Mali™ GPU OpenVG

Application Development Guide

ARM®

ARM DUI 0380D (ID121709)

Mali GPU OpenVG
Application Development Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
6 December 2007	A	Non-Confidential	First Release
29 May 2008	B	Non-Confidential	Second Release
30 September 2008	C	Non-Confidential	Update for improvements to OpenVG driver
15 December 2009	D	Non-Confidential	Update for Mali Developer portal

Proprietary Notice

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 OpenVG Application Development Guide

Preface

This preface introduces the *Mali Graphics Processing Unit (GPU) OpenVG Application Development Guide*. It contains the following sections:

About this guide

This is the *OpenVG Application Development Guide* (ADG) for the Mali GPU. It provides an introduction to the OpenVG graphics standard, and guidelines for using the OpenVG API to develop applications for a Mali GPU.

This document applies to the Mali GPU range. Any differences for particular GPUs are clearly indicated. The document describes how to achieve optimal use of the hardware and software.

Use the Application Development Guide in conjunction with the other Mali GPU documentation. See *Additional reading* on page vi, for a list of the other available documentation.

Intended audience

This guide is written for system designers, system integrators, and programmers who are designing or programming a *System-on-Chip* (SoC) that uses the Mali GPU. They are familiar with graphics programming and the OpenVG graphics standard.

Using this guide

This guide is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to developing OpenVG applications on the Mali GPUs.

Chapter 2 *Developing OpenVG Applications*

Read this chapter for information about how to develop efficient applications for the Mali GPU, using OpenVG.

Chapter 3 *Optimizing OpenVG Applications*

Read this chapter for information about optimizing performance when developing your OpenVG application.

Glossary Read this for definitions of terms used in this book

Conventions

Conventions that this guide can use are described in:

- *Typographical*.

Typographical

The typographical conventions 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.
`mono`space	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.

Additional reading

This section lists publications by ARM and by third parties.

See http://infocenter.arm.com/help/index.jsp for access to ARM documentation.

ARM publications

This guide contains information that is specific to this product. See the following documents for other relevant information about the Mali Development Tools:

- *Mali GPU Performance Analysis Tool User Guide* (ARM DUI 0502)
- *Mali GPU Texture Compression Tool User Guide* (ARM DUI 0503).

Other publications

This section lists relevant documents published by third parties:

- *OpenVG 1.1 Specification* - http://www.khronos.org
- *EGL 1.4 Specification* - http://www.khronos.org.

Note

The Mali-55 GPU supports the following earlier versions of the standards:

- *OpenVG 1.0.1 Specification* - http://www.khronos.org
- *EGL 1.3 Specification* - http://www.khronos.org.

Feedback

ARM welcomes feedback on the Mali 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 if appropriate.

Feedback on this guide

If you have any comments on this guide, send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0380D
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter provides information about developing applications for the Mali GPU, and contains the following sections:

1.1 Mali System Overview

The Mali GPU forms the basis of a high performance graphics processing solution. When implemented as part of an SoC device, the GPU forms an integral part of the graphics solution. The Mali GPU hardware consists of either programmable hardware or fixed-function hardware.

Programmable hardware typically includes a programmable pixel processor, and a programmable geometry processor. The geometry processor performs all geometric and vertex processing and passes this information, as data structures, to the pixel processor. The pixel processor performs 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 using a fixed-functionality pipeline. All geometric operations are preformed by software running on the CPU.

Mali GPUs accelerate most features of the OpenVG graphics standard, particularly:

- path rasterization
- paint generation, filling of path-interior, and strokes
- image rendering
- blending
- anti-aliasing
- scissoring
- masking
- stroke generation
- transformations
- color conversion
- image filtering.

1.2 Graphics standards

The Mali GPU supports the OpenVG API. OpenVG contains 2D functionality for hardware accelerated vector and raster graphics. It provides a device-independent interface for 2D graphical applications, enabling you to add hardware acceleration to devices ranging from mobile phones to desktop PCs.

The Mali GPU also supports the OpenGL ES API for 3D functionality. See the *Mali GPU OpenGL ES Application Development Guide*.

The Mali GPU uses the OpenVG API to provide a low-level interface between graphics software and hardware graphics acceleration.

Specifically, the Mali GPU implements:

OpenVG OpenVG is an API for hardware-accelerated 2D vector and raster graphics. It is specifically intended to support all drawing features required by a *Scalable Vector Graphics* (SVG) Tiny 1.2 renderer, and additionally to support functions that you can use for implementing an SVG Basic renderer.

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

The Mali GPU OpenVG driver and the EGL interface are implementations of these standards that control the graphics hardware.

See `http://www.khronos.org` for more information about these graphics standards.

1.3 Mali GPU Developer Tools

The Mali Developer Tools provide the following tools to help you optimize OpenVG graphics application performance:

- *Mali GPU Performance Analysis Tool*
- *Mali GPU Texture Compression Tool*.

1.3.1 Mali GPU Performance Analysis Tool

The Performance Analysis Tool helps you to analyze graphics application performance, by studying hardware and software performance counters, produced by the Mali Instrumented drivers.

The Instrumented drivers are alternatives to the standard drivers that you can use with the GPU to log performance data, error messages and framebuffer output. The drivers can create logs with more informative error messages than the OpenVG `vgGetError` function can retrieve. This helps find the precise function that generates an error, the function parameter that generates the error, and why the error is generated.

Instrumented drivers are provided as part of a *Board Support Package* (BSP) for the hardware platform you are developing.

The Performance Analysis Tool can display scene statistics and hardware performance counter values for each frame.

You can use the Performance Analysis Tool together with standard ARM development tools such as the *RealView® Developer Suite* (RVDS). See `http://www.arm.com` for a complete list of ARM software development tools.

Note

Many of these tools are semihosted. This means that you can operate them from your desktop computer using, for example, the ARM *RealView ICE®* or *RealView Multi-ICE®* units for communication with the development platform.

For more information about the Performance Analysis Tool and the Instrumented drivers, see the *Mali GPU Performance Analysis Tool User Guide.*

1.3.2 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 uncompressed 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.

Chapter 2
Developing OpenVG Applications

This chapter describes how to develop efficient applications for the Mali GPU, using the OpenVG API. It contains information about the development stages, and how to optimize your applications.

This chapter contains the following sections:

2.1 Developing applications

The different approaches to developing a graphics application for embedded devices depend on various hardware and software tools, such as compilers, graphics drivers, debuggers, and communications facilities. You can use any of the following approaches to develop OpenVG 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 suitable ARM reference platform for the Mali GPU, ARM recommends you use the GNU *gcc* build tool with reference platform. The reference platform helps you to evaluate the Mali GPU and design graphics software for different operating systems. See `http://www.arm.com` for more information.
- Start embedded application development on a desktop PC using the Khronos Reference Implementation. See `http://www.khronos.org` for more information about obtaining the Khronos Reference Implementation.
- Start embedded application development on a desktop PC using a desktop implementation of OpenVG that is OpenVG conformant.

2.2 Optimizing application speed

The execution speed of an OpenVG application is typically limited by a bottleneck somewhere in the processing pipeline. Operations that typically cause bottlenecks in an OpenVG application are:

- triangle geometry generation for path rendering
- calling CPU intensive API functions
- large data transfers
- operations causing a pipeline flush
- draw call overhead.

Chapter 3 *Optimizing OpenVG Applications* describes these operations in the context of the OpenVG API functionality.

In addition to these operations, high levels of geometry processing load and fragment processor load can cause bottlenecks.

You can use some of the Mali Developer Tools to identify bottlenecks. The following tools can help you to improve the efficiency of your application code:

- the Performance Analysis Tool enables you to view and analyze performance data from the GPU hardware, and the OpenVG software drivers
- the Mali Instrumented Drivers available as part of the BSP, enable you to collect performance data.

These tools are described in the *Mali GPU Performance Analysis Tool User Guide*.

Chapter 3
Optimizing OpenVG Applications

This chapter describes how to optimize OpenVG applications. It contains the following sections:

3.1 Identifying Bottlenecks

To identify bottlenecks in your software use the instrumented drivers together with the Performance Analysis Tool.

Typical bottlenecks can be classified into:

- Software bottlenecks
- Hardware bottlenecks.

To locate bottlenecks, run the application using the instrumented drivers and enable dumping of hardware and software counters. Use the Performance Analysis Tool to inspect the dumped Mali hardware and software counters.

3.1.1 Software counters

Software counters give a picture of CPU load. The following counters are most useful:

VG API calls

The **VG API calls** counter indicates the time spent in each VG function per frame.

VG Path Stats

The **VG Path Stats** counter shows details of scene complexity.

Note

To see a detailed description of a counter, hover the mouse pointer over the counter.

3.2 Path rendering

If `vgDrawPath` is taking too much time to perform path rendering, a number of steps can remedy this:

- *Reduce number of draw calls*
- *Optimize paths for overdraw*
- *Use efficient path segments*
- *Render complex paths to images*
- *Use the path cache*
- *Set correct path capabilities* on page 3-4.

3.2.1 Reduce number of draw calls

If you are drawing many different paths with the same transform and same style, it pays off to append all these paths onto a single path and draw them all using one `vgDrawPath`.

3.2.2 Optimize paths for overdraw

2D vector graphics tend to be designed in a back to front fashion resulting in a lot of overdraw where many of the drawn pixels are not visible in the final image. It pays off to limit overlapping regions in the path data set. This can however come at a cost if you have to add segments to your geometry to avoid overlaps. Reducing overlap needs to be balanced against the cost introduced by a higher segment count.

3.2.3 Use efficient path segments

Prefer simpler path segments over high order path segments where possible to limit SW processing time. On the Mali-200 GPU, using quadratic curves instead of arcs and cubics can be more efficient for stroked curves. The choice of segment types must be balanced against the cost of introducing new segments.

Segment types sorted in ascending order according to processing cost:

- lines
- quadratics
- cubics
- arcs.

3.2.4 Render complex paths to images

Complex paths can be cached in images if the overhead of drawing them using `vgDrawPath` becomes too high.

You lose the scaling benefits and proper anti-aliasing of path. However, if this is not required it can be an option to use a cached path image for higher performance. Bind a `VGImage` to a `pbuffer` and render geometry into it. Draw it using `vgDrawImage`.

3.2.5 Use the path cache

There are different path caches for the Mali-200 and Mali-400 GPUs and the entry-level Mali-55 GPU.

Mali-200 and Mali-400 GPU Path cache

When a path is first drawn a processed copy of it is stored in a path cache in the driver. When drawing the path again, this efficient copy is used in place of the general path.

The cached paths might be removed from the cache if the path is significantly modified by a VG function or the cache mechanism finds it more efficient to cache a different path in its place.

A cached stroke must be regenerated if its stroking parameters change.

Mali-55 GPU path cache

The Mali-55 GPU path cache stores the tessellated triangle mesh resulting from tessellation per path. This mesh can be reused in following draw calls for more efficient drawing. The mesh is invalidated and a new one is generated if:

- the path is modified by a VG function
- the path is drawn at a larger scale than what is cached
- the fill rule changes
- the stroke parameters change.

3.2.6 Set correct path capabilities

Specify what your paths are going to be used for. If `VG_PATH_CAPABILITY_*_FROM` or `VG_PATH_CAPABILITY_MODIFY` are not required, the driver can conserve memory as it can throw away the original input data.

3.3 Image rendering

This section contains the following suggestions on how to optimize image rendering performance:

- *Use the fastest image modes*
- *Use the fastest image formats*
- *Avoid image modification*
- *Avoid unnecessary render states*
- *Avoid operations causing a pipeline flush*.

3.3.1 Use the fastest image modes

To do this:

- use the image mode `VG_IMAGE_MODE_NORMAL`
- avoid `MULTIPLY` and `STENCIL` if the effects of these are not visible in the current draw call. These operations require paint evaluation, even if there is no visible effect.

3.3.2 Use the fastest image formats

To do this:

- use premultiplied `sRGB` formats, because this saves the graphics pipeline from having to do conversions
- use low bit depth image formats, because theses are easier to cache and therefore have higher performance.

3.3.3 Avoid image modification

Modification of an image drawn previously in the same frame causes the driver to make a copy of the previously drawn image. This is best avoided as it is a slow process and increases memory usage.

3.3.4 Avoid unnecessary render states

Setting global and object state involves input validation, modification and possibly memory allocation. The performance penalty is relatively small compared to draw functions however these operations are not completely free.

3.3.5 Avoid operations causing a pipeline flush

Some operations cause all hardware queued jobs to be flushed. When this happens the CPU stalls and waits before the pixels can be read for the rendering result to be read back. This process is slow so it is best avoided. For more information see *Flushing the pipeline* on page 3-9.

3.4 Paint generation

When you combine your image and path drawing with paints, the type of paint you use can affect performance. You can achieve more efficient code if you use the fastest suitable paint type. The following lists of different paint types are sorted by speed for each GPU, with the fastest paint type first:

- Color paint
- Linear gradient paint
- Pattern paint
- Radial gradient paint.

Note

On the Mali-55, pattern paint and radial gradient paint are slower than on the Mali-200 or Mali-400 because the Mali-55 does not contain a geometry processor.

For more information about these paint types, see the *OpenVG 1.0.1 Specification* or *OpenVG 1.1 Specification*.

3.5 Masking and scissoring

This section contains information about how you can optimize masking and scissoring performance.

- Do not leave masking enabled even if the current mask is all 1s. It does not affect the drawing operations. Masking is expensive and consumes memory.

  On the Mali-55 GPU, masking is best avoided if it is not necessary.

- Do not leave scissoring turned on if no scissoring is required. Specifying a full screen scissor rectangle is slower than disabling scissoring.

- When using the OpenVG API on the Mali-200 GPU and Mali-400 GPU you can issue a function call to instruct the driver how you want the scissoring to be set up. Based on this information, the driver decides if it can use the hardware scissor which is present in Mali, or if it needs to use a combination of Mali features.

  Note

  Using the one scissor rectangle provides best performance because you use a dedicated piece in the Mali-200 GPU or Mali-400 GPU hardware pipeline.

3.6 Image filtering

This section contains information about how you can optimize image filtering performance.

To improve the efficiency of image filtering:

- minimize image and kernel sizes. Convolution filtering speed is usually proportional to image size and kernel size, for example, a 3x3 kernel is faster than a 7x7 kernel
- minimize standard deviation in blurs

  use the iterative box or average blurs by using `vgSeparableconvolve` instead of `vgGaussianBlur`
- do not use 1-bit or 4-bit images as destination images, because Mali-200 and Mali-400 cannot render to these formats natively.

Note

- The Mali-55 GPU uses image filtering performed in software, filtering is therefore slow.
- The Mali-200 and Mali-400 filtering runs concurrently with other OpenVG calls. To use this for a greater effect, issue the filtering calls as early as possible because the `eglSwapBuffers()` function stalls if the filtering operations are not completed. The driver also stalls if the destination image is changed, for example by the `vgImageSubData` or `vgImageClear` functions, after it has been used as the filtering destination.

3.7 Flushing the pipeline

Flushing the pipeline is the same as rendering the frame, this takes time and must be kept to a minimum. One pipeline flush is required each frame to see the result of the rendering.

3.7.1 Operations that cause a pipeline flush

Some OpenVG functions require a pipeline flush. That is, these functions require an intermediate framebuffer result to be completed. Normally, the renderer can work while the application and driver prepare the next frame of animation. However, if the result from the renderer is required to complete an operation, the driver stalls until the intermediate result has been generated.

The following functions require a pipeline flush:

- `vgCopyPixels`
- `vgGetPixels`
- `vgReadPixels`.

Reading back the contents of the surface into an array causes a pipeline flush because the pixels have to be resolved. It might appear that reading from the surface is a simple, quick `memcpy` however, because it causes a flush the operation is slow.

If you are developing interactive applications, you might prefer to avoid using these functions. For example, an alternative to using `vgCopyPixels` is to redraw the geometry at the new location and clip it with the scissor boxes. However, the method you use is dependent on the application you are developing. If you are drawing a large amount of geometry, it might still be cheaper to use the functions that require a pipeline flush. In general, use the Performance Analysis Tool to analyze your application to help you decide which method to use.

Note

On the Mali-55 GPU, masking causes a pipeline flush.

3.7.2 Do not use vgFinish to finalize a frame

`eglSwapBuffers` starts the rendering process, when the frame is done, it is displayed in a window. This is necessary to see the result of the rendering.

Calling `vgFinish` causes rendering to start and the application stalls until rendering is complete, however nothing is displayed. Do not use this call as it wastes processor time.

Some applications call `vgFinish` before `eglSwapBuffers`, this is unnecessary and reduces performance.

3.8 Functions implemented in software

OpenVG functions that initialize VG states and objects, or do not generate any pixel data, are not accelerated by the Mali GPU, but are implemented in software running on the main CPU.

The general software functions are:

- functions that get and set values, for example `vgGetPaint` and `vgSetColor`
- functions that create and destroy objects, for example `vgCreatePaint` and `vgDestroyImage`.

The software path operation functions are:

- `vgAppendPath`
- `vgAppendPathData`
- `vgClearPath`
- `vgInterpolatePath`
- `vgModifyPathCoords`
- `vgPointAlongPath`
- `vgPathLength`
- `vgPathBounds`
- `vgPathTransformedBounds.`

With the exception of `vgPointAlongPath` and `vgPathLength`, these functions use only a few CPU cycles.The functions `vgPointAlongPath` and `vgPathLength` cache their results, so that successive calls to these functions are fast, although the first call per path uses more CPU cycles than subsequent calls.

The software image functions are:

- `vgChildImage`
- `vgClearImage`
- `vgCopyImage`
- `vgGetImageSubData`
- `vgGetParent`.

The `vgChildImage` and `vgGetParent` functions use only a few CPU cycles. The CPU cycle cost for calling the remaining functions is proportional to the size of the image data transfers made during the operation.

In addition, on fixed-function hardware without a dedicated vertex processor, the following functions are also implemented in software:

- `vgConvolve`
- `vgSeparableConvolve`
- `vgColorMatrix`
- `vgGaussianBlur`
- `vgLookup`.
- `vgLookupSingle`.

3.9 Anti-aliasing

Anti-aliasing is used to prevent aliasing noise, or *jaggies* in images. using this technique creates bettering, smoother images.

Traditionally, anti-aliasing has a high performance penalty, however the Mali GPUs can do 4x anti-aliasing with almost no penalty.

The quality of Anti-aliasing is controlled through the setting `VG_RENDERING_QUALITY`, this has the following options:

`VG_RENDERING_QUALITY_NONANTIALIASED`

No Anti-aliasing

`VG_RENDERING_QUALITY_BETTER`

4X Anti-aliasing

`VG_RENDERING_QUALITY_FASTER`

4X Anti-aliasing

3.10 Multi-threading

This section contains information about how you can optimize the use of OpenVG in a multi-threaded application.

Avoid having more than one thread accessing a single context simultaneously. If you must have more than one thread accessing a single context, you must implement some form of synchronization.

You can handle the synchronization using one of the following methods:

- Wait until the OpenVG context is available. This is not optimal for the graphics application, because other processes or threads might get a time slice instead. However, this method might be beneficial for the system as a whole.
- Perform some other task while waiting for the OpenVG context to become available.
- Use only one OpenVG context per thread. By doing so, you are not required to perform any synchronization. If you use only one context per thread, you can still share path, paint, and image objects between multiple contexts. That is, there is still one context per thread, but there can be many threads, each containing one context.

  For example, you can share paint, images, and paths between contexts as follows:

  1. Create a context A, using `EGLCreateContext`.
  2. Create a context B, using `EGLCreateContext`. passing context A as a shared context parameter.

  This procedure ensures that all paths, images and paints are available in both contexts. In this situation it is safe to bind context A to thread A and context B to thread B, and use the contexts without performing synchronization.

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 (API)
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.

EGL driver *See* Native platform graphics interface.

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.

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.

Graphics 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.

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.

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.

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.

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.

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 performs rendering operations to produce a final image for display.

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.

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

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

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.

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.