

Mali™ GPU User Interface Engine

Version: 2.3

User Guide



Mali GPU User Interface Engine

User Guide

Copyright © 2009-2010 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
14 October 2009	A	Non-Confidential	First release for v2.2
26 January 2010	B	Non-Confidential	Updates for v2.3 release, document name changed from Mali GPU Demo Engine User Guide.

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 User Interface Engine User Guide

	Preface	
	About this book	viii
	Feedback	x
Chapter 1	Introduction	
	1.1 About the Mali GPU User Interface Engine	1-2
Chapter 2	Installing the User Interface Engine	
	2.1 UI Engine contents	2-2
	2.2 Installing the UI Engine on Microsoft Windows	2-3
	2.3 Installing the UI Engine on Linux	2-4
Chapter 3	Building the User Interface Engine	
	3.1 Building the UI Engine on Microsoft Windows	3-2
	3.2 Running examples on Windows	3-3
	3.3 Building the UI Engine on Red Hat Linux	3-5
	3.4 Running examples on Red Hat Linux	3-6
	3.5 Building the UI Engine for ARM Embedded Linux	3-7
	3.6 Running examples on ARM Embedded Linux	3-8
Chapter 4	Using the User Interface Engine	
	4.1 Using the UI Engine Library components	4-2
	4.2 Using the Mali Binary Asset Format	4-10
	4.3 UI Engine tutorial examples	4-11
Chapter 5	Example Shaders	
	5.1 About the example shaders	5-2
	5.2 Example shaders	5-3

Glossary

List of Tables

Mali GPU User Interface Engine User Guide

	Change history	ii
Table 3-1	Example directories	3-3

List of Figures

Mali GPU User Interface Engine User Guide

Figure 3-1	Test suite	3-4
Figure 3-2	Texturing example	3-4
Figure 5-1	Simple shader example	5-3
Figure 5-2	Coloring example	5-4
Figure 5-3	Texturing example	5-4
Figure 5-4	Gouraud and Phong lighting examples	5-5
Figure 5-5	Bump mapping before and after examples	5-6
Figure 5-6	Example spherical environment mapping	5-7
Figure 5-7	Example cube map environment	5-8
Figure 5-8	Using environment reflection mapping	5-8

Preface

This preface introduces the *Mali GPU User Interface Engine User Guide*. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page x.

About this book

This is the *Mali GPU User Interface Engine User Guide*. It provides guidelines for using the Mali Developer Tools to assist in the development of applications for OpenGL ES 3D graphics applications. This document was previously known as the Mali GPU Demo Engine user Guide, it is part of a documentation suite for the Mali Developer Tools.

Intended audience

This guide is written for system integrators and software developers who are writing OpenGL ES applications using the Windows XP or Linux operating system, and want to progress onto writing applications for the Mali GPU range in the future.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

Read this for an introduction to the *User Interface* (UI) Engine.

Chapter 2 Installing the User Interface Engine

Read this chapter for information on how to install the UI Engine.

Chapter 3 Building the User Interface Engine

Read this chapter for information on how to build the UI Engine.

Chapter 4 Using the User Interface Engine

Read this for information on how to use the UI Engine Library, a bundled C++ class framework for developing OpenGL ES 2.0 applications.

Chapter 5 Example Shaders

Read this for information on examples of shaders that can be used with the Mali UI Engine.

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

Typographical Conventions

The typographical conventions are:

<i>italic</i>	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.
<u>monospace</u>	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<i>monospace italic</i>	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.

ARM publications

This guide contains information that is specific to the Mali Developer Tools. See the following documents for other relevant information:

- *Mali GPU Developer Tools Technical Overview* (ARM DUI 501)
- *Mali GPU Performance Analysis Tool User Guide* (ARM DUI 0502)
- *Mali GPU Texture Compression Tool User Guide* (ARM DUI 0503)
- *Mali GPU Shader Developer Studio User Guide* (ARM DUI 0504)
- *OpenGL ES 1.1 Emulator User Guide* (ARM DUI 0506)
- *Mali GPU Binary Asset Exporter User Guide* (ARM DUI 0507)
- *Mali GPU Shader Library User Guide* (ARM DUI 0510)
- *OpenGL ES 2.0 Emulator User Guide* (ARM DUI 0511)
- *Mali GPU Offline Shader Compiler User Guide* (ARM DUI 0513).

Other publications

This section lists relevant documents published by third parties:

- *OpenGL ES 1.1 Specification* at <http://www.khronos.org>.
- *OpenGL ES 2.0 Specification* at <http://www.khronos.org>.
- *OpenGL ES Shading Language Specification* at <http://www.khronos.org>.
- *OpenVG 1.1 Specification* at <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.

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 mali.developers@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 0505B
- the page numbers 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 the Mali GPU User Interface Engine, and describes how to start using it in your particular workflow. It contains the following section:

- *About the Mali GPU User Interface Engine* on page 1-2.

1.1 About the Mali GPU User Interface Engine

The Mali GPU UI Engine is a library of C++ functions that can be helpful when building OpenGL ES 2.0 applications for a platform with a Mali GPU. You can use it for creating new applications, training, and exploration of implementation possibilities.

The UI Engine runs on the following platforms:

- Microsoft Windows XP Professional Version 2002, service pack 2
- Red Hat Enterprise Linux, Release 4
- *ARM Embedded Linux* (AEL) with kernel version 2.6.24-arm2 on a Real View ARM11 MPCore board with a Mali-200 r0p5 GPU in a Virtex5 FPGA.

———— **Note** —————

AEL is only supported for running the Mali UI Engine. For building applications you must use Red Hat Enterprise Linux on a Linux workstation then deploy the application to the AEL system.

Chapter 2

Installing the User Interface Engine

This section describes how to install the UI Engine. It contains the following sections.

- *UI Engine contents* on page 2-2
- *Installing the UI Engine on Microsoft Windows* on page 2-3
- *Installing the UI Engine on Linux* on page 2-4.

2.1 UI Engine contents

The UI Engine bundle contains the following software:

- C++ source code for the UI Engine library
- Build files:
 - MSVC projects for Microsoft Windows
 - Makefiles for Linux
- Example applications and shaders.

For more information see the *User Interface Engine Release Note*.

2.2 Installing the UI Engine on Microsoft Windows

This section describes how to install the UI Engine on Microsoft Windows. It contains the following sections:

- *Installation requirements for the UI Engine on Microsoft Windows*
- *Installation procedure for the UI Engine on Microsoft Windows.*

2.2.1 Installation requirements for the UI Engine on Microsoft Windows

To install the UI Engine on Microsoft Windows, you require:

- Microsoft Windows XP Professional Version 2002, service pack 2
- Microsoft Visual Studio 2005
- The Microsoft .NET v2.0 framework
- A minimum of 113MB disk space to install the UI Engine library and applications. Building the UI Engine and applications requires approximately 180MB additional disk space.
- OpenGL 2.x compliant drivers

To be able to run the UI Engine examples on a desktop workstation you must have OpenGL2.x complaint drivers installed for the Graphics chipset in your system.

You must also have the libGLESv2 and libEGL Open GL ES 2.0 emulator libraries. These are available for download from the Mali Developer Center website at:

<http://www.malideveloper.com>

Note

The Mali UI Engine has been tested successfully on a 32-bit version of Microsoft Windows XP Professional edition.

2.2.2 Installation procedure for the UI Engine on Microsoft Windows

The procedure to install the UI Engine on Microsoft Windows is:

1. Go to the Mali Developer Center website at:
<http://www.malideveloper.com>
2. Download the UI Engine package.
3. Run the file `Mali_GPU_User_Interface_Engine_WinXP_vm.n.msi` by double clicking.
where:
m identifies the major version
n identifies the minor version.
4. Select the required installation options and then click **Finish** to complete the installation.

By default, the UI Engine is installed in:

`C:\Documents and Settings\%USERNAME%\My Documents\ARM Mali Developer Tools\Mali GPU UI Engine vm.n`

Where `%USERNAME%` is your Microsoft Windows user name.

2.3 Installing the UI Engine on Linux

This section describes how to install the UI Engine on Linux. It contains the following sections:

- *Installation requirements for the UI Engine on Linux*
- *Installation procedure for the UI Engine on Linux.*

Note

The UI Engine has been tested successfully on a 32-bit version of Linux OS.

2.3.1 Installation requirements for the UI Engine on Linux

This section describes the requirements for installing the UI Engine on a Linux System. It contains the following sections:

- *Supported Linux distributions*
- *Requirements.*

Supported Linux distributions

The following Linux distributions are supported:

- Red Hat Enterprise Linux, Release 4
- ARM Embedded Linux with kernel version 2.6.24-arm2 on a Real View ARM11 MPCore board with a Mali-200 r0p5 GPU in a Virtex5 FPGA.

Note

AEL is only supported for running the Mali User Interface Engine. For building applications you must install and build on Red Hat Enterprise Linux on a Linux workstation, then deploy the application to the AEL system.

Requirements

The following are required for Linux platforms:

- A minimum of 109MB disk space to install the Mali UI Engine library and applications. Building the UI Engine and applications requires approximately 160MB additional disk space.
- OpenGL 2.x compliant drivers
To be able to run the UI Engine examples on a desktop workstation you must have OpenGL2.x complaint drivers installed for the Graphics chipset in your system.
You must also have the libGLESv2 and libEGL Open GL ES 2.0 emulator libraries for Linux. These are available for download from the Mali Developer Center website.
- OpenGL ES 2.0 drivers
To run the UI Engine examples on the PB11Mpcore Mali-200 GPU hardware platform or any other Mali GPU hardware platform, you require OpenGL ES2.0 drivers for the Mali-200 GPU. The Mali UI Engine examples are tested with Mali OpenGL ES2.0 drivers that are part of the Mali GPU DDK from ARM.
- GNU gcc

For the Mali-200 GPU reference platform:

- GNU Toolchain: Sourcery G++ Lite for ARM GNU/Linux version 2007q1-21. This toolchain can be freely downloaded from the CodeSourcery website at <http://www.codesourcery.com>.

For Red Hat Enterprise Linux:

- GNU gcc compiler version 3.4.6 (Red Hat 3.4.6-8)

- GNU Make

The build system is designed to work with GNU Make has been run and tested on a Linux platform with GNU Make 3.81. Versions equal to or greater than this are expected to work correctly.

2.3.2 Installation procedure for the UI Engine on Linux

To install the UI Engine on Linux:

1. Locate the Mali Developer Center website at:
<http://www.malideveloper.com>
2. Download the following package:
`Mali_GPU_UI_Engine_RHEL4_vm.n.tar.gz`
3. To decompress the file:
 - a. open a command terminal and navigate to the directory where you have downloaded the package
 - b. type the following command:
`tar -zxvf Mali_GPU_UI_Engine_RHEL4_vm.n.tar.gz`

By default, the UI Engine is installed in:

`ARM/Mali_Developer_Tools/Mali_GPU_UI_Engine_vm.n`

Chapter 3

Building the User Interface Engine

This chapter describes how to build the UI Engine library and the accompanying examples. It contains the following sections:

- *Building the UI Engine on Microsoft Windows* on page 3-2
- *Running examples on Windows* on page 3-3
- *Building the UI Engine on Red Hat Linux* on page 3-5
- *Running examples on Red Hat Linux* on page 3-6
- *Building the UI Engine for ARM Embedded Linux* on page 3-7
- *Running examples on ARM Embedded Linux* on page 3-8.

3.1 Building the UI Engine on Microsoft Windows

To build the UI Engine on Microsoft Windows, using Visual Studio 2005:

1. Navigate to the following directory:

C:\Documents and Settings\%USERNAME%\My Documents\ARM Mali Developer Tools\Mali GPU UI Engine *vm.n*

where:

- m*** identifies the major version
- n*** identifies the minor version.

%USERNAME% is the Microsoft Windows user name of the user who installed the UI Engine.

———— **Note** ————

The installer creates a shortcut to this folder in your start menu.

2. Set the environment variable GLES2_LIB_DIR to the directory containing the emulator libraries `libGLESv2.lib` and `libEGL.lib`.

If this environment variable is not set at install time, the installer sets this to the standard install location for the emulator libraries.

———— **Note** ————

To add or set environment variables, select the following entry from the **start** menu:

Start → **Control Panel** → **System** → **Advanced** → **Environmental Variables**

3. Open the file `mali_ui_engine.sln`.

———— **Caution** ————

Do not use a *Universal Naming Convention* (UNC) path to locate this file. UNC paths, for example `\\server\Directory\Filename`, are not supported.

This file contains one Visual Studio project called `mde_library`. This is the UI Engine project. The file also contains the unit tests and one Visual Studio project for each example.

4. Right-click on a project and select **Build** to build a project. Alternatively, select **Build solution** from the Visual Studio **Build** menu to build all projects.

The `mde_library` is saved to one of the following directories, depending on the Visual Studio build configuration you selected:

- `lib\release`
- `lib\debug`

3.2 Running examples on Windows

There are various examples and unit tests that are provided with UI Engine, Table 3-1 shows the directories where these are stored. These folders also contain the MSVC project files used to build the corresponding examples.

Table 3-1 Example directories

Contents	Directory
Example Applications	example_applications\
Mali UI Engine Tutorial Examples	tutorial_examples\
OpenGL ES 2.0 shader examples	shader_library\
Mali UI Engine Unit tests	unit_tests\

To run the examples, first ensure the `libGLESv2.dll` and `libEGL.dll` DLLs are either in the current executable directory or on the system path.

OpenGL 2.0 or above compliant drivers for your graphics card must be installed to run UI Engine examples on a computer running Microsoft Windows.

———— **Note** —————

To add the DLLs to the system path:

1. Display the Control Panel by selecting the following application from the start menu.
Start → Control Panel → System → Advanced → Environmental Variables
2. From the Environmental Variables dialog box select **Edit**.
3. Add the locations of the two DLLs to the Path variable.

To run a UI Engine example:

1. To start an example, or the test suite, double-click on the `mali_ui_engine.sln` solution file. Microsoft Visual Studio starts up.

A screen similar to that shown in Figure 3-1 on page 3-4 is shown. In this example the **03-Texturing** project from the shader examples is already highlighted.

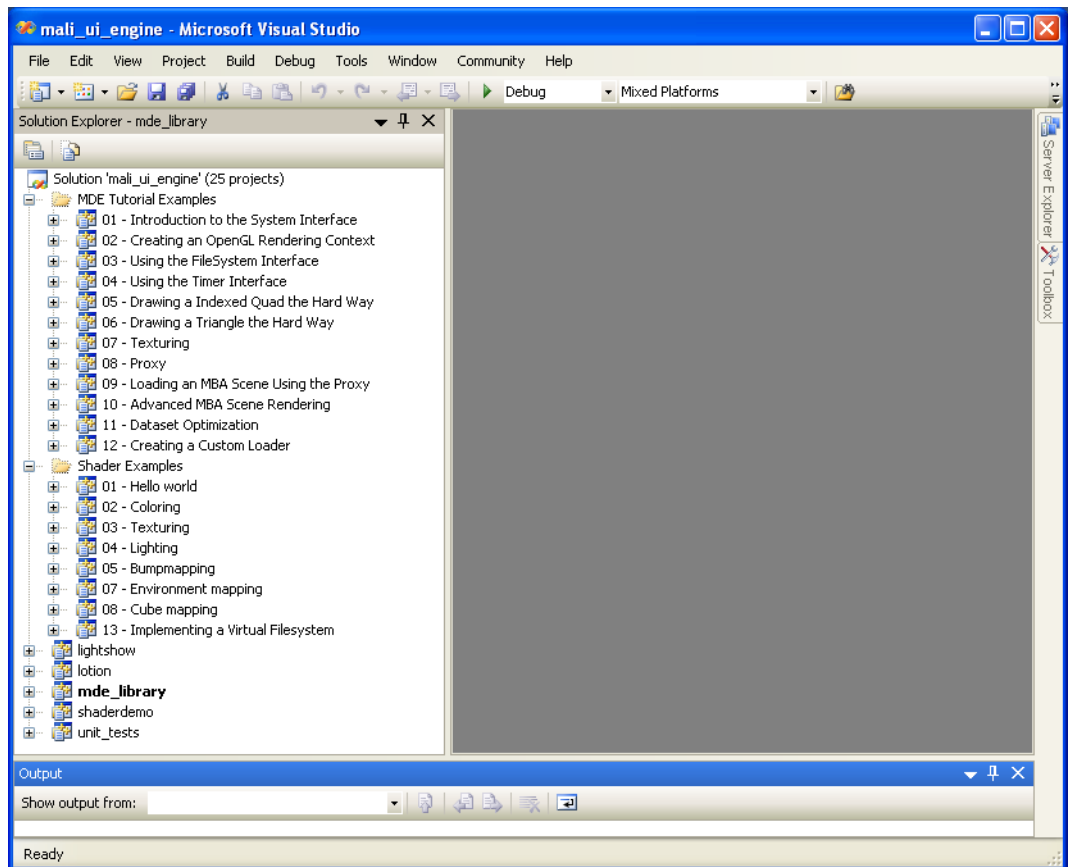


Figure 3-1 Test suite

2. To run a project, select the required project file to highlight it. In this example select **03 - Texturing** from the shader examples.
3. Select **Set as Startup Project** from the **Project** menu.
4. Select **Start without Debugging** from the **Debug** menu.

Figure 3-2 shows the generated texturing example.



Figure 3-2 Texturing example

3.3 Building the UI Engine on Red Hat Linux

To build the UI Engine on Linux, using Red Hat Linux:

1. Open a Linux command prompt.
2. Set the environment variable GLES2_LIB_DIR to the directory containing the OpenGL ES 2.0 emulator libraries for Linux, libGLESv2.so and libEGL.so.
3. `cd ARM/Mali_Developer_Tools/Mali_GPU_UI_Engine_vm.n/`

where:

- m*** identifies the major version
- n*** identifies the minor version.

4. Build the library and various examples by typing the following:
`make all CONFIG=release VARIANT=x11linux-gles20 TOOLCHAIN=gcc`

———— **Note** ————

- If you have previously built the ARM Embedded Linux version in the same directory, before building for Red Hat Linux clean it using the following command:
`make clean`
- To list the supported targets, enter:
`make help CONFIG=release VARIANT=x11linux-gles20 TOOLCHAIN=gcc`

This builds the UI Engine library and various examples. After the build is complete, an executable is created in a directory for each example.

3.4 Running examples on Red Hat Linux

To run the examples:

1. Ensure that you are directly logged into the Linux machine that has the graphics card.
2. Open a Linux command prompt.
3. `cd ARM/Mali_Developer_Tools/Mali_GPU_UI_Engine_vm.n/bin/<name-of-example>`
4. Set the `LD_LIBRARY_PATH` to define the location of the emulator 2.0 libraries for Linux:
 - If you are using a Bourne shell (sh), enter:
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:path_to_libraries`
 - If you are using a C shell (csh), enter:
`setenv LD_LIBRARY_PATH $(LD_LIBRARY_PATH):path_to_libraries`where *path_to_libraries* is the path to the emulator libraries for Linux.
5. Run the example by entering `./<name-of-example>` at the command prompt.

———— **Note** —————

You must run the examples from the directory corresponding to the example. Attempting to run from any other location results in failure.

3.5 Building the UI Engine for ARM Embedded Linux

To build the UI Engine for ARM Embedded Linux:

1. Open a Linux command prompt on a Linux host machine that is set up with ARM GNU tool chain 2007q1-21
2. Set the environment variable GLES2_LIB_DIR to the directory containing the Mali OpenGL ES 2.0 libraries for your target system:
 - libMali.so
 - libGLESv2.so
 - libEGL.so
3. `cd ARM/Mali_Developer_Tools/Mali_GPU_UI_Engine_vm.n/`
4. Build the UI Engine and various examples by typing the following:
`make all CONFIG=release VARIANT=armlinux-gles20 TOOLCHAIN=arm-linux-gcc`
This builds the UI Engine library along with various examples. When the build is complete, an executable is created in each directory corresponding to the example.

———— **Note** ————

- If you have previously built the Red Hat Linux version in the same directory, before building for ARM Embedded Linux clean it using the following command:
`make clean`
 - To list the supported targets, enter:
`make help CONFIG=release`
-

3.6 Running examples on ARM Embedded Linux

Proceed as follows to run the examples:

1. Login to the Mali-200 GPU platform that is set up with ARM Embedded Linux
2. Mount the `Mali_GPU_UI_Engine_vm.n/` directory from the host build machine onto the file system on the Mali-200 GPU platform
3. `cd Mali_GPU_UI_Engine_vm.n/bin/<name-of-example>`
4. Set the `LD_LIBRARY_PATH` to the path to the Mali OpenGL ES 2.0 libraries:
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:path_to_libraries`
5. Run the example by typing `./<name-of-example>` at the command prompt.

———— **Note** —————

You must run the examples from the directory corresponding to the example. Attempting to run from any other location results in failure.

Chapter 4

Using the User Interface Engine

This chapter describes the components of the UI Engine and how to use them. It contains the following sections:

- *Using the UI Engine Library components* on page 4-2
- *Using the Mali Binary Asset Format* on page 4-10
- *UI Engine tutorial examples* on page 4-11.

4.1 Using the UI Engine Library components

This section describes the various components of the UI Engine Library and how to use them. It contains the following sections:

- *The UI Engine Library components*
- *The UI Engine System component* on page 4-3
- *The UI Engine Assets component* on page 4-6
- *The UI Engine Graphics component* on page 4-7
- *The UI Engine Utility classes* on page 4-8.

4.1.1 The UI Engine Library components

The UI Engine Library consists of the following components:

System	The System component contains classes that provide an operating system-independent set of facilities for handling I/O devices such as files and keyboard. See <i>The UI Engine System component</i> on page 4-3.
Assets	<p>The Assets component consists of classes for loading and managing content from Mali Binary Asset files, including lists of vertices and graphics elements, texture formats and shaders. However, you are not restricted to using Mali Binary Asset content. The assets component also enables you to load the following assets without them being associated to any Mali Binary Asset file:</p> <ul style="list-style-type: none"> • shaders • programs • textures • bitmaps • text. <p>See <i>Using the Mali Binary Asset Format</i> on page 4-10 and the <i>Mali GPU Mali Binary Asset Exporter User Guide</i> for more information about the Mali Binary Asset file format. See <i>The UI Engine Assets component</i> on page 4-6.</p>
Graphics	The Graphics component is a set of classes that provides an interface to OpenGL ES 2.0 <i>Application Programming Interface</i> (API)s. The classes are easier to use than the APIs, and they mask some of the differences between, for example, OpenGL 2.0 on various systems and OpenGL ES 2.0. Your application can make calls to the graphics component, the OpenGL ES driver, or a combination of both. The OpenGL ES 2.0 calls within these classes are converted to Open GL calls by the OpenGL ES 2.0 Emulator. See <i>The UI Engine Graphics component</i> on page 4-7.
Vecmath	<p>The Vecmath component is a general utility library that contains vector and matrix arithmetic operations, including:</p> <ul style="list-style-type: none"> • building rotation, translation, scale, lookAt, and perspective transform matrices • generic vector and matrix math operations. <p>See <i>The UI Engine Utility classes</i> on page 4-8.</p>
Templates	The Templates component contains C++ template containers for arrays, maps, strings, and trees. These containers are used as components for the other parts of the Mali UI Engine.

4.1.2 The UI Engine System component

This section describes the UI Engine System component. It contains the following sections:

- *About the UI System component*
- *Creating an OpenGL ES 2.0 rendering context* on page 4-4
- *Using the FileSystem interface* on page 4-4
- *Using input devices* on page 4-5
- *Using the Timer interface* on page 4-5.

About the UI System component

The main part of the UI Engine Library is the System component. The System component handles operating system and platform-dependent tasks, such as:

- utilizing the native file system
- using input devices
- creating an OpenGL ES context.

The System component requires a platform-specific implementation. The UI Engine Library contains implementations for AEL, Windows and Red Hat Enterprise Linux.

To instantiate the System interface, use `MDE::system_createbackend()` as follows:

```
MDE::System* system = MDE::system_create_backend();
...
system->release();
```

———— Note —————

The text and code examples in this section assume that you are using the MDE namespace.

The `system_create_backend()` function returns a System object that you can use to instantiate other objects.

All objects created for you using the System object, follow the UI Engine object model. These objects are reference-counted and the UI Engine Library adds a reference for you.

You own objects created using a `create*` function, and you must call `release` on these objects after you are finished using them. The `release` function decrements the internal reference count of that object and deletes it if no other references exist.

You do not own objects retrieved using a `get*` function, and you therefore have no guarantee about the lifetime of the object. You must not call `release` on these objects.

The UI Engine Library offers a convenient way to release objects when they leave their current scope. This is achieved by encapsulating the object pointer in the `Managed<T>` template as follows:

```
Managed<System> system = system_create_backend();
{
    Managed<Keyboard> keyboard = system->createKeyboard();
}
```

The keyboard is released at the end of the scope, and you cannot use it any more.

For an example that does not use the `Managed<>` template to handle resources, see `01_introduction_to_the_system_interface/main.cpp` in the `tutorial_examples` directory.

Creating an OpenGL ES 2.0 rendering context

Use the System interface to create an OpenGL ES 2.0 context to draw with. Use the update() function on a Context object to make it swap buffers and return one of the following boolean values:

- true if the context is still meant to be drawn
- false if the context is not to be drawn. You can trigger this by closing the OpenGL or OpenGL ES 2.0 window.

Create a context as follows:

```
Managed<Context> context = system->createContext(640, 480);
while(true)
{
    glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    /* Draw content here */

    /* The update() function swaps buffers to refresh the image on the screen.
       The function returns false if the context is told to close by the operating
       system.*/

    if(!context->update()) break;
    if(keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE)) break;
}
```

Using the FileSystem interface

You can use the FileSystem interface to create streams for reading and writing files in the native or a virtual file system. You can use the System::createFileSystem() function to retrieve a FileSystem object.

The following code shows how to use a FileSystem object to create input and output streams:

```
...
Managed<FileSystem> filesystem = system->createFileSystem(".");
Managed<InputStream> input = filesystem->createInputStream("read.txt");
Managed<OutputStream> output = filesystem->createOutputStream("out.txt");
...
```

To read or write to the stream, you must define:

- a buffer to read from or write to
- the size of each element
- how many elements to read or write.

You can define these properties as follows:

```
// Read 10 elements where each element is the size of a char
char* inputBuffer = new char[10];
input->read(inputBuffer, sizeof(char), 10);

// write the content of the input buffer to the output stream
output->write(inputBuffer, sizeof(char), 10);

/* Depending on the underlying implementation, you might have to call the flush()
function to flush the output buffer if it is prudent to write the data to file
immediately. */
output->flush();
```

Using input devices

The Mali UI Engine Library supports keyboard input:

```
...
#ifdef MDE_OS_PC_LINUX
Managed<Keyboard> = system->createKeyboard(context);
#else
Managed<Keyboard> = system->createKeyboard();
#endif
while(true)
{
    // Check key press and break if escape or q is pressed:
    if(keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE)) break;
    if(keyboard->getCharacterKeyState('q')) break;
}
...
```

Note

There are minor difference in the creation of inputs between Windows and X11 platforms. Input devices are linked to specific contexts on X11 systems, for example, to a window on the display that this input device is used with.

The UI Engine Library also supports Mouse input, as follows:

```
...
Managed<Mouse> mouse= system->createMouse();
// Check if left Mouse button clicked
if(mouse->isButtonDown(Mouse::LEFT_BUTTON))
{
    // Get X and Y coordinates
    xClick = mouse->getAxisPosition(Mouse::X_AXIS);
    yClick = mouse->getAxisPosition(Mouse::Y_AXIS);
}
...
```

Event based mouse handling is supported using the *observer* pattern. To use this, create an observer object with its type derived from the `MDE::Observer` class, and call `mouse->attach(observer)`.

When a mouse event occurs, such as a mouse move or mouse button press, the mouse object calls `observer->updateObserver()` with an argument of mouse.

`mouse->getCurrentEvent()` to get the event that called `updateObserver()`.

You can extend the UI Engine Library to enable the use of any input device by creating a new input device class that inherits `InputDevice`.

Using the Timer interface

The Timer interface is an interface for creating timers. The timers can get the time passed since a timer was started or restarted, the *Frames Per Second* (FPS), and the interval.

Use the Timer interface as follows:

```
// Create System and Timer objects:
Managed<System> system = create_system_backend();
Managed<Timer> timer = system->createTimer();
timer->reset();
printf("Time since object creation is:\t%f\n", timer->getTime());
```

4.1.3 The UI Engine Assets component

This section describes the UI Engine Assets component, and describes:

- *Introduction to assets and resource management*
- *The Proxy class.*

Introduction to assets and resource management

An asset is a container class for a resource. Assets can be tangible concepts such as shaders, programs, bitmaps, or textures, or they can be more abstract concepts such as lights, animations, or geometries.

An asset contains resource data specific to the Asset type such as:

- bitmap data
- metadata such as the width and height of a bitmap
- convenience functions for a bitmap such as `flipY()`.

See the UI Engine API documentation for a full list of available asset types and their functions.

The Proxy class

The Mali UI Engine includes a number of asset loaders. Each asset loader is able to load a particular data format, or in some cases a range of data formats, as a particular asset type. For example the JPEG loader loads JPEG data streams as 2D Bitmap assets.

The Proxy class includes functions that allow the application load assets of any supported type without explicitly referencing the loaders. The `getAsset()` function will attempt to load the data in a file as a specified asset type. It does this by searching for a loader that supports the asset type and the data format provided. If it finds more than one such loader then it will use the first loader it finds.

The Proxy class also contains convenience functions for loading particular asset types. The following code shows how you can use `getAsset()` and the convenience functions `getBitmap2DAsset()` to load assets:

```
// Load Bitmap2DAsset using the getAsset() function:
Proxy proxy(filesystem, context);
Bitmap2DAsset* bitmap1 =
    (Bitmap2DAsset*)proxy.getAsset(Asset::TYPE_BITMAP2D, "image.jpg");
Bitmap2DAsset* bitmap2 = proxy.getBitmap2DAsset("image.pkm");
```

If there are several loaders for the same asset type, as is the case with `Bitmap2DAsset`, the proxy tries all of them to find a suitable loader for the file format if one exists.

You can add a loaders to support additional formats using the `addLoader()` function of the Proxy class. The `addPriorityLoader()` function lets you add a loader that is used in preference to the existing loaders.

See the UI Engine API documentation for a full list of the Proxy member functions of the class. The documentation is located in one of the following directories:

On Microsoft Windows:

Mali GPU UI Engine `vm.n\doc`

On Linux:

Mali_GPU_UI_Engine_`vm.n/doc`

4.1.4 The UI Engine Graphics component

This section describes the UI Engine graphics component, and contains information about:

- *The Context interface*
- *Buffer and vertex declarations* on page 4-8.

The Context interface

You can use the Context interface to create a context to draw with, and also to create GL objects that are linked to a certain context, including:

- textures
- buffers
- shader programs.

Use the Context interface as follows:

```
// Create a 800x600 context
Managed<Context> context = system->createContext(800, 600);

// Shader source for one Vertex and one Fragment shader
char* vertexsource =
"attribute vec4 POSITION; \
attribute vec4 COLOR;\
varying vec4 col;\
void main(void)\
{\
    col = COLOR;\
    gl_Position = POSITION;\
}";
char* fragmentsource =
"varying vec4 col;\
void main(void)\
{\
    gl_FragColor = col;\
}";

// Create two shaders
Shader* vertexshader = context->createShader(GL_VERTEX_SHADER, vertexsource);
Shader* fragmentshader = context->createShader(GL_FRAGMENT_SHADER, fragmentsource);

// Use the shaders to create a program:
Managed<Program> program = context->createProgram(vertexshader, fragmentshader);
context->setProgram(program);
do
{
    // Draw Something
}while(context->update());
context->close();
```

————— Note —————

The source for each shader is sent as individual strings. The escape character \ at the end of each line, instructs the shader compiler to ignore end-of-line characters. You can use these characters in your code to make it more readable.

Buffer and vertex declarations

You can create vertex buffers and index buffers, in a unified way, using `Context::createBuffer()`. When creating a buffer, specify the target as either:

- specify the target as `GL_ARRAY_BUFFER` for vertex buffers, to include vertex attributes
- `GL_ELEMENT_ARRAY_BUFFER` for index buffers, to include index lists for `glDrawElements`.

Use `Context::createBuffer()` as follows:

```
GLfloat vdata[] = {
    -1.0, -1.0,
     1.0, -1.0,
     0.0,  1.0
};
Managed<Buffer> vbo =
    context->createBuffer(GL_ARRAY_BUFFER, sizeof(vdata),
        sizeof(GLfloat)*2); /* Two GL floats per vertex */
vbo->setData(0, sizeof(vdata), vdata);
short idata[] = {
    0, 1, 3,
    1, 2, 3
}
Managed<Buffer> ibo =
    context->createBuffer(GL_ELEMENT_ARRAY_BUFFER,
        sizeof(short)*3, idata);
ibo->setData(0, sizeof(idata), idata);
```

For vertex buffers, you must also create a `VertexDeclaration`. The `VertexDeclaration` specifies how the data in the buffer is read from the active vertex buffer during draw calls. A `VertexDeclaration` consists of one or more `VertexElements`, where each `VertexElement` is a description of the data per stream. Create a `VertexDeclaration` for the vertex buffer object as follows:

```
VertexElement elements[1];

// Our vertex buffer has two components
elements[0].components = 2;

// The data starts at 0
elements[0].offset = 0;

// data in vertex buffer must be used for POSITION
elements[0].semantic = POSITION;

// It must use stream 0
elements[0].stream = 0;

// The components in the buffer are floats
elements[0].type = GL_FLOAT;
Managed<VertexDeclaration> vertexDeclaration = context->createVertexDeclaration(e1m,
1);
```

4.1.5 The UI Engine Utility classes

The UI Engine Library includes a set of Utility classes for performing vector and matrix math with vectors and matrices. The vector classes, `vec2`, `vec3`, and `vec4`, and the matrix classes, `mat2`, `mat3`, and `mat4` support the standard math operators such as addition and multiplication. The matrix classes also have additional functionality such as `identity()`, `rotation()`, and `invert()`.

The UI Engine Library also includes a set of custom implementations of commonly-used data structures such as template-based arrays, maps, strings, and trees:

- The Array implementation represents a dynamic array that resizes itself to allocate space for more elements.
- The Map implementation represents a map where one key maps to one value. If the requested key does not exist, it is created.
- The String implementation provides reference-counted strings.
- The Tree implementation represents a tree of nodes where each node has a single-linked list of child nodes. Each node can contain data and knows only about its first child and next sibling.

4.2 Using the Mali Binary Asset Format

This section describes the Mali Binary Asset format and how to load a Mali Binary Asset scene using the Proxy class, it contains the following sections:

- *The Mali Binary Asset format*
- *Loading an Mali Binary Asset scene using the Proxy class*

4.2.1 The Mali Binary Asset format

The Mali Binary Asset file format provides an optimized binary representation of a 3D scene. It can contain all the assets necessary to draw the scene using the UI Engine Library with the possibility to embed or reference external assets.

The Mali Binary Asset format is not intended as an intermediate file to be re-used later. This job is done better by industry-standardized formats such as COLLADA. Use the Mali Binary Asset Exporter to export from the COLLADA format to the Mali Binary Asset format. See the *GPU Mali Binary Asset Exporter User Guide* for more information.

4.2.2 Loading an Mali Binary Asset scene using the Proxy class

The following code shows how to use the Proxy class to load entire scenes that use the Mali Binary Asset format:

```
Managed<System> system = create_system_backend();
Managed<FileSystem> filesystem = system->createFileSystem("files/");
Managed<Context> context = system->createContext(800, 600);
Proxy proxy(filesystem, context);
SceneAsset* scene = proxy.getSceneAsset("mba_test_scene.MBA");

// You can now access parts of scene by name or id
GeometryAsset* sphere01_mesh = (GeometryAsset*)scene->getAsset(Asset::TYPE_GEOMETRY,
"Sphere01-mesh");
```

You can draw one of the geometry meshes contained in a Mali Binary Asset scene using the `GeometryAsset::draw` function:

```
while (true) {
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
    sphere01_mesh->draw();
    if (!context->update()) break;
}
```

4.3 UI Engine tutorial examples

The UI Engine also includes a set of code samples showing how to use some of the features of UI Engine Library. These examples are in the following directories:

On Microsoft Windows:

Mali GPU UI Engine `vm.n\tutorial_examples`

On Linux:

Mali_GPU_UI_Engine_vm.n/tutorial_examples

The examples are:

Introduction to the system interface

This illustrates how to use the System interface, how to create and manage MDE objects, and how to handle MDE exceptions.

Creating an OpenGL Rendering Context

This creates a rendering context and draws a simple image to it.

Using the FileSystem Interface

This demonstrates how files can be created and accessed using the UI Engine.

Using the Timer Interface

This exemplifies the use of the Timer class from the library.

Drawing a Indexed Quad the Hard Way

This example illustrates the use of `DrawElements()` to draw a simple shape with indexed data.

Drawing a Triangle the Hard Way

This example illustrates the use of `DrawArrays()` to draw a simple shape.

Texturing

This example draws a simple textured shape.

Proxy

This example demonstrates the use of the Proxy class.

Loading an MBA Scene Using the Proxy

This example illustrates loading and using a scene in Mali Binary Asset format using the Proxy class.

Advanced MBA Scene Rendering

This example demonstrates loading and using multiple assets from an MBA file.

Dataset Optimization

This example demonstrates techniques to optimize rendering.

Creating a Custom Loader

This example shows how to create and use a custom asset loader.

Chapter 5

Example Shaders

This chapter describes how to use the example shaders supplied with the UI Engine. It contains the following sections:

- *About the example shaders* on page 5-2
- *Example shaders* on page 5-3.

5.1 About the example shaders

Shaders are programs that provide *programmability* in the OpenGL ES 2.0 applications. ARM provides an extensive set of example shaders. See the *Mali GPU Shader Library Release Notes* and *Mali GPU Shader Library User Guide* for more information. Also see the *Mali GPU Shader Development Studio User Guide* for information on how to develop and view the effects produced by these shaders.

The UI Engine also contains some example shaders that are essentially a subset of the shaders in the Shader Library. These example shaders also contain application programs that use the UI Engine and standard OpenGL ES API functions to execute the shader programs and display the resulting images.

5.1.1 Shader demo programs

Each shader example includes a demo program written in C++. These programs use the UI Engine to simplify the task of initializing and running a shader program on a particular object.

The examples *Simple shader example* on page 5-3, *Coloring* on page 5-4, and *Texturing* on page 5-4 explicitly set up the model that the featured shader programs are used on.

The examples *Lighting* on page 5-5, *Bump mapping* on page 5-5, *Spherical environment mapping* on page 5-6, and *Cube environment reflection mapping* on page 5-7 use models stored in Mali Binary Asset format. See the *Mali GPU Mali Binary Asset Exporter User Guide* for more information about the Mali Binary Asset format.

In addition to initializing and drawing the model, each demo program enables you to:

- rotate the object using the arrows
- rotate the camera around the object using the arrows while holding the space button down.

5.1.2 OpenGL ES 2.0

When creating the OpenGL ES 2.0 API, Khronos removed all shader program fixed functionality from the graphics pipeline to prevent duplicate functionality. This means that when you draw objects using OpenGL ES 2.0, you must provide a shader program that specifies how to draw the object.

You must implement functionality, such as lighting, that was available by default in OpenGL ES 1.1. This approach is particularly suitable for embedded device applications, where resources are limited, because it provides greater flexibility to create specific visual effects and optimize shader programs, while minimizing the complexity of creating software drivers. See *Lighting* on page 5-5 for more information on the example Shading library program in the Mali Developer Tools.

5.2 Example shaders

This section provides examples of shaders that you can use with the UI Engine. It includes the following examples:

- *Simple shader example*
- *Coloring* on page 5-4
- *Texturing* on page 5-4
- *Lighting* on page 5-5
- *Bump mapping* on page 5-5
- *Spherical environment mapping* on page 5-6
- *Cube environment reflection mapping* on page 5-7.

5.2.1 Simple shader example

Figure 5-1 shows a simple shader example.

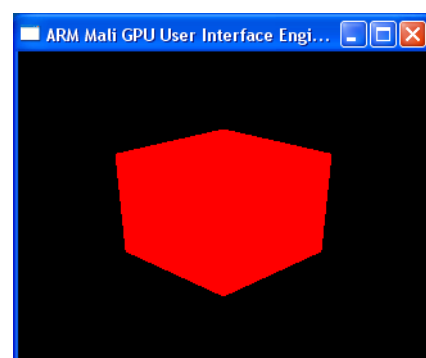


Figure 5-1 Simple shader example

This example demonstrates the simplest shader possible to draw 3D graphics. Each vertex is transformed to projection space and each fragment is colored red.

Because the shader is programmed to draw every fragment in the same red color, it is not easy to see that this object is actually 3-dimensional. To achieve a better 3D effect, you can, for example, draw the different sides of the cube in different colors, apply textures, or apply lighting to the object. The other examples in this section demonstrate these techniques.

The source code for this example is contained in the following files:

```
shader_library\shaders\hello_world\src\hello_world.cpp
shader_library\shaders\hello_world\shaders\hello_world.vert
shader_library\shaders\hello_world\shaders\hello_world.frag
```

5.2.2 Coloring

Figure 5-2 shows an example of per-vertex coloring.

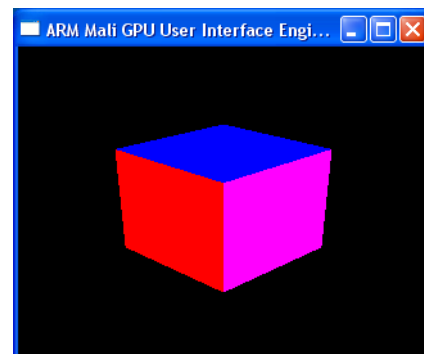


Figure 5-2 Coloring example

This example adds to the *Simple shader example* on page 5-3 by introducing per-vertex coloring. In addition to sending per-vertex positions, the client code also inputs per-vertex colors that are interpolated across each triangle using varyings. Because all the vertices for each side of the cube in this example are assigned the same color, each face is colored in one color.

The source code for this example is contained in the following files:

```
shader_library\shaders\coloring\src\coloring.cpp
```

```
shader_library\shaders\coloring\shaders\coloring.vert
```

```
shader_library\shaders\coloring\shaders\coloring.frag
```

5.2.3 Texturing

Figure 5-3 shows an example of using textures.



Figure 5-3 Texturing example

This example demonstrates how to apply textures to the faces of the cube from the single color examples. The per-vertex color attributes are replaced by per-vertex texture coordinates. Because OpenGL ES 2.0 has no fixed functionality, the texture is provided explicitly to the fragment shader in the form of a uniform.

The source code for this example is contained in the following files:

```
shader_library\shaders\texturing\src\texturing.cpp
```

```
shader_library\shaders\texturing\shaders\texturing.vert
```

```
shader_library\shaders\texturing\shaders\texturing.frag
```

5.2.4 Lighting

This example demonstrates how to apply per-vertex (Gouraud) lighting and per-fragment (Phong) lighting to an object. Lighting functionality is not included in OpenGL ES, and therefore you must implement your own lighting when developing your shader programs.

Figure 5-4 shows examples of Gouraud lighting and Phong lighting.

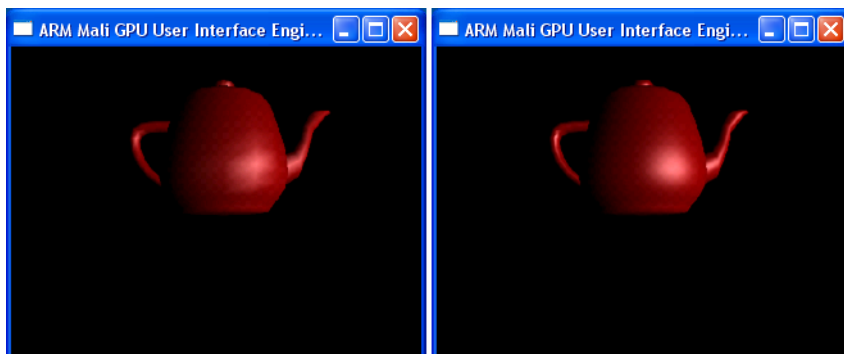


Figure 5-4 Gouraud and Phong lighting examples

With Gouraud lighting, most of the calculations are performed in the vertex shader and interpolated across the surface. Gouraud lighting provides a much cheaper algorithm than Phong because there are usually fewer vertices than fragments. However, it also means the lighting is very dependent on the number of triangles in the mesh so the results are typically poor.

With Phong lighting, more work is done in the fragment shader to produce per-pixel lighting. Phong lighting typically results in

The demonstration application for this example also introduces the Mali Binary Asset format and how to use the UI Engine to load scenes stored in this format. The source code for this example is contained in the following files:

```
shader_library\shaders\lighting\src\lighting.cpp
```

```
shader_library\shaders\lighting\shaders\per_vertex_lighting.vert
```

```
shader_library\shaders\lighting\shaders\per_vertex_lighting.frag
```

```
shader_library\shaders\lighting\shaders\per_fragment_lighting.vert
```

```
shader_library\shaders\lighting\shaders\per_fragment_lighting.frag
```

```
shader_library\shaders\lighting\data\teapot.dae
```

```
shader_library\shaders\lighting\data\teapot.mba
```

5.2.5 Bump mapping

Bump mapping is an advanced lighting model that is used for making an object appear to have more 3-dimensional information than it actually does. This is usually cheaper than modelling the object with this information by using more triangles.

Figure 5-5 on page 5-6 shows an example of an object before and after applying bump mapping.



Figure 5-5 Bump mapping before and after examples

If you are using bump mapping, you must provide a texture called a normal map or a bump map, in addition to the ordinary texture that describes the colors of the object. A bump map contains information about the direction of the surface normals at the various positions of the original texture, in the form of vectors encoded in the RGB values of the bump map texture.

These normals are then used in the lighting calculations instead of the vertex normals to calculate how the lighting must reflect off the object. This means that light is reflected differently depending on the following factors:

- the normals provided by the normal map
- the light position
- the camera position.

These factors make the surface look as though it has more 3D features than it really does. This example demonstrates how to use bump mapping together with diffuse, specular, and ambient lighting to achieve the effect described.

The source code for this example is contained in the following files:

```
shader_library\shaders\bumpmapping\src\bumpmapping.cpp
shader_library\shaders\bumpmapping\shaders\bumpmapping.vert
shader_library\shaders\bumpmapping\shaders\bumpmapping.frag
```

5.2.6 Spherical environment mapping

Spherical environment mapping is a technique for mapping a 2-dimensional texture representing the environment around a 3-dimensional object onto the actual object. The texture is mapped relative to the environment so that the texture keeps its position relative to the environment, even when the object it is mapped onto is rotated.

Figure 5-6 on page 5-7 shows an example of spherical environment mapping. The figure shows a 2-dimensional texture, together with the same texture mapped onto a 3-dimensional object, using spherical environment mapping.

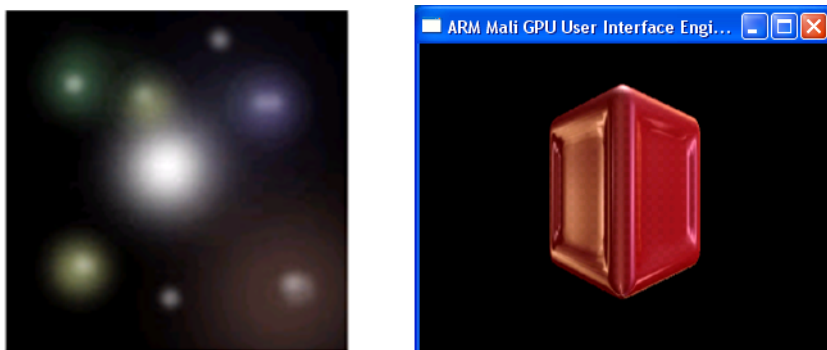


Figure 5-6 Example spherical environment mapping

You can use spherical environment mapping for creating a mirror-like effect, but in this example it is used for simulating several specular light sources. This lighting effect can be achieved by expanding the code in the lighting example, but this becomes expensive as the number of light sources increases. The technique demonstrated in this example is cheap and runs in constant time relative to the number of light sources.

The source code for this example is contained in the following files:

`shader_library\shaders\environment_mapping\src\environment_mapping.cpp`

`shader_library\shaders\environment_mapping\shaders\environment_mapping.vert`

`shader_library\shaders\environment_mapping\shaders\environment_mapping.frag`

`shader_library\shaders\environment_mapping\data\cube.dae`

`shader_library\shaders\environment_mapping\data\cube.mba`

`shader_library\shaders\environment_mapping\data\spotlight.jpg`

5.2.7 Cube environment reflection mapping

This example demonstrates how you can perform environment mapping using a cube map to achieve the same effect that the spherical technique example demonstrates. The cube map represents the environment around the object and this shader enables you to map this environment onto a 3D object using the camera position to create a metallic reflective surface.

Figure 5-7 on page 5-8 shows an example environment around an object, and Figure 5-8 on page 5-8 shows the same environment mapped onto a 3-dimensional object.



Figure 5-7 Example cube map environment



Figure 5-8 Using environment reflection mapping

As with spherical environment mapping, the texture keeps its position relative to the environment, even when the object is rotated.

The demonstration application for this example also draws the environment in the form of a surrounding cube with the same cube map textures applied. The teapot and the camera are inside the cube, to highlight the mirror-like property of this technique.

The source code for this example is contained in the following files:

`shader_library\shaders\cube_mapping\src\cube_mapping.cpp`

`shader_library\shaders\cube_mapping\shaders\skybox.vert`

`shader_library\shaders\cube_mapping\shaders\skybox.frag`

`shader_library\shaders\cube_mapping\shaders\cube_reflection_mapping.vert`

`shader_library\shaders\cube_mapping\shaders\cube_reflection_mapping.frag`

`shader_library\shaders\cube_mapping\data\1.png`

shader_library\shaders\cube_mapping\data\2.png
shader_library\shaders\cube_mapping\data\3.png
shader_library\shaders\cube_mapping\data\4.png
shader_library\shaders\cube_mapping\data\ned.png
shader_library\shaders\cube_mapping\data\opp.png
shader_library\shaders\cube_mapping\data\teapot.dae
shader_library\shaders\cube_mapping\data\teapot.mba

Glossary

This glossary describes some of the terms used in ARM manuals. Where terms can have several meanings, the meaning presented here is intended.

AEL *See* ARM Embedded Linux.

API *See* Application Programming Interface.

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.

ARM Embedded Linux (AEL)

A version of Linux ported to the ARM architecture.

COLLABorative Design Activity (COLLADA) files

Open-standard XML files that describe digital assets. COLLADA files are compatible with many graphics applications.

COLLADA *See* COLLABorative Design Activity (COLLADA) files.

Fps Frames per second. The number of individual frames that are rendered every second, to create an animation. Typical games and GUI applications run at 20-30 fps.

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

GPU *See* Graphics Processor Unit.

Graphics Processor Unit (GPU)

A hardware accelerator for graphics systems using OpenGL ES and OpenVG. The hardware accelerator comprises of an optional geometry processor and a pixel processor together with memory management. Mali programmable GPUs, such as the Mali-200 and Mali-400 MP GPUs, consist of a geometry processor and at least one pixel processor. Mali fixed-function GPUs, such as the Mali-55 GPU consist of a pixel processor only.

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.

See also Performance Analysis Tool.

Mali

A name given to graphics software and hardware products from ARM that aid 2D and 3D acceleration.

Mali Binary Asset

The Mali Binary Asset file format provides an optimized binary representation of a 3D scene for loading into the Mali User Interface Engine Library.

Mali User Interface Engine

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

Mali User Interface Engine Library

A C++ class framework for developing OpenGL ES 2.0 applications for the Mali GPU. The Mali User Interface Engine Library is a component of the Mali Developer Tools.

Mali Developer Tools

A set of development programs that enables software developers to create graphic applications.

Mesh

In graphics, a 3-dimensional arrangement of vertices and triangles, representing an object.

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 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.

Red Hat Enterprise Linux (RHEL)

A version of desktop Linux operating systems.

RHEL

See Red Hat Enterprise Linux (RHEL).

Performance variable

Data produced by the instrumented OpenGL ES 2.0 Emulator, that can be displayed and analyzed as statistical information in the Performance Analysis Tool.

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.

Shader Library

A set of shader examples, tutorials, and other information, designed to assist with developing shader programs for the Mali GPU. The Shader Library is a component of Mali Developer Tools.

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.