur if you use scissoring to specify an area that does not cover the entire real framebuffer that was specified in EGL calls.

Note

If possible, clear all buffers, that is the color, depth, and stencil buffers, when you clear the framebuffer.

Minimize the number of draw calls

When you call `glDrawArrays` or `glDrawElements`, the graphics driver collects all current OpenGL ES states, textures and vertex attribute data. The driver processes these to generate appropriate commands for the graphics hardware to perform the specified draw operation. This process can take a significant amount of time, so if you perform many draw calls, the overhead associated with these can be the bottleneck of the rendering. This is particularly so on embedded systems, which typically have much less CPU power available than traditional desktop systems.

Attempt to reduce the number of draw calls to as few as possible. For example, if multiple objects are drawn with the same rendering parameters but using different textures, merge the textures into one large texture and adjust the texture coordinates accordingly.

The number of draw calls that can be performed depends on how the rendering states are changed between the calls. This is typically in the order of a few thousand calls per second.

Avoid using `glFlush` and `glFinish`

Do not call `glFlush` or `glFinish` unless you have no alternative. Use `eglSwapBuffers` to signal the end of rendering for a frame.

2.6.12 Additional recommendations for OpenGL ES 1.1

Additional recommendations to promote efficiency when using a Mali GPU with OpenGL ES 1.1 are:

Avoid using too many texture stages

For maximum efficiency, use as few texture stages as possible. Consider combining textures in a preprocessing or content generation step, when feasible.

The Mali-55 GPU supports superimposing up to two textures. The Mali-200 and Mali-400 MP GPUs support up to eight textures.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Avoid using too many light sources

Vertex processing load increases with the number of enabled light sources. Reduce the number of light sources if the load is too high.

Use matrix calculation functions

If possible, use the matrix calculation functions rather than calculating your own matrices. In particular, and especially for texture matrices, set identity matrices using `glLoadIdentity` rather than `glLoadMatrixf`.

Avoid user-defined clip planes

In general, avoid using user-defined clip planes, because they do not offer performance improvements.

Note

Edges caused by the intersection of a triangle with the clip plane are rendered aliased, even when anti-aliasing is enabled.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

2.7 Identifying problems in applications

If you suspect that there is a performance bottleneck at a certain point in your application, you can apply optimizations at that point. If throughput increases, then it is possible that you have correctly identified the problem. However, identifying and correcting one bottleneck might expose other bottlenecks elsewhere in the application.

In general, finding bottlenecks can be difficult. In addition to using profiling tools such as the Performance Analysis Tool, the usual approach is to increase or decrease the load on individual graphics pipeline stages, to observe the effect on performance.

Note

With multi-pass rendering, bottlenecks can be different in each pass. Be aware of this when identifying and reducing bottlenecks.

Table 2-2 contains suggested actions you can take to resolve problems in the various pipeline stages. Use these techniques to identify and reduce bottlenecks.

Table 2-2 Application problems and suggested solutions

For problems with...	Try the following approach:
Application code	Reduce the amount of processing that is unrelated to OpenGL ES calls, such as input processing, game logic, collision detection, and audio processing.
Driver overhead	Group geometry with similar state together and eliminate unnecessary state changes.
Vertex attribute transfer	Use smaller data types for the values. Also, use a more economical triangle scheme, and in general use `glDrawElements` rather than `glDrawArrays`.
Vertex shader processing, or Transform and Lighting in OpenGL ES 1.1	Try the following options: • Use `glDrawElements` rather than `glDrawArrays`. • For OpenGL ES 1.1, reduce the number of lights. • Minimize the transformations of texture coordinates. You can avoid these transformations by setting the transformation matrix using OpenGL ES 1.1 function `glLoadIdentity`. • For OpenGL ES 2.0, simplify the vertex shader program.
Polygon list building	Use fewer graphics primitives. Also, avoid drawing significant amounts of the total geometry in any single call to `glDrawElements`.
Varying data transfer	In OpenGL ES 1.1, use fewer texture coordinates. In OpenGL ES 2.0, use fewer varyings, and specify lower precision on varying variables out of the vertex shader.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Table 2-2 Application problems and suggested solutions (continued)

For problems with...	Try the following approach:
Fragment shader processing, texture, color sum, and fog in OpenGL ES 1.1	Lower the resolution of the render target or reduce the size of the viewport. For OpenGL ES 1.1, use fewer texture stages. For OpenGL ES 2.0, simplify the fragment shader program.
Texture bandwidth	Try the following options: • Use fewer texture stages. • Lower the size of the textures, by using a smaller data format for each pixel, lower resolution, or texture compression. • Use a simpler texture filtering mode. • Collapse texture coordinates so that they always read from the same position in the texture.
Transfer to display framebuffer	Try the following options: • Use a mode with lower pixel precision. • Lower the resolution of the render target.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Appendix A
OpenGL ES 2.0 Limit Values and Optional Language Features

This appendix describes various language features that you can use in OpenGL ES 2.0 implementations. It also contains information about the sizes of various shader resources, compared with the minimum size required by the *OpenGL ES 2.0 Specification*.

This chapter contains information about:

- *Optional language features* on page A-2
- *Limit values* on page A-3.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

A.1 Optional language features

The *OpenGL ES 2.0 Specification* mentions various language features that are optional in OpenGL ES 2.0 implementations. When you develop OpenGL ES 2.0 applications on a Mali GPU, you can use all of these optional language features, with the exception of vertex texturing.

The following loop statements are fully supported with no restrictions on the expressions occurring in the header or the body of the loop:

- `for`
- `while`
- `do while`.

Indexing using non-constant index expressions is supported for the following data types and variable types:

- arrays
- vectors
- matrices
- uniforms
- attributes
- varyings
- local variables
- global variables.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

A.2 Limit values

The *OpenGL ES Shading Language* specification defines minimum values for the sizes of various shader resources. In Mali GPU implementations, some of these values are larger than the minimum required by the specification. Specifically, the Mali GPU implementation values are listed in Table A-1.

Table A-1 Mali GPU implementation values

Shader Resource	Mali Implementation Value	Minimum Value
`gl_MaxVertexAttribs`	16	8
`gl_MaxVertexUniformVectors`	256	128
`gl_MaxVaryingVectors`	12	8
`gl_MaxVertexTextureImageUnits`	0	0
`gl_MaxCombinedTextureImageUnits`	8	8
`gl_MaxTextureImageUnits`	8	8
`gl_MaxFragmentUniformVectors`	256	16
`gl_MaxDrawBuffers`	1	1

These values are accessible as built-in variables in shaders.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Glossary

This glossary describes some of the terms used in Mali GPU documents from ARM Limited.

Anti-aliasing
The process of removing or reducing aliasing artefacts, primarily jagged polygon edges, from an image. Anti-aliasing is particularly important for low-resolution displays. There exist several techniques to perform anti-aliasing, see multi-sampling and super-sampling.

API
See Application Programming Interface.

API driver
A specialized driver that controls graphics hardware. Examples are OpenGL ES driver and OpenVG driver.

Application Programming Interface
A specification for a set of procedures, functions, data structures, and constants that are used to interface two or more software components together. For example, an API between an operating system and the application programs that use it might specify exactly how to read data from a file.

Blending
A process where two sets of color and alpha values are blended together to form a new set of color and alpha values for a fragment.

Byte
An 8-bit data item.

Clipping
See scissoring.

Demo Engine
See Mali Demo Engine.

Depth testing
A process that pixel processors can use to reject fragments that are not visible because they are behind other fragments. The early-Z testing process uses depth testing.

Device driver
An operating system component that communicates with the graphics hardware.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Draw mode
The OpenGL ES APIs support several ways of specifying the primitives to draw, that is, different draw modes. The primitives can be specified individually or as a connected strip or fan. They can also be non-indexed, meaning that vertices are passed in a vertex array and processed in order, or indexed, meaning that vertices are passed as indices into a vertex array.

Early-Z
A Z-testing scheme that performs the actual Z-test before texturing or fragment shading when it is safe to do so, increasing performance and reducing the required bandwidth.

EGL driver
See Native platform graphics interface.

ESSL
See OpenGL ES Shading Language.

Ericsson Texture Compression (ETC)
A 4 *bit-per-pixel* (bpp) texture compression algorithm.

Fixed-function pipeline
A process that uses standard functions to draw graphics on fixed-function graphics hardware. For example, OpenGL ES 1.1 implements a fixed-function pipeline.

Fragment
A fragment consists of all data, such as depth, stencil, texture, and color information, required to generate a pixel in the framebuffer. A pixel is usually composed of several fragments. A fragment can either be multi-sampled or super-sampled.

Fragment shader
A program running on the pixel processor that calculates the color and other characteristics of each fragment.

Framebuffer
A memory buffer containing a complete frame of data.

Geometry processor
A geometry processor executes vertex shaders that typically contain transform and lighting calculations, and generates lists of primitives for a pixel processor to draw.

Graphic application
A custom program that executes in the Mali graphics system and displays graphics content.

Graphics driver
A software library implementing OpenGL ES or OpenVG, using graphics accelerator hardware. See also OpenGL ES driver and OpenVG driver.

Graphics pipeline
The series of functions, in logical order, that must be performed to compute and display computer graphics.

Instrumented drivers
Alternative graphics drivers that are used with the Mali GPU. The Instrumented drivers include additional functionality such as error logging and recording performance data files for use by the Performance Analysis Tool.

Mali Demo Engine
The Mali Demo Engine is a component of the Mali Developer Tools. The Mali Demo Engine library enables you to develop 3D graphics applications more easily than using OpenGL ES alone.

Mali Binary Asset Exporter
A converter tool for Windows that converts XML-based COLLADA documents to the Mali Binary Asset format for use with the Mali Demo Engine. The Mali Binary Asset Exporter is a component of the Mali Developer Tools.

Mali Demo Engine Library
A C++ class framework for developing OpenGL ES 2.0 applications for the Mali GPU.

Mipmap
A pre-calculated, optimized collection of bitmap images that accompanies a main texture, intended to increase rendering speed and reduce artifacts.

Multi-ICE
A JTAG-based tool for debugging embedded systems.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Multi-sampling
An anti-aliasing technique where each pixel in the framebuffer is split into multiple samples corresponding to different positions within the area covered by the pixel. Each fragment produced for the pixel is duplicated onto each sample, and operations such as alpha-blending and depth testing is performed on a per-sample basis. In the final image, the color of each pixel is the average between the colors of the samples for that pixel.

The Mali pixel processors support multi-sampling at four samples per pixel with negligible performance impact.

Native platform graphics interface (EGL) driver
A standardized set of functions that communicate between graphics software, such as OpenGL ES or OpenVG drivers, and the platform-specific window system that displays the image.

Offline Shader Compiler
A command line tool that translates vertex shaders and fragment shaders written in the ESSL into binary vertex shaders and binary fragment shaders that you can link and run on the GPU.

OpenGL ES driver
On graphics systems that use the OpenGL ES API, the OpenGL ES driver is a specialized driver that controls the graphics hardware.

OpenGL ES Shading Language (ESSL)
A programming language used to create custom shader programs that can be used within a programmable pipeline, on graphics hardware. You can also use pre-defined library shaders, written in ESSL.

OpenVG driver
On graphics systems that use the OpenVG API, the OpenVG driver is a specialized driver that controls the graphics hardware.

Performance Analysis Tool
A fully-customizable GUI tool that displays and analyzes performance data files produced by the Instrumented drivers, together with framebuffer information.

See also Instrumented drivers, Performance data file.

Performance counter
Data produced by the Instrumented drivers and the GPU hardware, that can be displayed and analyzed as statistical information in the Performance Analysis Tool.

Performance data file
Files that contain a description of the performance counters, together with the performance counter data in the form of a series of values and images. Performance data files are saved in `.ds2` format and can be loaded directly into the Performance Analysis Tool.

Pixel
A pixel is a discrete element that forms part of an image on a display. The word pixel is derived from the term Picture Element.

Pixel processor
A pixel processor, such as the Mali-200 or Mali-400, performs rendering operations to produce a final image for display.

Polygon list
A list of triangles produced by the geometry processor, containing data that describes triangles that have been transformed. The list is input to the PLB.

Polygon List Builder (PLB)
The PLB assembles transformed vertices into a data structure containing data that describes the triangles. The data structure is ordered into tiles. A tile is an approximately square area, covering a number of pixels on the display screen. The fragment processor processes the data structure.

Primitive
A basic element that is used, with other primitives, to generate images. A primitive can be a point, a line, or a triangle. Each primitive is divided into fragments so that there is one fragment for each pixel covered by the primitive.

Programmable pipeline
A process that uses custom programs to draw graphics on programmable graphics hardware. For example, OpenGL ES 2.0 implements a programmable pipeline.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access

Quad
A rendering primitive with four vertices.

Rasterization
The process of identifying the fragment of each triangle that is seen through each pixel on the display screen. The pixel processor performs rasterization.

Sample
A sample refers to a value or set of values at a point in space. The defining point of a sample is that it is a chosen value out of a continuous signal. In the context of graphics, the sample point is usually in the middle of a pixel, and what is sampled is the geometry descriptions of polygons.

Scissoring
A process that prevents rendering in certain portions of a rendering surface.

Shader
A program, usually an application program, running on the GPU, that calculates some aspect of the graphical output. See fragment shader and vertex shader.

Shading language
A programming language used to define custom shader programs to run on programmable graphics hardware. Different graphics APIs support different shading languages.

SoC
System-on-Chip.

Sub-pixel
Full-color displays are made by combining red, green, and blue light in varying degrees to produce different shades of colors. In a display with a fixed pixel structure, such as LCDs or plasma panels, the red, green, and blue light comes from adjacent cells in the display's physical structure. The light from these three subpixels, one for each color, combine to create a single pixel. There are also pixel structures that do not rely on three subpixels.

Super-sampling
An anti-aliasing technique where the image is rendered in a higher resolution than the framebuffer and then scaled down before being written to the framebuffer.

Texture Compression tool
A component of the Mali Developer Tools that you can use to compress textures and images, using the ETC algorithm.

Tile buffer
A memory buffer inside the GPU that holds the framebuffer contents for the tile that is currently being rendered. The tile buffer can be accessed without using the memory bus.

Vertex
A set of data defining the properties of one point of a primitive. For example, a point primitive, an endpoint of a line primitive, or a corner of a triangle primitive.

Vertex shader
A program running on the geometry processor, that calculates the position and other characteristics, such as color and texture coordinates, for each vertex.

Copyright © 2007-2009 ARM. All rights reserved.
Non-Confidential, Unrestricted Access