Mali™ GPU User Interface Engine

Version: 2.3.1

Application Development Guide

Non-Confidential - Draft - Beta

ARM®

ARM DUI 0527A-02a (ID070710)

Mali GPU User Interface Engine
Application Development Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
30 June 2010	A02a	Non Confidential	First version of document. Beta release.

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.

Product Status

The information in this document is for a Beta product, that is a product under development.

Web Address

`http://www.arm.com`

Contents

Mali GPU User Interface Engine Application Development Guide

List of Tables
Mali GPU User Interface Engine Application Development Guide

List of Figures
Mali GPU User Interface Engine Application Development Guide

Preface

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

About this book

This is the *Mali GPU User Interface Engine Application Development 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 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 C++ applications for the Mali GPU range.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this for an introduction to the Mali User Interface Engine.

Chapter 2 ***Tutorial on the System Classes***

Read this for tutorial on using the basic System objects.

Chapter 3 ***Tutorial on Drawing 2D Shapes***

Read this for tutorial on drawing simple 2D shapes.

Chapter 4 ***Tutorial on Drawing 3D Shapes***

Read this for a tutorial on drawing 3D shapes.

Chapter 5 ***Tutorial on the Lotion User Interface Classes***

Read this for a description of the lotion application and the objects in the lotion library.

Chapter 6 ***Tutorial on Constructing Custom Shaders***

Read this for a tutorial on creating custom shaders.

Appendix A ***Matrix and Vector Operations***

Read this for a review of matrix and vector operations that are typically used by shaders.

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

Typographical Conventions

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.
< 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 `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 0527A-02a
- 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 the MDE library. 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 User Interface Engine is a library of C++ functions that can be helpful when building OpenGL ES 2.0 applications and user interfaces for a platform with a Mali GPU. You can use it for creating new applications, training, and exploration of implementation possibilities.

The Mali GPU User Interface Engine and its libraries are supplied as part of the Mali Developer Tools which help you to develop OpenGL ES graphics applications on your host computer.

You can use the Mali GPU User Interface Engine and OpenGL ES Emulator to visualize and debug your graphics applications before the Mali GPU hardware is available.

1.1.1 Development environment

The complete development environment for Mali GPU Developer Tools is shown in Figure 1-1:

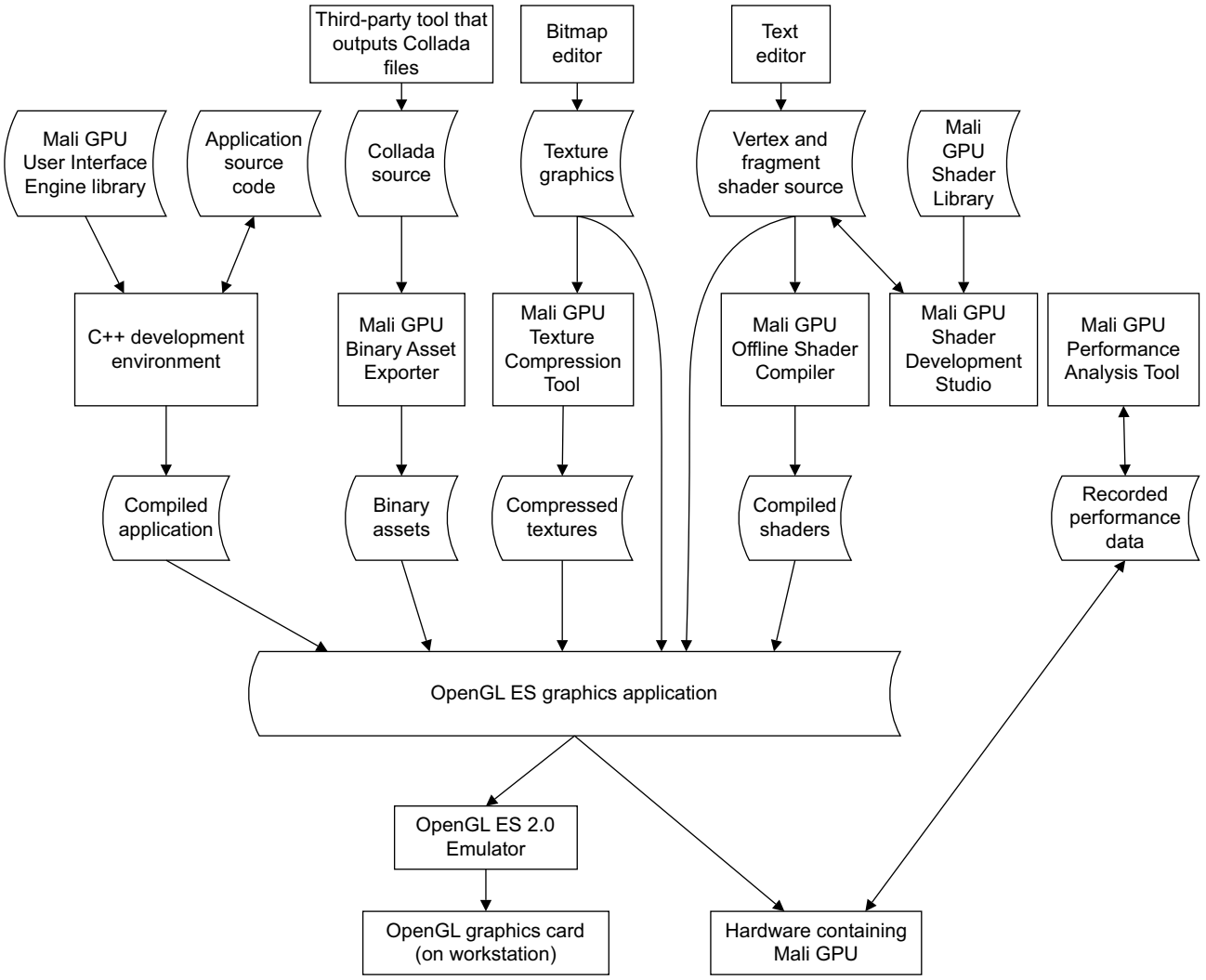

Figure 1-1 Mali GPU Development Tools work flow

This guide concentrates on the Mali GPU User Interface Engine and OpenGL ES Emulator. See the documentation for the other tools for information on using them in a development environment.

The demonstration applications provided with Mali GPU User Interface Engine include examples of source and compiled assets.

1.1.2 Class hierarchy

This section provides a basic overview of the MDE library modules:

- *Assets module*
- *System module* on page 1-4
- *Graphics module* on page 1-5
- *Vector Math module* on page 1-5
- *Templates module* on page 1-5.

Assets module

The Assets module contains classes related to loading, or creating, assets.

The `Asset` class shown in Figure 1-2 is the parent class for assets.

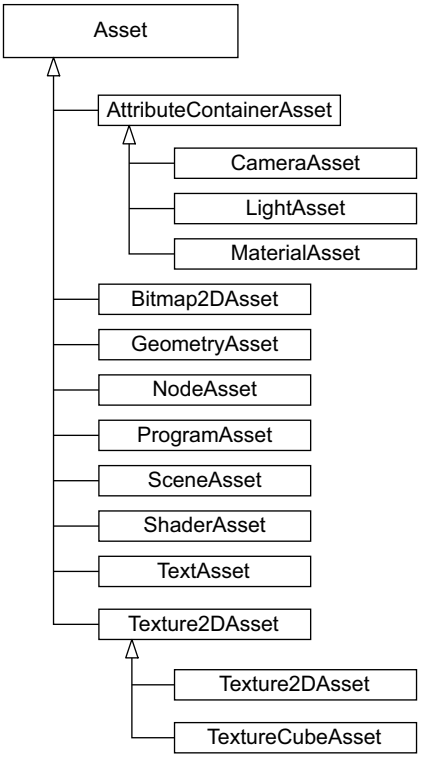

Figure 1-2 Asset classes

The `Loader` classes shown in Figure 1-3 on page 1-4 read an input stream and create an asset. The `Proxy` class is a collection of assets that provides a unified interface to the various loader classes.

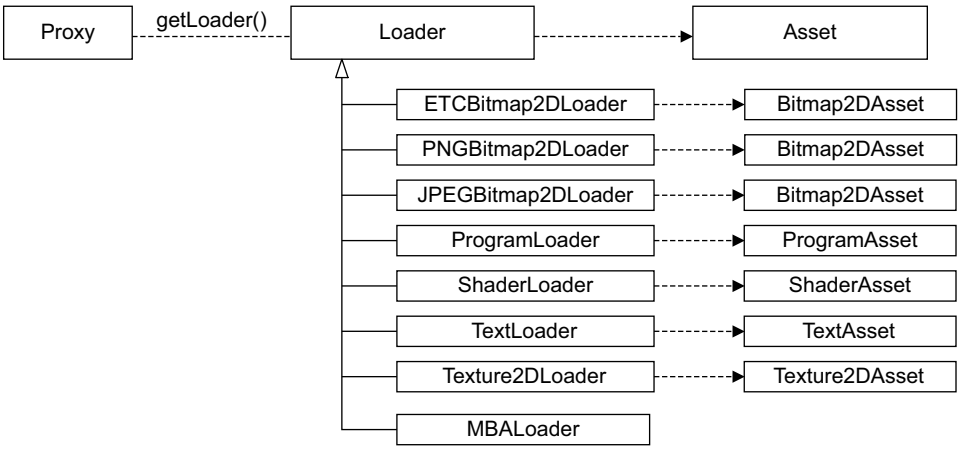

Figure 1-3 Loader classes

Unlike the other general loaders, the `MBALoader` loads scene assets. A scene asset might however contain or reference a different asset type. See Figure 1-3:

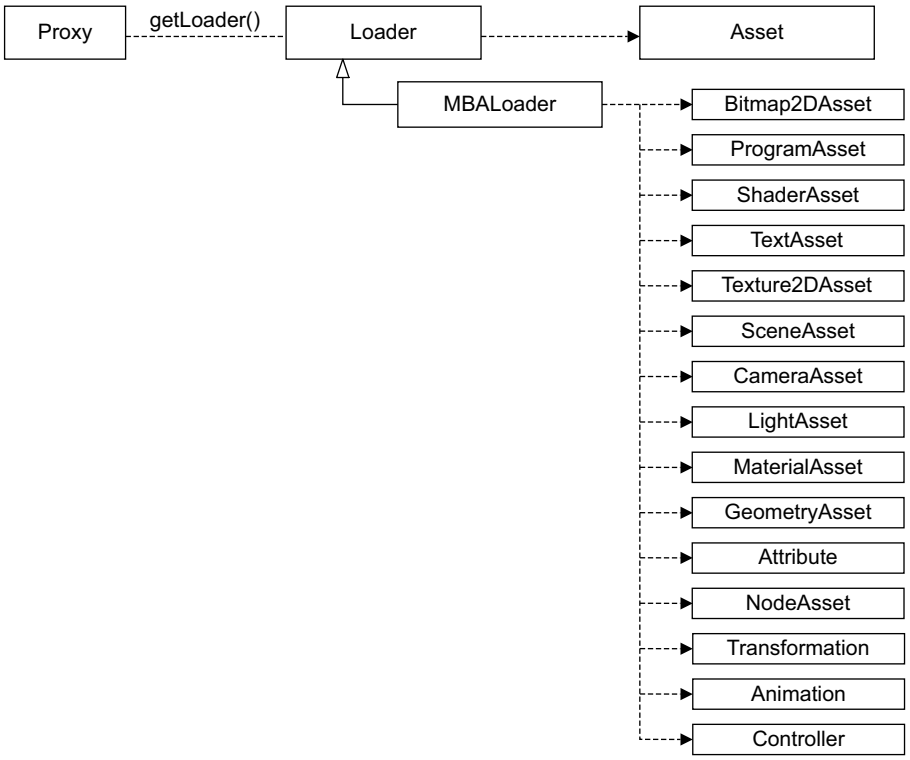

Figure 1-4 Loader classes

System module

The System module uses abstract interfaces to encapsulate platform-specific concepts:

- `Exception`
  - `FileNotFoundException`
  - `GLException`
  - `IOException`
  - `NotSupportedException`

- `FileSystem`
- `InputDevice`
  - `Joystick`
  - `Keyboard`
  - `Mouse`
- `Managed<T>`
- `Object` (base class for all library objects)
- `ObjectMonitor`
- `System` (main interface for the MDE library)
- `Timer`.

Graphics module

The Graphics module contains classes related to graphic primitives and textures such as:

- `VertexElement`
- `VertexDeclaration`
- `Buffer`
- `Context` abstract interface
- `Shader`
- `Texture`
- `Texture2D`
- `TextureCube`.

Vector Math module

The Math module contains classes related to matrix and vector math:

- `mat2`
- `mat3`
- `mat4`
- `vec2t<T>`
- `vec3t<T>`
- `vec4t<T>`.

Templates module

The Templates module contains reusable data structures. It is a partial replacement for the STL library:

- `Array<T>`
- `Map<KeyType, ValueType>`
- `Tree<T>`.

Chapter 2
Tutorial on the System Classes

This chapter contains a step-by-step tutorial on using the main classes in the MDE Library `System` module. It contains the following sections:

2.1 About the System components

A fundamental part of the MDE 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 MDE library contains implementations for *ARM Embedded Linux* (AEL), Windows, and Red Hat Enterprise Linux.

2.2 Creating a simple application with just the System and Keyboard classes

The following steps describe how to create a System and Keyboard interface:

1. Open the `main.cpp` file from the **01 - Introduction to the System Interface** project in the `MDE Tutorial Examples` folder.

   Example 2-1 shows how to create a `System` interface to use with the other library components:

Example 2-1 Create a System object and a context

```
#include <mde/mde.h>
#include <cstdio>

int main()
{
MDE::Managed<MDE::System> system = MDE::create_system_backend();

#ifdef MDE_OS_PC_LINUX
    MDE::Context* context = system->createContext(320,240);
#endif
...
```

   The `Managed<>` template provides automatic lifetime checking and resource deletion. ARM recommends that you always use it when creating library objects. If `Managed<>` is not used, there must be a corresponding call to `release()` for each object creation. For example:

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

   The `createContext()` parameters specify the width and height of the graphics window. For this example, the framebuffer is created but will not be used.

2. Example 2-2 shows how to create a `Keyboard` interface:

Example 2-2 Create a Keyboard object

```
...
#ifdef MDE_OS_PC_LINUX
    MDE::Keyboard* keyboard = system->createKeyboard(context);
#else
    MDE::Keyboard* keyboard = system->createKeyboard();
#endif
...
```

   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.

3. Keyboard input sequences the program to the next stage. Example 2-3 on page 2-4 shows how to prompt the user to press the **Esc** key:

Example 2-3 Reading keyboard input

```
...
    printf("Press ESC to continue\n");
    while (true)
    {
        if (keyboard->getSpecialKeyState(MDE::Keyboard::KEY_ESCAPE)) break;
    }

    keyboard->release();
...
```

Because the `Keyboard` object in Example 2-2 on page 2-3 was not created with the `Managed<>` template, you must explicitly call `release()` on the keyboard interface.

4. Example 2-4 shows the next stage of the application that deliberately raises an exception by trying to create a non-existent file system:

Example 2-4 Basic exception handling

```
...
    try
    {
        MDE::Managed<MDE::FileSystem> fs = system->createFileSystem("non-existing root");
        fs->createInputStream("non-existing file");

    }
    catch (MDE::Exception& e)
    {
        printf("An expected exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

You can put the standard C++ `try` blocks around code that you are testing.

Exceptions generated by the UI Engine have types derived from `MDE::Exception`. Use a `catch(MDE::Exception)` statement to catch all exceptions generated by the UI Engine. For example code, ARM recommends putting a `try` block around all of the code in `main()` as shown in Example 2-5:

Example 2-5 Using a try catch block

```
main()
{
    try
    {
    // all executable code
    }
    catch (MDE::Exception& e)
    {
    // exception handling code
    }
}
```

5. Compile and run the project to test keyboard input and exception trapping. Figure 2-1 on page 2-5 shows the system window:

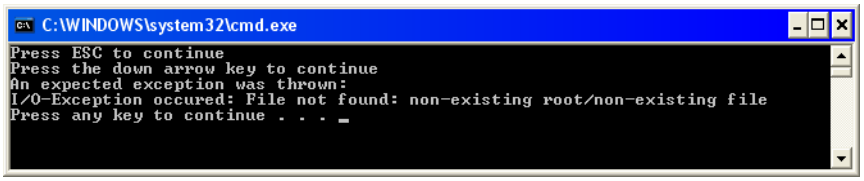

Figure 2-1 System window showing exception trapping

Note

There is not a graphical display window because nothing was drawn to a context.

2.3 Creating an OpenGL ES rendering context

This example draws to the context:

1. Open the `main.cpp` file from the **02 - Creating an OpenGL Rendering Context** project.

   Example 2-6 shows the creation of the System, Context, and Keyboard interfaces:

Example 2-6 Create a rendering context

```
#include <mde/mde.h>
using namespace MDE;
int main()
{
    try // all of the code is in a try block
    {
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);

        #ifdef MDE_OS_PC_LINUX
            MDE::Managed<MDE::Keyboard> keyboard = system->createKeyboard(context);
        #else
            MDE::Managed<MDE::Keyboard> keyboard = system->createKeyboard();
        #endif

 ...
```

2. Example 2-7 shows how to create a mouse object with the MDE library:

Example 2-7 Creating a mouse

```
      int xClick, yClick;
      // create a mouse object
      Managed<Mouse> mouse= system->createMouse(context);
```

3. Example 2-8 shows the draw loop:

Example 2-8 Draw loop for empty context

```
 ...
      while(true)
       {
           glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
           glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

           /*
           * The update() function will swap 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;
...
```

The clear color is specified as `(1.0f, 0.0f, 0.0f, 1.0f)` which is red.

Each call to `update()` swaps the draw buffers and returns `true` if the context is still active.

Drawing and swapping buffers continues until either:

- the context is told to close by the operating system. If this happens, `context->update()` returns `false`.
- the keyboard interface processes an **Esc** keypress.

4. The mouse detection code is shown in Example 2-9:

Example 2-9 Reacting to mouse events

```
        if(mouse->isButtonDown(Mouse::LEFT_BUTTON))
        {
            // Get X and Y coordinates
            xClick = mouse->getAxisPosition(Mouse::X_AXIS);
            yClick = mouse->getAxisPosition(Mouse::Y_AXIS);
            printf("Mouse click at: %i, %i \n", xClick, yClick);
        }
    }
```

Note

- The `getAxisPosition()` functions are called whenever the mouse button is down, so multiple `printf()` statements are called for each button depression.
- If the mouse click is outside the draw window, an event is still generated.
- You can use the `Observer` class to simplify using the mouse in an application.

5. Example 2-10 shows the exception-handler code for the application:

Example 2-10 Catching an exception from the draw loop

```
 ...
    }
    catch(Exception& e)
    {
        printf("An exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

6. Compile and run the project.

   Because the context is used in this example, the graphics output window shown in Figure 2-2 is displayed:

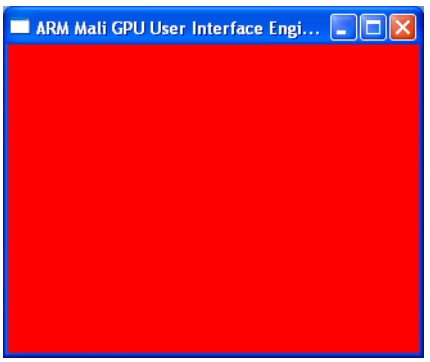

Figure 2-2 Blank graphics context

The background is red because `glClearColor()` was called with red as the color parameter.

2.4 Creating a filesystem interface

You can use the `FileSystem` interface to create streams for reading and writing files in the native or a virtual file system. Use the `System::createFileSystem()` function to retrieve a `FileSystem` object. As with other System objects, you can use `Managed<>` to eliminate the requirement to manually release the created object.

The following steps describe how to create a FileSystem interface:

1. Open the `main.cpp` file from the **03 - Using the FileSystem Interface** project.

   Example 2-11 shows how to create a `system` and `filesystem` interface:

Example 2-11 Create a System object

```
#include <mde/mde.h>
#include <cstdio>

int main()
{
    try
    {
       MDE::Managed<MDE::System> system = MDE::create_system_backend();
       Managed<FileSystem> filesystem = system->createFileSystem(".");
...
```

   The parameter passed to `createFileSystem()` specifies the root directory.

2. Example 2-12 shows how to use the filesystem to attempt to read a file. For this example, the target file does not exist, and the exception is trapped.

Example 2-12 Create an InputStream object

```
...
        printf("Trying to load a non-existent file:\n");

        try
        {
            Managed<InputStream> nonexisting = filesystem->createInputStream("data/non-existing-file.txt");
        }
        catch(IOException& ioe)
        {
            printf("An expected exception was thrown:\n%s\n", ioe.getMessage().getCharString() );
        }

...
```

3. Example 2-13 shows how to create two streams and read the contents of a file:

Example 2-13 Reading from a file

```
...
           Managed<OutputStream> output = filesystem->createOutputStream("data/write.txt");
        Managed<InputStream> input = filesystem->createInputStream("data/read.txt");

        int length = input->getLength();
        // Allocate the size of the file, plus one extra byte for a null-terminator
        char* inputBuffer = new char[length+1];
```

```
        input->read(inputBuffer, sizeof(char), length);
        inputBuffer[length] = '\0';                          // Add null-termination character
    }
...
```

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.

The `Filesystem` methods are:

- `read()` to read content from a file.
- `write()` to write content to a file.
- `getLength()` to get the length of the file
- `getPosition()` to get the current position
- `setPosition()` to manually set the position.

4. Example 2-14 shows how to write the buffer to a file:

Example 2-14 Writing to a file

```
...
        output->write(inputBuffer, sizeof(char), length);
        output->flush();
...
```

5. Example 2-15 shows deleting unused objects and the final `catch` block:

Example 2-15 Deleting the buffer

```
...
        delete [] inputBuffer;
    }
    catch(Exception& e)
    {
        printf("An unexpected exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

The `system` and `filesystem` objects do not require manual deletion because the `Managed<>` template tracks usage and automatically destroys the objects when the application exits.

2.5 Creating a Timer interface

The following steps describe how to create a Timer object:

1. Open the `main.cpp` file from the **04 - Using the Timer Interface** project.
   Example 2-16 shows how to create a System and Timer interface:

Example 2-16 Create a timer

```
#include <mde/mde.h>
#include <cstdio>

using namespace MDE;

/* Portable sleep function */
#ifdef MDE_PLATFORM_ARM_LINUX
        #include <unistd.h>
    void Sleep(unsigned int time)
    {
        usleep(time * 1000);
    }
#endif

#ifdef MDE_OS_PC_LINUX
        #include <unistd.h>
    void Sleep(unsigned int time)
    {
        usleep(time * 1000);
    }
#endif

int main()
{
    try
    {
        Managed<System> system = create_system_backend();
        Managed<Timer> timer = system->createTimer();
...
```

2. Example 2-17 shows how to track changes in the timer.

Example 2-17 Timer sleep calls

```
...
        printf("Reseting timer\n");
        timer->reset();
        printf("Time since object reset is: %f\n", timer->getTime());

        printf("\nSleeping 1000 milliseconds\n");
        Sleep(1000);
        printf("Time since object creation is: %f\n", timer->getTime());

        printf("\nReseting timer\n");
        timer->reset();
        printf("Time since timer reset is: %f\n", timer->getTime());

        printf("\nSleeping 1000 milliseconds\n");
```

```
        Sleep(1000);
        printf("Time since timer reset is: %f\n", timer->getTime());
...
```

3. Example 2-18 shows additional print statements that validate the timer behavior:

Example 2-18 Printing timer values

```
        printf("\nTesting timer intevals:\n");
        printf("\nReseting timer\n");
        timer->reset();
        for(int i = 0; i < 4; i++)
        {
            printf("Interval: %f\n", timer->getInterval());

            printf("\nSleeping %d milliseconds\n", i*50*(i%2));
            Sleep(i*50*(i%2));
        }

        printf("Interval: %f\n", timer->getInterval());
```

4. Example 2-19 shows the final `catch` block:

Example 2-19 Exception catch block

```
...
    catch(Exception& e)
    {
        printf("An unexpected exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

Chapter 3
Tutorial on Drawing 2D Shapes

This chapter describes how to use the MDE library to draw 2D shapes. It contains the following sections:

Note

These examples uses OpenGL ES, the 3D graphics standard, to draw the shapes. OpenVG, the 2D graphics standard, is not supported by the MDE library.

3.1 Drawing a triangle

This section describes how to draw a 2D triangle:

1. Select the tutorial project **06 - Drawing a Triangle the Hard Way** and open the `main.cpp` file.
2. Example 3-1 shows placing the processing code, including the initialization code for the context and keyboard, in to a `try` block:

Example 3-1 Creating the system objects

```
#include <mde/mde.h>

using namespace MDE;

int main(int argc, char** argv)
{
    try
    {
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        Managed<Timer> timer = system->createTimer();
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
...
```

Use the `Context` interface to:

- create a context to draw with
- create GL objects that are linked to a context such as:
  - textures
  - buffers
  - shader programs.

3. Because this example actually draws a shape to the context, there must be a vertex and fragment shader. Example 3-2 shows how to define simple identity shaders:

Example 3-2 Identity shaders

```
...
        /*
        * Define the shader sources. The following GLSL shaders define a simple identity-transform vertex
        * shader and a fragment shader which outputs the color value calculated in the vertex shader.
        */

        const char* vertexsource =
            "attribute vec4 POSITION; \
            attribute vec4 COLOR;\
            varying vec4 col;\
            void main(void)\
            {\
            col = COLOR;\
            gl_Position = POSITION;\
            }";
```

```
        const char* fragmentsource =
           "precision mediump float; \
           varying vec4 col;\
           void main(void)\
           {\
           gl_FragColor = col;\
           }";
...
```

4. The shaders in `vertexsource` and `fragmentsource` are text strings that contain the source. Example 3-3 shows how to associate the shaders with the context:
   a. compile the shaders with `createShader()`
   b. use `createProgram()` to create a new program that uses the compiled shaders
   c. activate the program for the context with `setProgram()`.

Example 3-3 Building the identity shaders

```
...
        /*
        * Create Shader and Program objects. This is done using the createShader() and createProgram() functions
        * on the context object. We then set our newly created program active using setProgram().
        */
        Managed<Shader> vertexshader = context->createShader(GL_VERTEX_SHADER, vertexsource);
        Managed<Shader> fragmentshader = context->createShader(GL_FRAGMENT_SHADER, fragmentsource);
        Managed<Program> program = context->createProgram(vertexshader, fragmentshader);
        context->setProgram(program);
...
```

The `vertexshader`, `fragmentshader`, and `program` all use the `Managed<>` template format to automatically manage object destruction.

5. Specify the triangle as shown in Example 3-4. There are two parts to the specification:
   - Because this is a 2D triangle, there are three coordinates and each coordinate is specified with two numbers that locate the vertex in the XY plane.
   - This example uses vertex coloring, but all of the colors for the triangle vertices are set to red. Colors are specified with four numbers that indicate the red, blue, green, and alpha channel values.

Example 3-4 Triangle coordinates

```
...
        GLfloat vertexData[] =
        {
            -1.0, -1.0,
             1.0, -1.0,
             0.0,  1.0
        };

        GLubyte colorData[] =
        {
            255, 0, 0, 255,
            255, 0, 0, 255,
            255, 0, 0, 255
        };
...
```

6. The coordinate and color information are associated with new buffers as shown in Example 3-5:

Example 3-5 Creating vertex and color buffers for the triangle

```
...
        Managed<Buffer> vertexBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData),
                                       sizeof(GLfloat)*2);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);

        Managed<Buffer> colorBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(colorData),
                                      sizeof(GLubyte)*4);
        colorBuffer->setData(0, sizeof(colorData), colorData);

        VertexElement elements[2];
        elements[0].components = 2;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[0].normalize = false;

        elements[1].components = 4;
        elements[1].offset = 0;
        elements[1].semantic = COLOR;
        elements[1].stream = 1;
        elements[1].type = GL_UNSIGNED_BYTE;
        elements[1].normalize = true;

        Managed<VertexDeclaration> vertexDeclaration = context->createVertexDeclaration(elements, 2);
        context->setVertexDeclaration(vertexDeclaration);
...
```

There are several buffers created in Example 3-5:

a. `createBuffer()` creates `vertexBuffer` and `colorBuffer` objects from `vertexData` and `colorData`

b. A `VertexElement` array named `elements` is created. The array size is two because there are position and color attributes for each vertex.

c. Each VertexElement object describes one vertex attribute. The values of the each of the `elements[]` attributes is initialized:

- The first element has two components because each vertex specifies a position in the XY plane. The type is `GL_FLOAT` because floating-point numbers specify the position.
- The second element has four components because it specifies a color. The type is `GL_UNSIGNED_BYTE` because, in this example, 8-bit numbers specify the color and alpha channel values.

For more information on attributes, see *Passing data as uniforms* on page 6-7.

d. A `vertexDeclaration` object is created from elements. There are two objects in the elements array.

e. `setVertexDeclaration()` associates the new `vertexDeclaration` object with the context.

7. The triangle has been specified, so it can be drawn as shown in Example 3-6 on page 3-5.

For each iteration of the loop:

a. The first element of the vertex buffer that is associated with context is set to the `vertexBuffer` object defined previously.

b. The second element of the vertex buffer that is associated with context is set to the `colorBuffer` object defined previously.

c. `drawArrays()` uses the vertex buffer arrays to draw the first object:

- The object to draw is `GL_TRIANGLES` so three vertices are used.
- The second parameter is 0, so the vertices start from the beginning of the buffer.
- The third parameter specifies the `primitiveCount`. There is only one primitive, a triangle, so the value is 1.

d. A test for the escape key provides a way to end the display.

e. If the context window is closed, `update()` returns `false` and the do loop ends.

Example 3-6 Using the vertex information to draw the triangle

```
...
        do
        {

            context->setVertexBuffer(0, vertexBuffer);
            context->setVertexBuffer(1, colorBuffer);

            /* Draw indexed triangles using the first 3 vertices of the buffers */
            context->drawArrays(GL_TRIANGLES, 0, 1);

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

8. All of the execution code was enclosed in one try catch block which ends with the code in Example 3-7. If there is an exception, a message is displayed in the system window.

Example 3-7 Trapping an exception in the triangle example

```
...

    }
    catch(Exception& e)
    {
        printf("MDE \n%s\n\n", e.getMessage().getCharString());
    }
}
```

9. Figure 3-1 on page 3-6 shows the triangle:

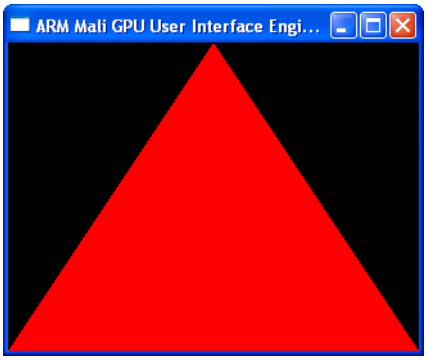

Figure 3-1 The 2D red triangle

3.2 Vertex coloring

Vertex coloring modifies the color of a pixel based on its distance from colored vertices:

3.2.1 Per-vertex coloring with 8-bit color values

1. Locate the tutorial project **06 - Drawing a Triangle the Hard Way** and open the `main.cpp` file.
2. Modify the triangle data from Example 3-4 on page 3-3 to give each vertex a different color as shown in Example 3-8. The vertex coordinates are not changed.

Example 3-8 Vertex shading of simple triangle

```
...
        GLfloat vertexData[] =
        {
            -1.0, -1.0,
             1.0, -1.0,
             0.0,  1.0
        };

        GLubyte colorData[] =
        {
            255, 0, 0, 255,
            0, 255, 0, 255,
            0, 0, 255 255
        };
...
```

3. Rebuild the project and run it. Figure 3-2 shows the triangle with per-vertex coloring:

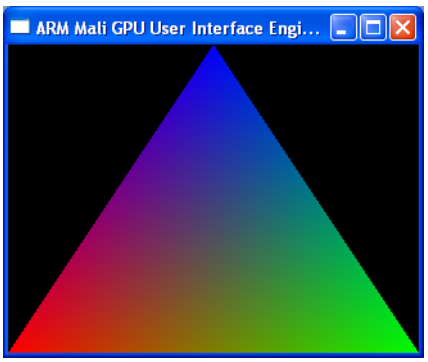

Figure 3-2 The 2D triangle with per-vertex coloring

3.2.2 Per-vertex coloring with floating-point color values

1. Locate the tutorial project **06 - Drawing a Triangle the Hard Way** and open the `main.cpp` file.
2. Modify the triangle data from Example 3-8 to give each vertex a floating-point color value as shown in Example 3-9 on page 3-8. The vertex coordinates are not changed.

Example 3-9 Vertex shading of simple triangle with floating-point color values

```
...
        GLfloat vertexData[] =
        {
            -1.0, -1.0,
             1.0, -1.0,
             0.0,  1.0
        };

        GLfloat colorData[] =
        {
            1.0, 0, 0, 1.0,
            0, 1.0, 0, 1.0,
            0, 0, 1.0 1.0
        };
...
```

3. Change the vertex definition code to use floating-point for color values as shown in Example 3-10:

Example 3-10 Using floating-point values for the color buffers

```
...
        Managed<Buffer> vertexBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData),
sizeof(GLfloat)*2);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);

        Managed<Buffer> colorBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(colorData),
sizeof(GLfloat)*4);
        colorBuffer->setData(0, sizeof(colorData), colorData);

        VertexElement elements[2];
        elements[0].components = 2;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[0].normalize = false;

        elements[1].components = 4;
        elements[1].offset = 0;
        elements[1].semantic = COLOR;
        elements[1].stream = 1;
        elements[1].type = GL_FLOAT;
        elements[1].normalize = true;
...
```

4. Rebuild the project and run it. Figure 3-3 on page 3-9 shows the triangle appearance has not changed:

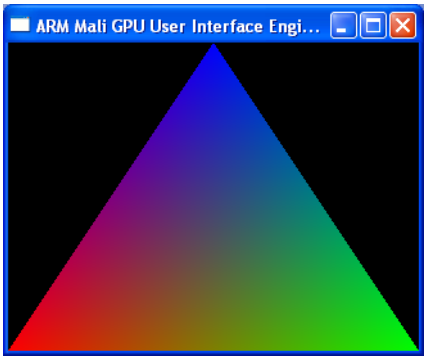

Figure 3-3 The 2D triangle with per-vertex coloring and floating-point color values

3.2.3 Combining the position coordinates and color values into a single array

1. Locate the tutorial project **06 - Drawing a Triangle the Hard Way** and open the `main.cpp` file.
2. Modify the triangle data from Example 3-9 on page 3-8 to move the values from `colorData[]` to the corresponding row in `vertexData[]`. Delete the code that assigns values to `colorData[]`.

Example 3-11 Vertex shading of simple triangle with floating-point color values

```
...
        GLfloat vertexData[] =
        {
            -1.0, -1.0, 1.0, 0, 0, 1.0,
             1.0, -1.0, 0, 1.0, 0, 1.0,
             0.0,  1.0 0, 0, 1.0, 1.0
        };
...
```

3. Modify the vertex information from Example 3-5 on page 3-4 to create `vertexBuffer` based on the larger `vertexData` array as shown in Example 3-12:

Example 3-12 Using floating-point values for the color buffers

```
...
        Managed<Buffer> vertexBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData),
sizeof(GLfloat)*6);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);

        VertexElement elements[2];
        elements[0].components = 2;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[0].normalize = false;

        elements[1].components = 4;
        elements[1].offset = 2*sizeof(GLfloat);
        elements[1].semantic = COLOR;
        elements[1].stream = 1;
```

```
        elements[1].type = GL_FLOAT;
        elements[1].normalize = true;
...
```

The `colorBuffer` array is not required because the color information is now part of the `vertexData` array.

The `offset` for `elements[1]` must be specified as `2*sizeof(GLfloat)` to jump over the position values.

4. The triangle has been specified, so it can be drawn as shown in Example 3-6 on page 3-5.

Example 3-13 Using the vertex information to draw the triangle

```
...
        do
        {

            context->setVertexBuffer(0, vertexBuffer);
            context->setVertexBuffer(1, vertexBuffer);

            /* Draw indexed triangles using the first 3 vertices of the buffers */
            context->drawArrays(GL_TRIANGLES, 0, 1);

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

    }
    catch(Exception& e)
    {
        printf("MDE \n%s\n\n", e.getMessage().getCharString());
    }
}
```

Unlike in Example 3-6 on page 3-5, the context uses vertexBuffer for both stream 0 and stream 1 because both stream values are contained in one array.

5. Rebuild the project and run it. Figure 3-3 on page 3-9 shows the triangle appearance has not changed:

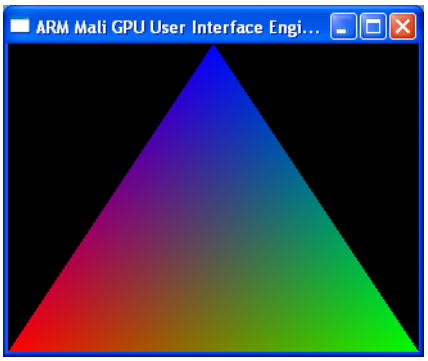

Figure 3-4 The 2D triangle with combined position and color values in one vertexBuffer array

3.3 Drawing a rectangle with a custom texture

This section describes how to draw a rectangle and apply a custom texture:

1. Locate the tutorial project **07 - Texturing** and open the `main.cpp` file.
2. Example 3-1 on page 3-2 shows the standard includes and initialization:

Example 3-14 Creating the context for the texture example

```
#include <mde/mde.h>

using namespace MDE;

int main()
{
    try
    {
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        // Define the shader sources. Basic texturing shaders.
        const char* vertexsource =
            "attribute vec4 POSITION;\
            attribute vec2 TEXCOORD0;\
            varying vec2 texcoord;\
            void main(void)\
            {\
            gl_Position = POSITION;\
            texcoord = TEXCOORD0;\
            }";

        const char* fragmentsource =
            "precision mediump float; \
            varying vec2 texcoord;\
            uniform sampler2D texture;\
            void main(void)\
            {\
            gl_FragColor = texture2D(texture, texcoord);\
            }";

        // Create Shader and Program objects:
        Managed<Shader> vertexshader = context->createShader(GL_VERTEX_SHADER, vertexsource);
        Managed<Shader> fragmentshader = context->createShader(GL_FRAGMENT_SHADER, fragmentsource);
        Managed<Program> program = context->createProgram(vertexshader, fragmentshader);
        context->setProgram(program);
...
```

3. A bit-mapped texture is typically read from a file rather than created in the application. This example however defines the texture data and uses it to create a new texture object as shown in Example 3-2 on page 3-2:

Example 3-15 Identity shaders

```
...
        const int size = 8;
        unsigned int textureData[size*size];
        unsigned int color = 0xFFFFFFFF;
        for(int y = 0; y < size; y++)
        {
            for(int x = 0; x < size; x++ )
            {
                color ^= 0xFFFFFF;
                textureData[y*size+x] = color;
            }
            color ^= 0xFFFFFF;
        }

        // create an empty texture named checkerBoard
        Managed<Texture2D> checkerBoard = context->createTexture2D();
        checkerBoard->setFilterMode(GL_NEAREST, GL_NEAREST);

        // use the textureData array to fill the texture
        checkerBoard->buildMipmaps(0, size, size, GL_RGBA, textureData);
...
```

The MipMaps object is used, and the dimensions of the rendered shape and the texture source are different.

Note

- The `checkboard` texture is a 2D shape.
- The `fragmentshader` texture sets the pixel value based on the value of a point in the 2D texture shape:
  `gl_FragColor = texture2D(texture, texcoord);`

4. As with Example 3-9 on page 3-8 which combined the vertex and color data in one array, the code in Example 3-16 combines the vertices and texture locations:

Example 3-16 Specifying the location for the shape and texture

```
...
        GLfloat vertexData[24] =
        {
            // X     Y     U      V
            -1.0f, -1.0f, 0.0f, 0.0f,   // values for each vertex of the first triangle
             1.0f, -1.0f, 1.0f, 0.0f,
            -1.0f,  1.0f, 0.0f, 1.0f,
             1.0f, -1.0f, 1.0f, 0.0f,  // values for each vertex of the second triangle
             1.0f,  1.0f, 1.0f, 1.0f,
            -1.0f,  1.0f, 0.0f, 1.0f
        };

        /* Set up the vertex buffer */
        Managed<Buffer> vertexBuffer = context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData),
                                       sizeof(GLfloat)*4);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
...
```

5. The second element in the `elements` array is defined, and the `semantic` attribute is `TECOORD0`. See Example 3-3 on page 3-3:

Example 3-17 Setting VertexElement to use a texture

```
...
        VertexElement elements[2];
        elements[0].components = 2;                // xy position
        elements[0].offset = 0;
        elements[0].semantic = POSITION;           // this element is the vertices for the shapes
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[1].components = 2;               // uv position
        elements[1].offset = 2*sizeof(GLfloat);
        elements[1].semantic = TEXCOORD0;         // this element is texture information
        elements[1].stream = 0;
        elements[1].type = GL_FLOAT;

        Managed<VertexDeclaration> vertexDeclaration = context->createVertexDeclaration(elements, 2);
        context->setVertexDeclaration(vertexDeclaration);
...
```

6. The shape and texture have been defined, so draw the shape as shown in Example 3-18.

Example 3-18 Draw a shape with a custom texture

```
...
        do
        {
            glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
            glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
            context->setVertexBuffer(0, vertexBuffer);

            /*
            * When dealing with 2D textures and shaders, we need to assign the texture to a sampler.
            */
            context->setUniformSampler("texture", (Texture*)checkerBoard);

            context->drawArrays( GL_TRIANGLES, 0, 2 );
            if(keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE)) break;
        }while( context->update() );

    }
    catch(Exception& e)
    {
        printf("MDE Exception: \n%s\n\n",e.getMessage().getCharString());
    }
}
```

7. Figure 3-5 on page 3-14 shows the shape with its texture:

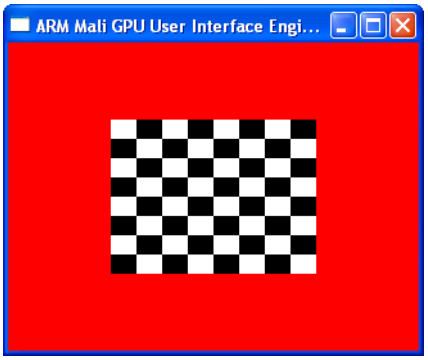

Figure 3-5 A custom texture on a 2D shape

Chapter 4
Tutorial on Drawing 3D Shapes

This chapter describes how to use the MDE library to create 3D shapes. It contains the following sections:

4.1 Drawing a simple 3D cube

This tutorial describes a constructing a simple cube:

1. Locate the shader example project **01 Hello World** and open the `hello_world.cpp` file.
2. Example 4-1 shows the initialization code and the function to calculate the camera position:

Example 4-1 Camera position for hello world

```
#include <mde/mde.h>
using namespace MDE;

static const float PI = 3.14159265;

/**
* Function that calculates the camera position given an angle and a radius. This function
* enables the camera to rotate around the scene object.
*/
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), 2.0f, radius * sin(angleRad));
}

int main(int argc, char * argv[])
{
    try
    {
        // Initialize the demo engine classes
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        Managed<FileSystem> filesystem = system->createFileSystem(".");

        Proxy proxy(filesystem, context);
...
```

3. Unlike with the simple 2D examples, the shaders code is in an external file. The function `getProgram()` loads the shaders and manges their eventual destruction. See Example 4-2.

Example 4-2 Loading the hello_world shaders

```
...
       Program *helloWorldProgram =
          proxy.getProgram("shaders/hello_world.vert;shaders/hello_world.frag");
...
```

4. Set the vertex position as shown in Example 4-3 on page 4-3.

Example 4-3 Setting the vertex position for hello world

```
...
        GLfloat vertexData[] = {
            // FRONT
            -0.5f, -0.5f,  0.5f,
            0.5f, -0.5f,  0.5f,
            -0.5f,  0.5f,  0.5f,
            0.5f,  0.5f,  0.5f,
            // BACK
            -0.5f, -0.5f, -0.5f,
            -0.5f,  0.5f, -0.5f,
            0.5f, -0.5f, -0.5f,
            0.5f,  0.5f, -0.5f,
            // LEFT
            -0.5f, -0.5f,  0.5f,
            -0.5f,  0.5f,  0.5f,
            -0.5f, -0.5f, -0.5f,
            -0.5f,  0.5f, -0.5f,
            // RIGHT
            0.5f, -0.5f, -0.5f,
            0.5f,  0.5f, -0.5f,
            0.5f, -0.5f,  0.5f,
            0.5f,  0.5f,  0.5f,
            // TOP
            -0.5f,  0.5f,  0.5f,
            0.5f,  0.5f,  0.5f,
            -0.5f,  0.5f, -0.5f,
            0.5f,  0.5f, -0.5f,
            // BOTTOM
            -0.5f, -0.5f,  0.5f,
            -0.5f, -0.5f, -0.5f,
            0.5f, -0.5f,  0.5f,
            0.5f, -0.5f, -0.5f,
        };

        Managed<Buffer> vertexBuffer =
            context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData), sizeof(GLfloat)*3);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
        context->setVertexBuffer(0, vertexBuffer);

        VertexElement elements[1];
        elements[0].components = 3;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;


        Managed<VertexDeclaration> vertexDeclaration =
            context->createVertexDeclaration(elements, 1);
        context->setVertexDeclaration(vertexDeclaration);
...
```

5. This project uses a camera position and object position. The view and projection matrices and multiply them to get the modelview projection matrix.

Example 4-4 Setting the hello_world projection

```
...
        // The camera position, target and up vector for use when creating the view matrix
        float cameraAngle = 45.0f;
        vec3 camPos = calculateCamPos(4.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 1.0f, 0.0f);

        // Set up the view and projection matrices and multiply them to get the
        // modelviewprojection matrix.
        mat4 world = mat4::identity();
        mat4 proj = mat4::perspective(35.0f, 1.0f, 0.1f, 20.0f);
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 wvpMatrix = proj * view;

        context->setProgram(helloWorldProgram);
        glEnable(GL_DEPTH_TEST);
 ...
```

6. The do loop checks for keyboard input and either rotates the object or changes the camera position. See Example 4-5.

Example 4-5 Processing hello_world input and changing the view

```
...
        do
        {
            // Exit the application if escape pressed
            if (keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
            {
                return 0;
            }

            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    camPos = calculateCamPos(4.0f, ++cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    camPos = calculateCamPos(4.0f, --cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
            }
            else
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    world *= mat4::rotation(1.0f, vec3(0.0f, 1.0f, 0.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world *= mat4::rotation(-1.0f, vec3(0.0f, 1.0f, 0.0f));
```

```
                }
        }
...
```

7. The projection has been calculated based on the keyboard input and can be redrawn. See Example 4-6.

Example 4-6 Redrawing the hello_world view

```
...
            // Recalculate modelviewprojection matrix in case it has been altered (rotation)
            wvpMatrix = proj * view * world;
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);

            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

            // Draw the object
            context->drawArrays(GL_TRIANGLE_STRIP, 0, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 4, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 8, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 12, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 16, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 20, 2);
        }while ( context->update() );
    }
    catch (Exception &e)
    {
        printf(e.getMessage().getCharString());
    }
}
```

8. Rebuild the project and run it. Figure 4-1 shows the shape:

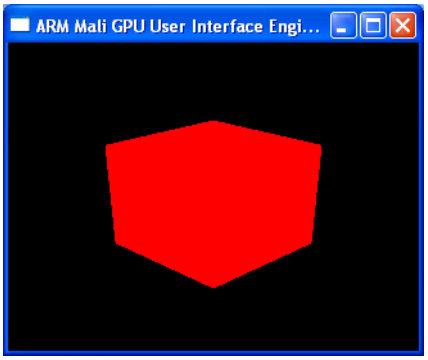

Figure 4-1 The Hello World shape

9. Press the left or right arrows and observe that the object rotates.

10. Press and hold the spacebar while pressing the left or right arrows and observe that the camera position changes.

4.2 Coloring the cube faces

This tutorial shows how to color the cube faces based on the color of the vertices.

1. Locate the shader example project **02 Coloring** and open the `coloring.cpp` file.
2. The `vertexData[]` array in Example 4-3 on page 4-3 did not specify the face color. Example 4-7 shows how both the position and color are specified in the vertex array:

Example 4-7 Setting the vertex position and color

```
...
        GLfloat vertexData[] = {
            // FRONT
            -0.5f, -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, 1.0f,   // x,y,z, r,g,b,alpha
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, 1.0f,
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f, 0.0f, 1.0f,
             0.5f,  0.5f,  0.5f, 1.0f, 0.0f, 0.0f, 1.0f,
            // BACK
            -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            // LEFT
            -0.5f, -0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
            // RIGHT
            0.5f, -0.5f, -0.5f, 1.0f, 0.0f, 1.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f, 1.0f, 1.0f,
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f, 1.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f, 1.0f, 1.0f,
            // TOP
            -0.5f,  0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 0.0f, 0.0f, 1.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f,
            // BOTTOM
            -0.5f, -0.5f,  0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            0.5f, -0.5f,  0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 0.0f, 1.0f,
        };

        Managed<Buffer> vertexBuffer =
            context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData), sizeof(GLfloat)*7);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
        context->setVertexBuffer(0, vertexBuffer);

        VertexElement elements[2];     // two elements: xyz for vertex and rgba for color
        elements[0].components = 3;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[1].components = 4;
        elements[1].offset = 3*sizeof(GLfloat);
        elements[1].semantic = COLOR;    // the color is determined by the vertex color
        elements[1].stream = 0;
        elements[1].type = GL_FLOAT;
```

```
        Managed<VertexDeclaration> vertexDeclaration =
            context->createVertexDeclaration(elements, 2); // two elements
        context->setVertexDeclaration(vertexDeclaration);
...
```

3. Rebuild the project and run it. Figure 4-2 shows the shape with vertex coloring:

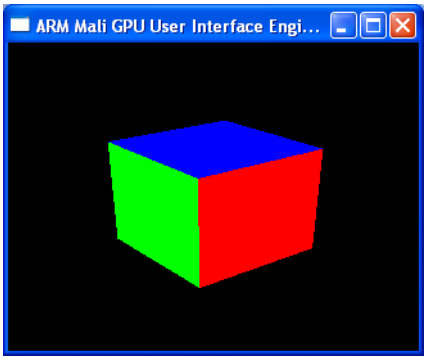

Figure 4-2 The Hello World shape with vertex coloring

4.3 Texturing the cube faces

This tutorial combines:

- drawing a 3D cube
- texturing each 2D face of the cube
- reacting to keyboard input to rotate the cube or move the camera.

1. Locate the shader example **03 - Texturing** and open the `texturing.cpp` file
2. Example 4-8 shows the standard initialization code and loading the shaders:

Example 4-8 Initalization for the textured cube

```
...
#include <mde/mde.h>
using namespace MDE;
static const float PI = 3.14159265;
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), 2.0f, radius * sin(angleRad));
}

int main(int argc, char * argv[])
{
    try
    {
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        Managed<FileSystem> filesystem = system->createFileSystem(".");

        Proxy proxy(filesystem, context);
        Program *texturingProgram =
            proxy.getProgram("shaders/texturing.vert;shaders/texturing.frag");
        Texture2D *texture = proxy.getTexture2D("data/rock_t.png");
        texture->setFilterMode(GL_NEAREST, GL_NEAREST);
...
```

3. Example 4-9 shows how to set the locations for the cube vertices and the corresponding locations in the texture bitmap.

Example 4-9 vertexData for the textured cube

```
...
        GLfloat vertexData[] = {
            // FRONT
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //xyz for vertices, uv for texture
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            // BACK
            -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
```

```
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // LEFT
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // RIGHT
            0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            // TOP
            -0.5f,  0.5f,  0.5f, 0.0f, 0.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // BOTTOM
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
           }
...
```

4. Example 4-10 shows the buffer assignments:

Example 4-10 Buffer assignments for the textured cube

```
...
        Managed<Buffer> vertexBuffer =
            context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData), sizeof(GLfloat)*5);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
        context->setVertexBuffer(0, vertexBuffer);

        VertexElement elements[2];
        elements[0].components = 3;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;
        elements[1].components = 2;
        elements[1].offset = 3*sizeof(GLfloat);
        elements[1].semantic = TEXCOORD0;
        elements[1].stream = 0;
        elements[1].type = GL_FLOAT;

        Managed<VertexDeclaration> vertexDeclaration =
            context->createVertexDeclaration(elements, 2);
        context->setVertexDeclaration(vertexDeclaration);

...
```

5. Example 4-11 shows the camera, projection, view, and world setup:

Example 4-11 Camera and view setup

```
...
        // The camera position, target and up vector for use when creating the view matrix
        float cameraAngle = 45.0f;
```

```
        vec3 camPos = calculateCamPos(4.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 1.0f, 0.0f);

        // Set up the view and projection matrices and multiply them to get the
        // modelviewprojection matrix.
        mat4 proj = mat4::perspective(35.0f, 1.0f, 0.1f, 100.0f);
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 world = mat4::identity();

        mat4 wvpMatrix = proj * view * world;

        // Set the program to use and initialize the uniforms
        context->setProgram(texturingProgram);
        context->setUniformSampler("texture", static_cast<Texture*>(texture));

        glEnable(GL_DEPTH_TEST);

...
```

6. Example 4-12 shows the draw loop which reacts to keyboard input to change the camera and view:

Example 4-12 Reacting to keyboard input to change the view

```
...
        do
        {
            // Exit the application
            if (keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
            {
                return 0;
            }

            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    camPos = calculateCamPos(4.0f, ++cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    camPos = calculateCamPos(4.0f, --cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
            }
            else
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    world *= mat4::rotation(1.0f, vec3(0.0f, 1.0f, 0.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world *= mat4::rotation(-1.0f, vec3(0.0f, 1.0f, 0.0f));
                }
            }

            wvpMatrix = proj * view * world;
```

```
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);


            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
            context->drawArrays(GL_TRIANGLE_STRIP, 0, 2); // each cube face uses four vertices with two triangles
            context->drawArrays(GL_TRIANGLE_STRIP, 4, 2); // skip to the next triangle strip
            context->drawArrays(GL_TRIANGLE_STRIP, 8, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 12, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 16, 2);
            context->drawArrays(GL_TRIANGLE_STRIP, 20, 2); // last of the six faces
        }while ( context->update() );
    }
    catch (Exception &e)
    {
        printf(e.getMessage().getCharString());
    }
}
```

7. Rebuild the project and run it. Figure 4-3 shows the shape with texturing:

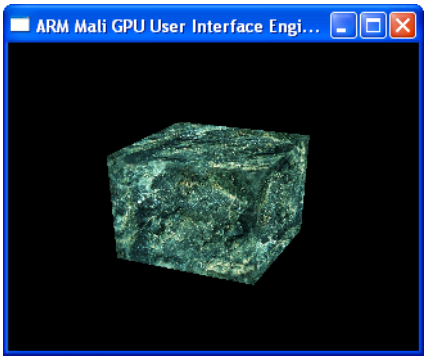

Figure 4-3 The cube shape with texturing

4.3.1 Converting the vertexBuffer and drawArrays to draw triangles

The textured cube uses triangle strips for each face of the cube. An alternative way to draw the shape is to draw each triangle individually. The triangle strip or triangle fan approach is typically faster.

Modify the example to use triangles:

1. Example 4-13 shows how to set the locations for the cube vertices and the corresponding locations in the texture bitmap.

Example 4-13 The vertexData array for the triangle array

```
...
        GLfloat vertexData[] = {
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //FRONT xyz for vertices, uv for texture
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f,    // second triangle for front face
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,   //BACK xyz for vertices, uv for texture
            -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
```

```
            0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //LEFT xyz for vertices, uv for texture
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  //RIGHT xyz for vertices, uv for texture
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 0.0f,  //TOP xyz for vertices, uv for texture
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //BOTTOM xyz for vertices, uv for texture
            -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
           }
...
```

2. Example 4-14 shows how to change the draw loop use `GL_TRIANGLES` instead of `GL_TRIANGLE_STRIPS`:

Example 4-14 Drawing GL_TRIANGLES

```
...
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
            context->drawArrays(GL_TRIANGLES, 0, 12); // draw each triangle
        }while ( context->update() );
...
```

3. Rebuild the project and run it.

4.3.2 Drawing individual triangles from the cube shape

To emphasize that the graphic consists of many separate 2D surfaces, remove some of the faces:

1. Delete every other triangle as shown in Example 4-15:

Example 4-15 Deleting triangles from the vertexData array

```
...
        GLfloat vertexData[] = {
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //FRONT xyz for vertices, uv for texture
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
           // 0.5f, -0.5f,  0.5f, 1.0f, 0.0f,    // second triangle for front face
           // -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
          //  0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
```

```
            -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,   //BACK xyz for vertices, uv for texture
            -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
             0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
           // -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
           // 0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
           // 0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //LEFT xyz for vertices, uv for texture
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
           // -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
           // -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
           // -0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 0.0f, 0.0f,  //RIGHT xyz for vertices, uv for texture
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
          //  0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
          //  0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
          //  0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 0.0f,  //TOP xyz for vertices, uv for texture
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
          //  0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
          //  -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
          //  0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,  //BOTTOM xyz for vertices, uv for texture
            -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
           // -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
           // -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
          //  0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
           }
...
```

2. Change the number of triangles as shown in Example 4-16:

Example 4-16 Drawing 11 triangles

```
...
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
            context->drawArrays(GL_TRIANGLES, 0, 6); // draw each triangle
        }while ( context->update() );
...
```

3. Rebuild the project and run it. Figure 4-4 shows the shape with half of the triangles removed:

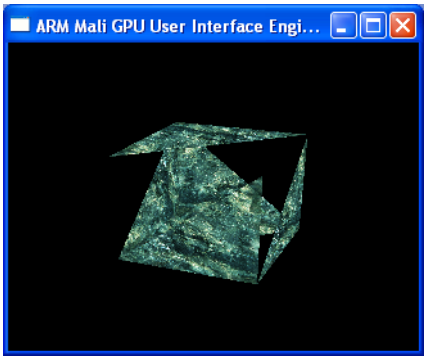

Figure 4-4 The cube shape with half of the triangles removed

The triangle is visible from both the front and back sides. The specification for a shape can select drawing only the front sides of triangles that are visible. This optimization is not covered in the 3D tutorial.

4.4 Controlling lighting effects

This tutorial combines:

- using a geometry asset, in this example it is a teapot
- selecting between per-vertex and per-pixel lighting
- reacting to keyboard input to rotate the cube or move the camera.

1. Locate the shader example **04 - Lighting** and open the `lighting.cpp` file.
2. Example 4-17 shows the standard initialization code. This example also has vectors for the light sources:

Example 4-17 Initialization for the teapot lighting application

```
...
#include <mde/mde.h>
using namespace MDE;

static const float TIMEOUT = 0.3f;

static const vec3 AMBIENT_LIGHT = vec3 (0.1f, 0.1f, 0.1f);
static const vec3 DIFFUSE_LIGHT = vec3(0.5f, 0.5f, 0.5f);
static const vec3 SPECULAR_LIGHT = vec3(0.5f, 0.5f, 0.5f);

static const vec4 COLOR = vec4(1.0f, 0.0f, 0.0f, 1.0f);

static const float PI = 3.14159265;

/**
* Function that calculates the camera position given an angle and a radius. This function
* enables the camera to rotate around the scene object.
*/
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), radius * sin(angleRad), 0.0f);
}

int main(int argc, char * argv[])
{
    try
    {
        // Initialize the demo engine classes
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        Managed<FileSystem> filesystem = system->createFileSystem("./data");
        Managed<Timer> timer = system->createTimer();

        Proxy proxy(filesystem, context);
```

3. Example 4-18 on page 4-16 shows the code that loads the lighting programs:

Example 4-18 Shaders

```
        // Loading programs
        Program* perVertexLightingProgram =
            proxy.getProgram("../shaders/per_vertex_lighting.vert;../shaders/per_vertex_lighting.frag");
        Program* perFragmentLightingProgram =
            proxy.getProgram("../shaders/per_fragment_lighting.vert;../shaders/per_fragment_lighting.frag");
...
```

There are two lighting programs, and each program has a fragment shader and a vertex shader:

Vertex lighting program

The `per_vertex_lighting.frag` fragment shader is based on the ambient, diffuse, and specular lighting as shown in Example 4-19:

Example 4-19 per_vertex_lighting.frag

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

uniform vec3 AMBIENT_LIGHT;
uniform vec4 COLOR;

varying vec3 vDiffuse;
varying vec3 vSpecular;

void main(void)
{
    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * COLOR +
                   vec4(vDiffuse, 1.0)      * COLOR +
                   vec4(vSpecular, 1.0);
}
```

The `per_vertex_lighting.vert` vertex shader is listed in Example 4-20:

Example 4-20 per_vertex_lighting.vert

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;
uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;

attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vDiffuse;
varying vec3 vSpecular;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;

    vec3 normal = normalize(WORLD * NORMAL);
```

```
    vec3 pos = WORLD * POSITION.xyz;

    vec3 lightVector = normalize(LIGHT_POSITION - pos);
    float nDotL = max(dot(normal, lightVector), 0.0);

    vDiffuse = DIFFUSE_LIGHT * nDotL;

    // No point in calculating specular reflections from the backside of vertices.
    float specPow = 0.0;
    if (nDotL > 0.0)
    {
        vec3 cameraVector = normalize(CAMERA_POSITION - pos);

        vec3 reflectVector = reflect(-cameraVector, normal);
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 4.0);
    }

    vSpecular = SPECULAR_LIGHT * specPow;
}
```

Fragment lighting program

`per_fragment_lighting.vert` for the vertex shader and `per_fragment_lighting.frag` for the fragment shader. The `per_fragment_lighting.frag` fragment shader uses the data in the normal bitmap to calculate the reflections as shown in Example 4-21:

Example 4-21 per_fragment_lighting.frag

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

uniform vec3 AMBIENT_LIGHT;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;
uniform vec4 COLOR;

varying vec3 vNormal;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    vec3 normal = normalize(vNormal);
    vec3 lightVector = normalize(vLightVector);

    float nDotL = max(dot(normal, lightVector), 0.0);
    vec3 diffuse = DIFFUSE_LIGHT * nDotL;

    float specPow = 0.0;
    if (nDotL > 0.0)
    {
        vec3 cameraVector = normalize(vCameraVector);

        vec3 reflectVector = reflect(-cameraVector, normal);
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 16.0);
    }

    vec3 specular = SPECULAR_LIGHT * specPow;

    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * COLOR +
```

```
                    vec4(diffuse, 1.0)       * COLOR +
                    vec4(specular, 1.0);
}
```

The per_fragment_lighting.vert vertex shader is listed in Example 4-22:

Example 4-22 per_fragment_lighting.vert

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;

uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;

attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vNormal;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;

    vec3 pos = WORLD * POSITION.xyz;

    // No point in normalizing before the fragment shader as the normalization will
    // be offset by the interpolation anyway.
    vNormal = WORLD * NORMAL;
    vLightVector = LIGHT_POSITION - pos;
    vCameraVector = CAMERA_POSITION - pos;
}
```

4. Previous examples specified the locations for each vertex. Example 4-23 shows how to load an asset file that contains the information:

Example 4-23 Load the asset for the teapot

```
...
        // Loading scene
        SceneAsset* scene = proxy.getSceneAsset("teapot.mba");
        GeometryAsset* teapot = static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));
...
```

Example 4-23 is loading a .mba file which enables mesh data to be compressed and conditioned. The COLLADA format source is also present in the data directory as teapot.dae. Using a GeometryAsset and the proxy loader simplifies loading complex shapes.

5. Example 4-24 on page 4-19 shows the camera, projection, view, and world setup:

Example 4-24 Set the camera, projection, view, and world for the teapot

```
...
        // The camera position, target and up vector for use when creating the view matrix
        float cameraAngle = 0.0f;
        vec3 camPos = calculateCamPos(75.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 0.0f, 1.0f);

        // Set up the view, projection and world matrices and multiply them to get the
        // modelviewprojection and modelview matrices.
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 500.0f);
        mat4 world = mat4::rotation(90.0f, vec3(0.0f, 0.0f, 1.0f));

        mat4 wvpMatrix = proj * view * world;

        bool perVertexLighting = true;
        vec3 lightPosition(75.0f, 75.0f, 0.0f);

        // Enable depth test
        glEnable(GL_DEPTH_TEST);

...
```

6. Example 4-25 shows the part of the draw loop that reacts to keyboard input to change the camera, view, and lighting:

Example 4-25 Use the to keyboard input to change the teapot view, camera, and lighting

```
...
        do
        {
            // Exit the application if escape key pressed
            if (keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
            {
                return 0;
            }

            // toggle lighting model if UP key is pressed
            static float time = timer->getTime();
            if (keyboard->getSpecialKeyState(Keyboard::KEY_UP))
            {
                perVertexLighting = !perVertexLighting;
            }

            if (perVertexLighting)
            {
                context->setProgram(perVertexLightingProgram);
            }
            else
            {
                context->setProgram(perFragmentLightingProgram);
            }

            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
```

```
                    // calculate new camera position
                    camPos = calculateCamPos(75.0f, --cameraAngle);
                    // create a view based on the new camera position
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    camPos = calculateCamPos(75.0f, ++cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
            }
            else
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    // create a world matrix to rotate the teapot
                    world *= mat4::rotation(1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world *= mat4::rotation(-1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
        }
...
```

7. Example 4-26 shows the part of the draw loop that sets the uniform values and draws the teapot:

Example 4-26 Drawing the teapot with lighting

```
...
            wvpMatrix = proj * view * world;

            // Have to set the uniforms each frame since the above call to setProgram
            // clears them.
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);
            context->setUniformMatrix("WORLD", world.toMat3());
            context->setUniform("LIGHT_POSITION", lightPosition);
            context->setUniform("CAMERA_POSITION", camPos);
            context->setUniform("AMBIENT_LIGHT", AMBIENT_LIGHT);
            context->setUniform("DIFFUSE_LIGHT", DIFFUSE_LIGHT);
            context->setUniform("SPECULAR_LIGHT", SPECULAR_LIGHT);
            context->setUniform("COLOR", ::COLOR);

            // Clear the screen and draw teapot with lighting
            glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

            teapot->draw();
        }while ( context->update() );
    }
    catch (Exception &e)
    {
        printf(e.getMessage().getCharString());
    }
}
```

8. Rebuild the project and run it. Figure 4-5 on page 4-21 shows the shape:

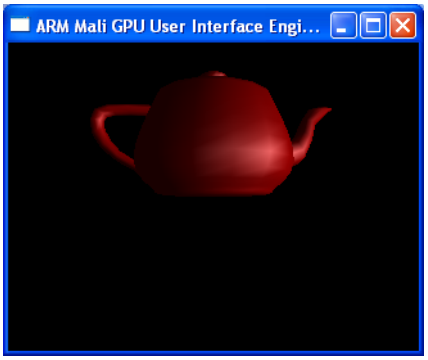

Figure 4-5 The teapot shape with lighting effects

9. Use the keyboard to change the view:
   - press the left or right arrows to rotate the teapot
   - depress the spacebar and then press the left or right arrows to move the camera
   - press the up arrow to change between Gouraud and Phong lighting.

4.4.1 Replacing the teapot with a cube

The advantage of using an asset in Mali Binary Asset format is that you can quickly load a new shape:

1. Example 4-23 on page 4-18 showed how to load the teapot asset file. Copy the `cube.mba`, `rock_n.png`, and `rock_t.png` files from `bumpmapping\data` to the `lighting\data` area.

   The `rock_n.png`, and `rock_t.png` files are not used in this example, but they are referenced by the `cube.mba` file so they must also be copied.

2. Modify the `getSceneAsset()` call to load the cube asset as shown in Example 4-27:

Example 4-27 Load the asset for the teapot

```
...
        // Loading scene
        // SceneAsset* scene = proxy.getSceneAsset("teapot.mba");
        SceneAsset* scene = proxy.getSceneAsset("cube.mba");
        GeometryAsset* teapot = static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));
...
        teapot->draw();
...
```

3. Rebuild the project and run it. Figure 4-6 on page 4-22 now shows a cube shape instead of a teapot:

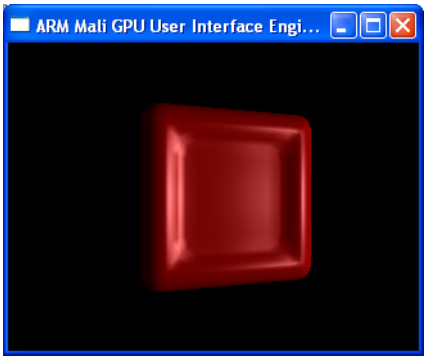

Figure 4-6 The lighting example with a cube shape instead of the teapot

4.4.2 Drawing both the teapot and the cube

This example demonstrates loading and drawing two shapes:

1. Use the modified code from *Replacing the teapot with a cube* on page 4-21 as a starting point.
2. Modify the `getSceneAsset()` call to both the load the cube and teapot assets as shown in Example 4-28:

Example 4-28 Load the assets for the cube and teapot

```
...
        // Loading scene
        SceneAsset* scene = proxy.getSceneAsset("teapot.mba");
        SceneAsset* scene2 = proxy.getSceneAsset("cube.mba");
        GeometryAsset* cube = static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));
        GeometryAsset* teapot = static_cast<GeometryAsset*>(scene2->getAsset(Asset::TYPE_GEOMETRY, 0));
...
        teapot->draw();
        cube->draw();
...
```

3. Rebuild the project and run it. Both shapes are displayed, but they overlap each other. The location and size of the objects must be modified to fit them in the display area as distinct objects.
4. Modify the world-view-projection matrix and add translation and scale multiplications as shown in Example 4-29:

Example 4-29 Moving the teapot and cube

```
...
            wvpMatrix = proj * view * world;
            // scale the teapot to 70% and move it left, down, and forward
            mat4 wvpMatrix_teapot = wvpMatrix *
                 mat4::translation(-12.0, -12.0, -12.0)* mat4::scale(0.7, 0.7, 0.7);

            // Clear the screen and set the uniforms
            ...

            teapot->draw();
```

```
        // scale the cube to 30% and move it right, up, and back
        mat4 wvpMatrix_cube = wvpMatrix *
              mat4::translation(12.0, 12.0, 12.0)* mat4::scale(0.3, 0.3, 0.30);
        context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix_cube);
        cube->draw();
    }while ( context->update() );
```

5. Rebuild the project and run it. Figure 4-7 now shows a cube shape and a teapot:

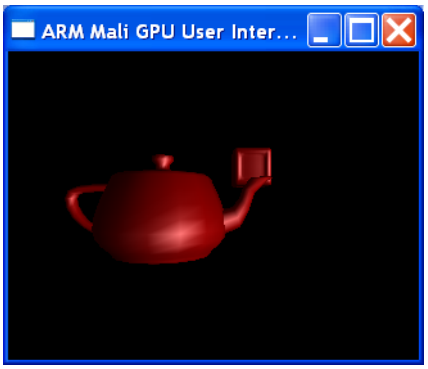

Figure 4-7 The lighting example with a cube and teapot

6. Use the arrow keys to rotate the objects and the camera. Because both objects are in the same world, they keep the same relationship and move together.

7. Create separate world matrices for the objects shown in Example 4-30:

Example 4-30 Independent worlds for the teapot and cube

```
...
        // Set up the view, projection and world matrices and multiply them to get the
        // modelviewprojection and modelview matrices.
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 500.0f);
        mat4 world_teapot = mat4::scale(0.7, 0.7, 0.7) *
                mat4::rotation(90.0f, vec3(0.0f, 0.0f, 1.0f));
        mat4 world_cube = mat4::scale(0.3, 0.3, 0.30) *
                mat4::rotation(50.0f, vec3(0.0f, 0.0f, 1.0f));

        mat4 wvpMatrix_teapot = proj * view * world_teapot;
        mat4 wvpMatrix_cube = proj * view * world_cube;
...
```

8. Modify the code that rotates the world so that the left arrow moves the teapot and the right arrow rotates the cube as shown in Example 4-31:

Example 4-31 Independent worlds for the teapot and cube

```
...
            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
               // rotate camera
            }
            }
            else
```

```
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    world_teapot *= mat4::rotation(1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world_cube *= mat4::rotation(-1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
            }
...
```

9. Modify the code that draws the shapes to use the new world-view-projections as shown in Example 4-32:

Example 4-32 Independent worlds for the teapot and cube

```
...
            wvpMatrix_teapot = proj * view * world_teapot
                 * mat4::translation(0.0, 0.0,20.0);

            // Clear the screen and set the uniforms
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix_teapot);
            context->setUniformMatrix("WORLD", world_teapot.toMat3());
           ...

            teapot->draw();

            wvpMatrix_cube = proj * view * world_cube
                 * mat4::translation(0.0, -0.0, -20.0);

            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix_cube);
            context->setUniformMatrix("WORLD", world_cube.toMat3());
            cube->draw();
        }while ( context->update() );
```

10. Rebuild the project and run it. Figure 4-8 now shows a shapes at different location on the z axis:

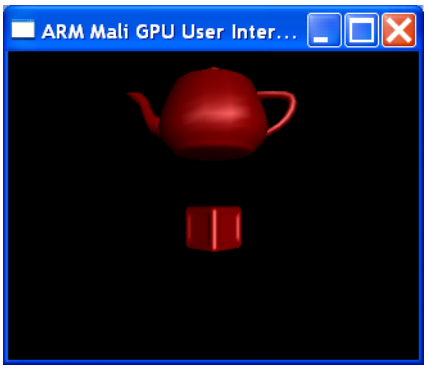

Figure 4-8 The lighting example with a cube and teapot vertically offset

4.5 Bump mapping on the cube

This tutorial combines:

- using a geometry asset, in this example it is a cube
- bump mapping
- reacting to keyboard input to rotate the cube or move the camera.

1. Locate the shader example **05 - BumpMapping** and open the `bumpmapping.cpp` file.
2. Example 4-33 shows the standard initialization code:

Example 4-33 Initialization for the Bump Mapping application

```
...
##include <mde/mde.h>
using namespace MDE;

const float AMBIENT = 0.2;
const float DIFFUSE_CONTRIBUTION = 0.7;
const float SPECULAR_CONTRIBUTION = 0.4;

static const vec3 AMBIENT_LIGHT = vec3 (0.2f, 0.2f, 0.2f);
static const vec3 DIFFUSE_LIGHT = vec3(0.7f, 0.7f, 0.7f);
static const vec3 SPECULAR_LIGHT = vec3(0.4f, 0.4f, 0.4f);

static const float PI = 3.14159265;

/**
* Function that calculates the camera position given an angle and a radius. This function
* enables the camera to rotate around the scene object.
*/
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), radius * sin(angleRad), 0.0f);
}

int main(int argc, char * argv[])
{
    try
    {
        // Initialize the demo engine classes
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        Managed<FileSystem> filesystem = system->createFileSystem("./data");
        Managed<Timer> timer = system->createTimer();

        Proxy proxy(filesystem, context);

        // Loading programs
        Program* perVertexLightingProgram =
            proxy.getProgram("../shaders/bumpmapping.vert;../shaders/bumpmapping.frag");...
```

3. Previous examples specified the locations for each vertex. Example 4-34 on page 4-26 shows how to load an asset file that contains the information:

Example 4-34 Load the assets for bump mapping

```
...
        SceneAsset* scene = proxy.getSceneAsset("cube.mba");
        GeometryAsset* teapot =
            static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));

        Texture2D *diffuseTexture =
            proxy.getTexture2DAsset("rock_t.png")->getTexture2D();

        Texture2D *normalMap =
            proxy.getTexture2DAsset("rock_n.png")->getTexture2D();
...
```

4. The two `.png` files shown in Figure 4-9 contain the texture and the normal information:

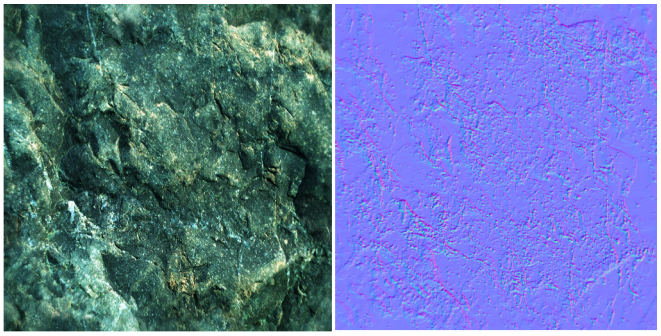

Figure 4-9 Contents of rock texture and normal files

Note

The normal file in Figure 4-9 can be displayed as a colored graphic, but the pixels actually represent three-element vectors. The x, y, and z values for the vectors are determined by the intensity of the red, green, and blue colors at that point.

5. The vertex shader computes the location for the fragment, but it also computes the angles as shown in Example 4-35:

Example 4-35 bumpmapping.vert contents

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;

// the light and camera positions are constant for this primitive shape
uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;

// the vectors are interpolated for this location on the primitive shape
attribute vec4 POSITION;
attribute vec3 NORMAL;
attribute vec3 TANGENT;
attribute vec3 BINORMAL;
attribute vec3 TEXCOORD0;

// texture coordinate to pass to fragment shader to look up the color for the fragment
varying vec2 vTexCoord;
// pass the fragment shader the light and camera vectors to this point
// these are used to calculate the effects of diffuse and specular lighting
```

```
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    // set the global that indicates this fragment position
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    // set the value to look up from the image bitmap
    vTexCoord = TEXCOORD0.st;

    // calculate the vector that identifies this point in the world projection
    vec3 pos = WORLD * POSITION.xyz;
    // calculate the normals for the point on the shape
    mat3 TBN = WORLD *
             mat3(normalize(TANGENT), normalize(BINORMAL), normalize(NORMAL));

    // No point in normalizing before the fragment shader as the normalization will
    // be offset by the interpolation anyway.

    // calculate the light vector for the point and multiply it b the world transform
    vLightVector = (LIGHT_POSITION - pos) * TBN;
    // calculate the camera vector for the point and multiply it b the world transform
    vCameraVector = (CAMERA_POSITION - pos) * TBN;
}
```

6. The fragment shader uses the contents of the `rock_n.png` file to compute the reflection as shown in Example 4-36:

Example 4-36 bumpmapping.frag contents

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

// constant values for the primitive shape
uniform mat3 M_MATRIX;
uniform sampler2D diffuseTexture;
uniform sampler2D normalMap;
uniform vec3 AMBIENT_LIGHT;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;

uniform float DIFFUSE_CONTRIBUTION;
uniform float SPECULAR_CONTRIBUTION;
uniform float AMBIENT;

// passed from the vertex shader
varying vec2 vTexCoord;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    // An optimization would be to assume that the precomputed normals in the normalMap
    // is already of a length of 1.0. That way the normalization would be unnecessary.
    vec3 normal = normalize(texture2D(normalMap, vTexCoord).rgb * 2.0 - 1.0);
    vec4 color = texture2D(diffuseTexture, vTexCoord);

    // what is the contribution of the diffuse light based on the relative
    // angle between the light and the normal read from the bitmap file
    vec3 lightVector = normalize(vLightVector);
```

```
    float nDotL = max(dot(normal, lightVector), 0.0);
    vec3 diffuse = DIFFUSE_LIGHT * nDotL;

    float specPow = 0.0;
    if (nDotL > 0.0)  // are there any specular reflections?
    {
        vec3 cameraVector = normalize(vCameraVector);

        // calculate how much specular light will be reflected
        vec3 reflectVector = reflect(-cameraVector, normal);
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 4.0);
    }
    // calculate the specular light based on the light value and the reflection value
    vec3 specular = SPECULAR_LIGHT * specPow;

    // sum all of the light sources together
    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * color +
                   vec4(diffuse, 1.0) * color +
                   vec4(specular, 1.0);
}
```

7. Example 4-37 shows setting the light and position values and passing them to the shaders:

Example 4-37 Set the camera, projection, view, and world

```
...
        float cameraAngle = 0.0f;
        vec3 camPos = calculateCamPos(75.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 0.0f, 1.0f);

        // Set up the view, projection and world matrices and multiply them to get the
        // modelviewprojection and modelview matrices.
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 500.0f);
        mat4 world = mat4::identity();

        mat4 wvpMatrix = proj * view * world;

        vec3 lightPosition(75.0f, 75.0f, 0.0f);

        // Set the program to use and initialize the uniforms
        context->setProgram(perVertexLightingProgram);
        context->setUniform("LIGHT_POSITION", lightPosition);
        context->setUniform("CAMERA_POSITION", camPos);
        context->setUniform("AMBIENT_LIGHT", AMBIENT_LIGHT);
        context->setUniform("DIFFUSE_LIGHT", DIFFUSE_LIGHT);
        context->setUniform("SPECULAR_LIGHT", SPECULAR_LIGHT);
        context->setUniformSampler("diffuseTexture", diffuseTexture);
        context->setUniformSampler("normalMap", normalMap);

        // Enable depth test and start drawing loop
        glEnable(GL_DEPTH_TEST);
...
```

8. Example 4-38 on page 4-29 shows the part of the draw loop that reacts to keyboard input to change the camera, view, and lighting:

Example 4-38 Use the to keyboard input to change the view, camera, and lighting

```
...
        do
        {
            // Exit the application if the escape key is pressed
            if (keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
            {
                return 0;
            }

            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    camPos = calculateCamPos(75.0f, --cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    camPos = calculateCamPos(75.0f, ++cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
            }
            else
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    world *= mat4::rotation(1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world *= mat4::rotation(-1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
            }
...
```

9. Example 4-39 shows the part of the draw loop that sets the uniform values and draws the teapot:

Example 4-39 Drawing the bump-mapped object

```
...
            wvpMatrix = proj * view * world;

            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);
            context->setUniformMatrix("WORLD", world.toMat3());

            // Clear the screen and draw teapot with lighting
            glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

            teapot->draw();
        }while ( context->update() );
    }
    catch (Exception &e)
    {
```

```
        printf(e.getMessage().getCharString());
    }
}
```

10. Rebuild the project and run it. Figure 4-10 shows the shape:

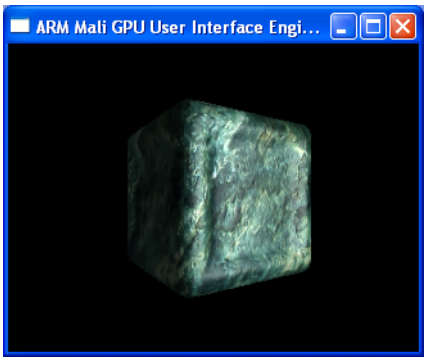

Figure 4-10 The cube shape with bump-mapped shading

11. Use the keyboard to change the view:
    - press the left or right arrows to rotate the cube
    - depress the spacebar and then press the left or right arrows to move the camera.

12. The initialization code sets both the contribution from each light source and its color:

```
const float AMBIENT = 0.2;
const float DIFFUSE_CONTRIBUTION = 0.7;
const float SPECULAR_CONTRIBUTION = 0.4;

static const vec3 AMBIENT_LIGHT = vec3 (0.2f, 0.2f, 0.2f);
static const vec3 DIFFUSE_LIGHT = vec3(0.7f, 0.7f, 0.7f);
static const vec3 SPECULAR_LIGHT = vec3(0.4f, 0.4f, 0.4f);
```

    Change the intensity of the different light sources and observe how the image changes:
    - ambient light comes from all directions and provides uniform base lighting as shown in Figure 4-11
    - diffuse light comes from a single direction and is a flat even light as shown in Figure 4-12 on page 4-31
    - specular light comes from a single direction, and reflects off the image at a specific angle between the viewer and the light source as shown in Figure 4-13 on page 4-31. Specular light reflections are typically just the color of the specular light source and not the object color. Change the color of the specular light and notice the colored reflections shown in Figure 4-14 on page 4-31.

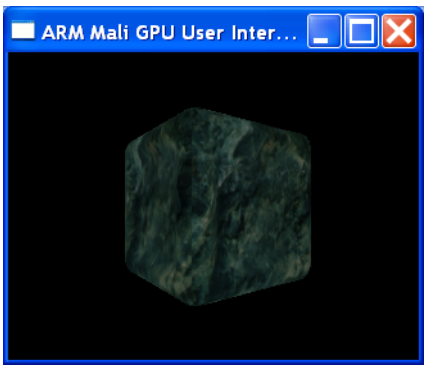

Figure 4-11 Ambient light only

Figure 4-12 Diffuse light only

Figure 4-13 Specular light only

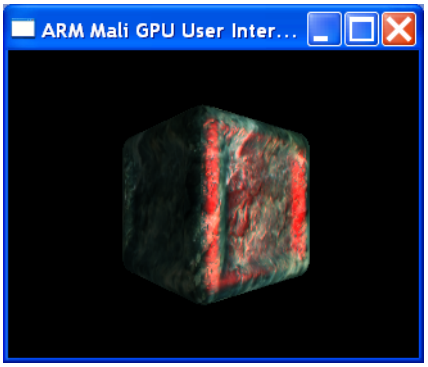

Figure 4-14 Red specular light

4.6 Cube mapping

This tutorial combines:

- using a geometry asset, in this example it is a teapot
- using mirror texturing
- creating a skybox that gives the illusion of a complex background.

Open the **8-Cube Mapping** shader example:

1. Example 4-40 shows the standard initialization code. The textureCube provides the environment that contains the shape:

Example 4-40 Initialization for the cube mapping application

```
...
#include <mde/mde.h>
using namespace MDE;

static const float PI = 3.14159265;

/**
 * Function that calculates the camera position given an angle and a radius. This function
 * enables the camera to rotate around the scene object.
 */
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), 0.0f, radius * sin(angleRad));
}

/**
 * Function that loads 6 textures from the hard drive and creates a cubemap from them.
 */
Managed<TextureCube> loadCubeMap(Managed<Context> context, Proxy &proxy)
{
    Managed<TextureCube> textureCube = context->createTextureCube();

    MDE::Bitmap2DAsset* tmpTex = NULL;

    tmpTex = proxy.getBitmap2DAsset("data/3.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0,
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    tmpTex = proxy.getBitmap2DAsset("data/opp.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0,
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    tmpTex = proxy.getBitmap2DAsset("data/4.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0,
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    tmpTex = proxy.getBitmap2DAsset("data/1.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0,
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    tmpTex = proxy.getBitmap2DAsset("data/ned.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0,
```

```
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    tmpTex = proxy.getBitmap2DAsset("data/2.png");
    textureCube->buildMipmaps(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0,
        tmpTex->getWidth(), tmpTex->getHeight(),
        tmpTex->getPixelFormat(), tmpTex->getPixels() );

    return textureCube;
}
...
```

The cube map uses the following bitmaps:

1.pgn This is the image for wall one, see Figure 4-15:

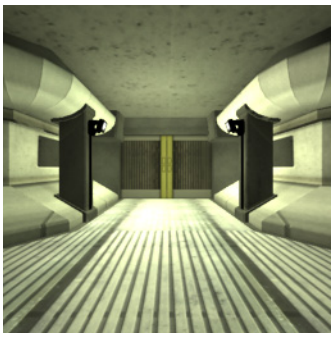

Figure 4-15 Environment bitmap 1.png

2.pgn This is the image for wall two, see Figure 4-16:

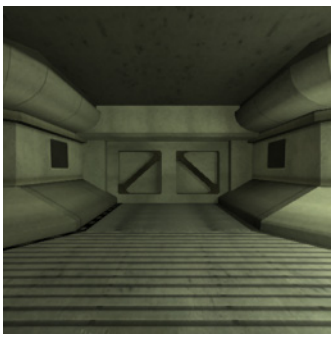

Figure 4-16 Environment bitmap 2.png

3.pgn This is the image for wall three, see Figure 4-17:

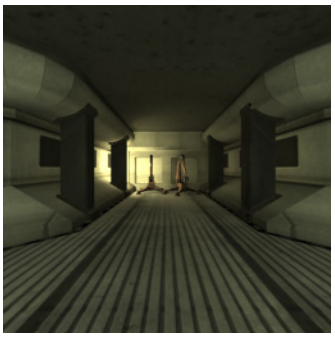

Figure 4-17 Environment bitmap 3.png

4.pgn This is the image for wall four, see Figure 4-18 on page 4-34:

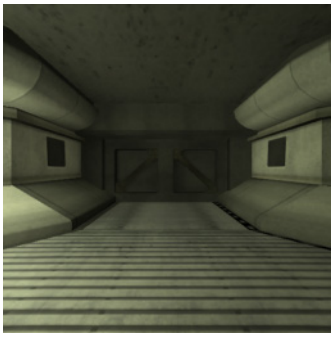

Figure 4-18 Environment bitmap 4.png

opp.pgn This is the image for the environment ceiling, see Figure 4-19:

Figure 4-19 Environment bitmap opp.png

ned.pgn This is the image for the environment floor, see Figure 4-20:

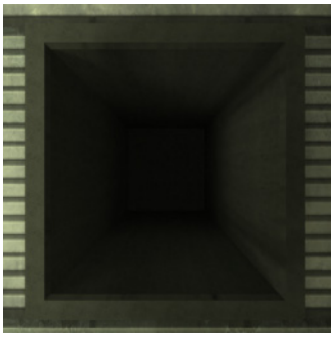

Figure 4-20 Environment bitmap ned.png

2. Previous examples specified the locations for each vertex in the shape. Example 4-41 shows how to load an asset file that contains the information:

Example 4-41 Load the assets for cube mapping

```
...
int main(int argc, char * argv[])
{
    try
    {
        // Initializing
        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
```

```
        #endif
        Managed<FileSystem> filesystem = system->createFileSystem(".");
        Managed<Timer> timer = system->createTimer();

        Proxy proxy(filesystem, context);

        // Load programs
        Program* cube_reflection =
            proxy.getProgram("shaders/cube_reflection_mapping.vert;shaders/cube_reflection_mapping.frag");

        Program* skybox = proxy.getProgram("shaders/skybox.vert;shaders/skybox.frag");

        // Load scene
        SceneAsset* scene = proxy.getSceneAsset("data/teapot.mba");
        GeometryAsset* teapot =
            static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));

        // Load cubemap
        Managed<TextureCube> textureCube = loadCubeMap(context, proxy);
        textureCube->setWrapMode(GL_CLAMP_TO_EDGE);
        textureCube->setFilterMode(GL_LINEAR, GL_LINEAR_MIPMAP_LINEAR);

...
```

3. There are four shaders:

   skybox.vert

   The vertex shader in Example 4-42 calculates the vertex coordinate to use in the fragment shader:

Example 4-42 skybox.vert shader for cube mapping

```
uniform mat4 WORLD_VIEW_PROJECTION;
attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vTexCoord;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    vTexCoord = POSITION.xyz;
}
```

   skybox.frag

   The fragment shader in Example 4-43 maps the environment bitmap to the skybox cube.

Example 4-43 skybox.frag shader for cube mapping

```
uniform samplerCube skyboxCubeMap;
varying vec3 vTexCoord;
void main(void)
{
    gl_FragColor = textureCube(skyboxCubeMap, vTexCoord);
}
```

cube_reflection_mapping.vert

The vertex code in Example 4-44 calculates the view vector based on the normal and the camera view:

Example 4-44 cube_reflection_mapping.vert shader for cube mapping

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;
uniform vec3 CAMERA_POSITION;

attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vNormal;
varying vec3 vViewVector;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;

// multiply the normal by the model matrix so that the teapot can
// be rotated
    vNormal = WORLD * NORMAL;
    vViewVector = CAMERA_POSITION - (WORLD * POSITION.xyz);
}
```

cube_reflection_mapping.frag

The code in Example 4-45 uses the normal vector to copy the cube map values to the teapot surface. This creates a mirror effect.

Example 4-45 cube_reflection_mapping.frag shader for cube mapping

```
uniform samplerCube environmentCubeMap;
varying vec3 vNormal;
varying vec3 vViewVector;

void main(void)
{
// normalize the normals as the interpolation changes their length
    vec3 normal = normalize(vNormal);
    vec3 viewVector = normalize(vViewVector);
    vec3 reflectionVector = reflect(-viewVector, normal);
    gl_FragColor = textureCube(environmentCubeMap,
        reflectionVector.xyz);
}
```

4. Example 4-46 shows the `vertexDeclaration` code. The coordinates for the skybox are entered manually:

Example 4-46 Load the assets and set the skybox parameters

```
...
        // Set up position data and index data of the skybox manually
        GLfloat vertexData[] =
        {
            -0.5f, -0.5f, -0.5f, // 0
             0.5f, -0.5f, -0.5f, // 1
```

```
             0.5f,  0.5f, -0.5f, // 2
            -0.5f,  0.5f, -0.5f, // 3
            -0.5f, -0.5f,  0.5f, // 4
             0.5f, -0.5f,  0.5f, // 5
             0.5f,  0.5f,  0.5f, // 6
            -0.5f,  0.5f,  0.5f  // 7
        };

        GLushort indexData[] =
        {
            0, 1, 2, //front
            0, 2, 3,
            1, 5, 6, //right
            1, 6, 2,
            5, 4, 7, //back
            5, 7, 6,
            4, 0, 3, //left
            4, 3, 7,
            3, 2, 6, //top
            3, 6, 7,
            4, 5, 1, //bottom
            4, 1, 0
        };

        // Set up vertex buffer and index buffer
        Buffer* vertexBuffer =
            context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData), sizeof(GLfloat)*3);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
```

5. Example 4-47 shows how to create an element buffer that contains the index lists:

Example 4-47 Creating the element buffer

```
        Buffer* indexBuffer =
            context->createBuffer(GL_ELEMENT_ARRAY_BUFFER, sizeof(indexData), sizeof(unsigned short));
        indexBuffer->setData(0, sizeof(indexData), indexData);

        // Set up vertex declaration for the vertex data stream.
        VertexElement elements[1];
        elements[0].components = 3;
        elements[0].offset = 0;
        elements[0].semantic = POSITION;
        elements[0].stream = 0;
        elements[0].type = GL_FLOAT;

        Managed<VertexDeclaration> vertexDeclaration =
            context->createVertexDeclaration(elements, 1);
...
```

To standardize creation of vertex buffers and index buffers, use `Context::createBuffer()`. When creating a buffer, specify the target as either:

- `GL_ARRAY_BUFFER` for vertex buffers that will contain vertex attributes
- `GL_ELEMENT_ARRAY_BUFFER` for index buffers that will contain index lists for `glDrawElements`.

6. Example 4-48 on page 4-38 shows the camera, projection, view, and world setup:

Example 4-48 Set the camera, projection, view, and world for cube mapping

```
...
        // The camera position, target and up vector for use when creating the view matrix
        float cameraAngle = 0.0f;
        vec3 camPos = calculateCamPos(75.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 1.0f, 0.0f);

        // Set up the view, projection and world matrices. We don't precalculate the
        // modelviewprojection-matrices because it will have to be changed for each frame
        // as we change the view matrix to rotate the camera.
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);
        mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 300.0f);

        // Set up the world matrix for the object.
        mat4 world_object = mat4::rotation(90.0, vec3(-1.0f, 0.0f, 0.0f)) *
            mat4::rotation(90.0, vec3(0.0f, 0.0f, 1.0f));
        mat4 wvpMatrix_object;

        // Set up the world matrix for the skybox.
        mat4 world_skybox = mat4::scale(200.0f, 200.0f, 200.0f);
        mat4 wvpMatrix_skybox;

        mat4 tmp;

        glEnable(GL_DEPTH_TEST);
...
```

7. Example 4-49 shows the part of the draw loop that reacts to keyboard input to change the camera, view, and lighting:

Example 4-49 Use the to keyboard input to change the cube mapping view

```
...
        do
        {
            glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

            // Exit the application
            if (keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
            {
                return 0;
            }

            // Rotate object or camera
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    camPos = calculateCamPos(75.0f, --cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    camPos = calculateCamPos(75.0f, ++cameraAngle);
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
            }
            else
```

```
            {
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    world_object *= mat4::rotation(1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
                if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    world_object *= mat4::rotation(-1.0f, vec3(0.0f, 0.0f, 1.0f));
                }
            }
...
```

8. Example 4-50 shows the part of the draw loop that sets the uniform values and draws the teapot:

Example 4-50 Drawing the teapot in the cube mapping environment

```
...
            tmp = proj * view;
            wvpMatrix_object = tmp * world_object;
            wvpMatrix_skybox = tmp * world_skybox;

            // Initialize the object's program/uniforms and draw it.
            context->setProgram(cube_reflection);
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix_object);
            context->setUniformMatrix("WORLD", world_object.toMat3());
            context->setUniform("CAMERA_POSITION", camPos);
            context->setUniformSampler("environmentCubeMap", textureCube);

            teapot->draw();
...
```

9. Example 4-51 shows the part of the draw loop that sets the uniform values and draws the environment:

Example 4-51 Drawing the cube environment

```
...

            // Initialize skybox's program/uniforms and draw it.
            context->setProgram(skybox);
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix_skybox);
            context->setUniformSampler("skyboxCubeMap", textureCube);

            context->setVertexDeclaration(vertexDeclaration);
            context->setVertexBuffer(0, vertexBuffer);
            context->drawElements(GL_TRIANGLES, 0, 12, indexBuffer);
        }while ( context->update() );
    }
    catch (Exception &e)
    {
        printf(e.getMessage().getCharString());
    }
}
```

10. Rebuild the project and run it. Figure 4-21 on page 4-40 shows two views of the shape in the environment:

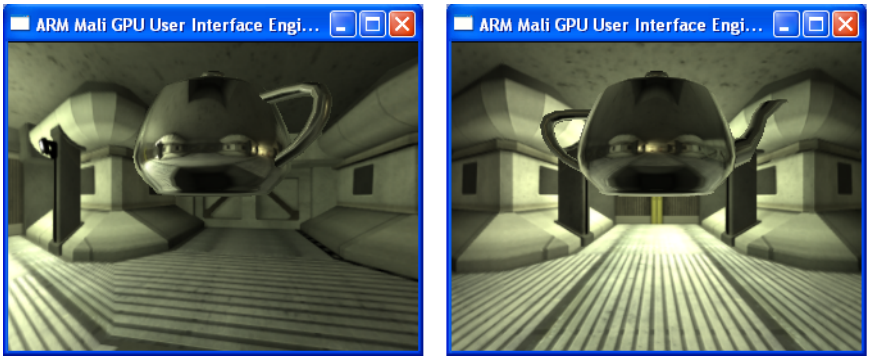

Figure 4-21 The mirrored teapot shape with cube mapping for the environment

11. Use the keyboard to change the view:
    - press the left or right arrows to rotate the teapot
    - depress the spacebar and then press the left or right arrows to move the camera

    Note

    The parallel lines on the floor are distorted in the first view in Figure 4-21. The cube environment has limitations, but it is much faster than evaluating all of the shapes that this environment would contain.

4.7 Advanced MBA scene rendering

This tutorial combines:

- a complex geometry asset
- multiple varying light sources
- a timer
- automated movement of the camera.

Open the **10-Advanced MBA Scene Rendering** example:

1. Example 4-52 shows the node traversal function:

Example 4-52 Traversal function

```
...
#include <mde/mde.h>

#define TIME_INCREMENT 0.03f
#define FIXED_FRAMERATE 1

using namespace MDE;

// the function takes as parameters: a node in the asset, the drawing context, a projection_view matrix,
// a world matrix, and the current time
void traverse(Tree<NodeAsset*>* node, Context* context, mat4 prjViw, mat4 world, float time)
{
    if (!node) return;  // no node to draw
    // find the sibling node and recursively traverse it
    traverse(node->getNextSibling(), context, prjViw, world, time);

    if (node->data)  // skip if the node does not have any data
    {
        world *= node->data->sampleLocalTransform(time);
        mat4 inv_world;
        world.invert(inv_world);  // create the inverse of the world matrix and invert it
        mat3 tiv3 = inv_world.transposed().toMat3();
        // pass the calculated uniforms to the shaders
        context->setUniformMatrix("TRA_INV_WRL", tiv3);
        context->setUniformMatrix("WORLD", world);
        context->setUniformMatrix("PRJ_VIW_WRL", prjViw*world);
        node->data->setAbsoluteTransform(world);

        for (unsigned int i = 0; i < node->data->getBatchCount(); i++)
        {
            Batch b = node->data->getBatch(i);
            if (b.material)  // calculate how the material responds to the light sources
            {
                // Check for diffuse texture
                TextureAsset* diffuse = b.material->getMap("diffuse");
                if (diffuse)
                {
                    context->setUniformSampler("TEXTURE_DIFFUSE", diffuse->getTexture());
                }

                // Check for spec level texture
                TextureAsset* spec_level = b.material->getMap("spec_level");
                if(spec_level)
                {
                    context->setUniformSampler("TEXTURE_SPEC_LEVEL", spec_level->getTexture());
                }
```

```
                // Check for ambient attribute
                Attribute* ambient = b.material->getAttribute("ambient");
                if(ambient)
                {
                    float values[4];
                    ambient->getValue(time, values);
                    context->setUniform("MATERIAL_AMBIENT", 4, 1, values);
                }

                // Check for specular attribute
                Attribute* specular = b.material->getAttribute("specular");
                if(specular)
                {
                    float values[4];
                    specular->getValue(time, values);
                    context->setUniform("MATERIAL_SPECULAR", 4, 1, values);
                }

                // Check for shininess attribute
                Attribute* shininess = b.material->getAttribute("shininess");
                if(shininess)
                {
                    float value;
                    shininess->getValue(time, &value);
                    context->setUniform("MATERIAL_SHININESS", value);
                }


            }
            if (b.geometry)
            {
                // draw the geometry
                b.geometry->draw();
            }
        }
    }
    // continue traversal over the child nodes
    traverse(node->getFirstChild(), context, prjViw, world, time);
}
...
```

2. Example 4-53 shows the initialization and loading of the asset files:

Example 4-53 Creating the context and loading the scene assets

```
...
int main()
{
    try
    {
        Managed<System> system = create_system_backend();
        Managed<FileSystem> filesystem = system->createFileSystem("data/");
        Managed<Context> context = system->createContext(320, 240);
        #ifdef MDE_OS_PC_LINUX
            Managed<Keyboard> keyboard = system->createKeyboard(context);
        #else
            Managed<Keyboard> keyboard = system->createKeyboard();
        #endif
        Managed<Timer> timer = system->createTimer();
        Proxy proxy(filesystem, context);
```

```
        SceneAsset* scene = proxy.getSceneAsset("lightshow2_eksempelscene.MBA");
        Program* program = proxy.getProgramAsset("default.vert;default.frag")->getProgram();
        context->setProgram(program);
...
```

3. Example 4-54 shows the camera and lighting setup:

Example 4-54 Set the camera and light values

```
...
        //int numCams = scene->getAssetCount(Asset::TYPE_CAMERA);
        int currentCam = 0;
        float camFov = 270.0f;
        float camAspect = 4.0f/3.0f;
        float time = timer->getTime();

        vec3 lightPos(0.0f, 0.0f, 0.0f);
        vec3 camTarget(0.0f, 0.0f, 0.0f);
        vec3 camPos(0.0f, 0.0f, 0.0f);

        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);
        glEnable(GL_DEPTH_TEST);
...
```

4. Example 4-55 shows the part of the draw loop that manages the camera:

Example 4-55 Getting the camera asset

```
...
        do
        {
#if FIXED_FRAMERATE == 1
            time += TIME_INCREMENT;
#else
            time = timer->getTime();
#endif
            ::printf("%f\n", time);
            glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);


            // Get camera asset
            CameraAsset* cam = (CameraAsset*)scene->getAsset(Asset::TYPE_CAMERA, currentCam);
            if (cam)
            {
                // Find nodes in camera:
                Array<MDE::Tree<MDE::NodeAsset*>*> nodes;
                scene->findAssetNodes(cam, scene->getTree(0), nodes);

                // Set camera and target node:
                NodeAsset* camNode = nodes[0]->data;
                NodeAsset* targetNode = cam->getTargetNode();

                // Set camera position
                if(camNode)
                    camPos = camNode->getAbsolutePosition();

                // Set target node position
```

```
            if(targetNode)
                camTarget = targetNode->getAbsolutePosition();

            // Get camera FOV
            Attribute* fovAttribute = camNode->getCamera(0)->getAttribute("xfov");
            if(fovAttribute && fovAttribute->getComponentCount() == 1)
                fovAttribute->getValue(time, &camFov);

            // Get camera aspect
            Attribute* aspectAttribute = camNode->getCamera(0)->getAttribute("aspect");
            if(aspectAttribute && aspectAttribute->getComponentCount() == 1)
                aspectAttribute->getValue(time, &camAspect);

            context->setUniform("CAMERA_POSITION", 3, 1, camPos);
            // Terminate on pressing key ESCAPE
            if(keyboard->getSpecialKeyState(Keyboard::KEY_ESCAPE))
                break;
        }
...
```

5. Example 4-56 shows the part of the draw loop that modifies the lighting:

Example 4-56 Modifying the lighting

```
...
        // For each light
        for(int i = 0; i < 3; i++)
        {
            LightAsset* light = (LightAsset*)scene->getAsset(Asset::TYPE_LIGHT, i);
            if (light)
            {
                Array<MDE::Tree<MDE::NodeAsset*>*> nodes;
                scene->findAssetNodes(light, scene->getTree(0), nodes);

                NodeAsset* lightNode = nodes[0]->data;
                if (lightNode)
                {
                    vec3 lightPos = lightNode->getAbsolutePosition();
                    char buffer[255];

                    // Get and set attributes for light:
                    for(unsigned int j = 0; j < light->getAttributeCount(); j++)
                    {
                        // Check for the color attribute
                        Attribute* attr = light->getAttribute(j);
                        if(attr->name == "color")
                        {
                                float values[3];
                                attr->getValue(time, values);
                                sprintf(buffer, "LIGHT_COLOR[%i]", i);
                                context->setUniform(buffer, 3, 1, values);
                        }
                        // Check for the far attenuation start attribute
                        else if(attr->name == "far_attenuation_start")
                        {
                                float value;
                                attr->getValue(time, &value);
                                sprintf(buffer, "LIGHT_FAR_ATTENUATION_START[%i]", i);
                                context->setUniform(buffer, value);
                        }
                        // Check for the fat attenuation end attribute
```

```
                        else if(attr->name == "far_attenuation_end")
                        {
                                float value;
                                attr->getValue(time, &value);
                                sprintf(buffer, "LIGHT_FAR_ATTENUATION_END[%i]", i);
                                context->setUniform(buffer, value);
                        }
                    }

                    // Set light position
                    sprintf(buffer, "LIGHT_POSITIONS[%i]", i);
                    context->setUniform(buffer, 3, 1, lightPos.coord);
                }
            }
        }
...
```

6. Example 4-57 shows the world, view, and project calculation and scene traversal:

Example 4-57 Traversing the scene

```
...
            mat4 world = mat4::identity();
            mat4 view = mat4::lookAt( camPos, camTarget, vec3(0.0f, 0.0f, 1.0f) );
            mat4 proj = mat4::perspective(camFov, camAspect, 1.0f, 1000.0f);
            mat4 prjViw = proj*view;
            //context->setUniformMatrix("WORLD", world);
            //context->setUniformMatrix("PRJ_VIW_WRL", prjViw*world);
            //context->setUniformMatrix("PROJECTION", 4, 1, proj);

            traverse(scene->getTree(0), context, prjViw, world, time);
        }while (context->update());
    }
    catch(Exception& e)
    {
        printf("An exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

7. Rebuild the project and run it. Figure 4-22 shows the shape:

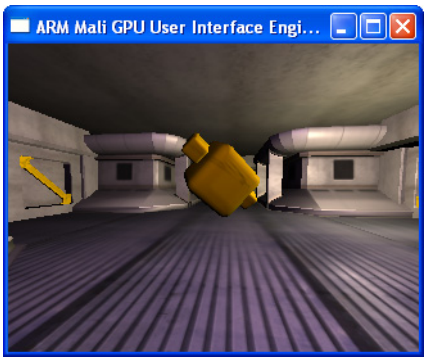

Figure 4-22 The advanced mba shape with tree traversal

The asset contains:

- all of the shaders
- the environment bitmaps

- the geometry for the central rotating shape
- the camera path
- the lighting sources.

The asset file is organized in nodes. The application calls the `traverse()` function to calculate the current position and orientation for the central shape and the camera.

4.8 Lightshow animation with moving sprite and camera

This tutorial combines:

- a complex geometry asset
- multiple varying light sources
- frame rate calculation
- automated movement of the camera
- automated movement of a simple shape.

Open the **Lightshow** example application:

1. Example 4-58 shows the code in `main.cpp` that draws the flare:

Example 4-58 Drawing the flare

```
...
void prepareFlares(Context* context, Program* p)
{
    float texcoords[] = { 0,0, 1,0, 1,1, 0,1 };
    flareTexcoord = context->createBuffer(GL_ARRAY_BUFFER, sizeof(texcoords), 8);
    flareTexcoord->setData(0, sizeof(texcoords), texcoords);

    flareTexcoordsLoc = p->getAttribLocation(TEXCOORD0);

}

void drawFlare(Context* context, vec3 position, vec3 color, float size)
{
    context->setUniform("position", position);
    context->setUniform("color", color);
    context->setUniform("size", size);

    glEnableVertexAttribArray(flareTexcoordsLoc);
    glBindBuffer(GL_ARRAY_BUFFER, flareTexcoord->getHandle());
    glVertexAttribPointer(flareTexcoordsLoc, 2, GL_FLOAT, GL_FALSE, 0, 0);


    unsigned short indices[] =
    {
        0, 1, 2,
        0, 2, 3
    };

    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, indices);

    glDisableVertexAttribArray(flareTexcoordsLoc);
}
...
```

2. Example 4-59 shows the code that traverses the tree that contains the paths:

Example 4-59 Path traversal functions

```
void traverseTransform(Tree<NodeAsset*>* node, mat4 parentTransform, float time)
{
    if (!node) return;
    traverseTransform(node->getNextSibling(), parentTransform, time);
    if (node->data)
```

```
    {
        mat4 localTransform = node->data->sampleLocalTransform(time);
        parentTransform = localTransform * parentTransform;
        node->data->setAbsoluteTransform(parentTransform);
    }
    traverseTransform(node->getFirstChild(), parentTransform, time);
}

void applyMaterial(MaterialAsset* m, Context* context)
{
    if (m)
    {
        if (m->commonMaps.diffuse) context->setUniformSampler("diffuseMap", m->commonMaps.diffuse->getTexture());

        if (m->commonMaps.bump) context->setUniformSampler("normalMap", m->commonMaps.bump->getTexture());

        if (m->commonMaps.spec_level)
            context->setUniformSampler("glossMap", m->commonMaps.spec_level->getTexture());
    }
}
void traverseDraw(Tree<NodeAsset*>* node, Context* context)
{
    if (!node) return;
    traverseDraw(node->getNextSibling(), context);
    if (node->data)
    {
        mat4 world = node->data->getAbsoluteTransform();
        context->setUniformMatrix("WORLD", world);

        mat4 inv_trans_world = world.transposed();
        inv_trans_world.invert(inv_trans_world);
        context->setUniformMatrix("INV_TRANS_WORLD", inv_trans_world);


        for (unsigned int i = 0; i < node->data->getBatchCount(); i++)
        {
            Batch b = node->data->getBatch(i);
            applyMaterial(b.material, context);
            b.geometry->draw();
        }
    }
    traverseDraw(node->getFirstChild(), context);
}
...
```

3. Example 4-60 shows the initialization and loading of the asset files:

Example 4-60 Creating the context and loading the scene assets

```
...
int main(int argc, char** argv)
{
    t_args args;
    int width = 640;
    int height = 480;
    static int frameCount = 1;
    float current_time = 0.0;
    float old_time = 0.0;
    static int first_time = 1;

    if ( 0 != parse_arguments( argc, argv, &args )) exit(1);
```

```
    /* Note that the following are already swapped in case of user input = -rotated */
    if ( args.height != 0 ) height = args.height;
    if ( args.width  != 0 ) width  = args.width;

    try
    {
        // Run from the directory containing the executable, so that we can find the data wherever it is run from
        cdToExecutable();

        Managed<System> system = create_system_backend();
        Managed<Context> context = system->createContext(width, height);
        Managed<FileSystem> fs = system->createFileSystem("data");
        Managed<Timer> timer = system->createTimer();
        Managed<Timer> fps_timer = system->createTimer();

        Proxy proxy(fs, context);

        SceneAsset* scene = proxy.getSceneAsset("lightshow.MBA");
        Program* flarePrg = proxy.getProgram("flare.vert;flare.frag");
        Program* prg = proxy.getProgram("lightshow.vert;lightshow.frag");
        NodeAsset* camNode = (NodeAsset*)scene->getAsset(Asset::TYPE_NODE, "Camera01-node");
        NodeAsset* targetNode = (NodeAsset*)scene->getAsset(Asset::TYPE_NODE, "Camera01.Target-node");
        NodeAsset* light0Node = (NodeAsset*)scene->getAsset(Asset::TYPE_NODE, "Omni01-node");
        NodeAsset* light1Node = (NodeAsset*)scene->getAsset(Asset::TYPE_NODE, "Omni02-node");

        Texture* flareTex = proxy.getTexture2D("flare.png");

        glEnable(GL_DEPTH_TEST);

...
```

4. Example 4-61 shows start of the draw loop and the code that calculates and displays the current frame rate:

Example 4-61 Draw loop with frame rate calculation

```
...
        float time = 0.0;

        do
        {
            /*
               If FPS measurement is required and start_frame has been entered and current frame is the
               start_frame then store current time only once
                OR
               if FPS measurement is required and start_frame has not been entered then store current
               time every frame
            */
            if ( args.print_fps && ( args.start_frame == 0))
            {
                current_time = fps_timer->getTime();
            }

            if ( args.print_fps && ( args.start_frame != 0) && ( frameCount == args.start_frame ))
            {
                old_time = fps_timer->getTime();
            }

            if( first_time )
            {
```

```
            first_time=0;
            old_time=current_time;
        }

        if ( 0 != args.start_frame && ( frameCount >= ( args.start_frame + args.number_of_frames )))
        {
            break;
        }

        if ( 0 != args.framerate )
            time += (float)1/args.framerate;
        else
            time = timer->getTime()*0.3f;

        /* Measure each frame if start_frame was not entered */
        if ( args.print_fps  && ( args.start_frame == 0))
        {
            printf("Frame %d: ", frameCount);
            printf("%.3f fps\n", 1/(current_time-old_time));
            old_time = current_time;
        }
        else
            printf("Frame %d\n", frameCount);


        float scenetime = time;
        while (scenetime > 10) scenetime -= 10;
...
```

5. Example 4-62 shows the camera, lighting, and environment code:

Example 4-62 Set the camera and drawing the environment

```
...
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

        mat4 rootTransform = mat4::identity();
        traverseTransform(scene->getTree(0), rootTransform, scenetime);
        vec3 camPos = camNode->getAbsolutePosition();
        vec3 targetPos = targetNode->getAbsolutePosition();

        mat4 view = mat4::lookAt(camPos, targetPos, vec3(0,0,1));
        mat4 proj = mat4::perspective(80, 1.33333f, 1, 500);

        context->setProgram(prg);
        context->setUniformMatrix("VIEW_PROJECTION", proj*view);

        vec3 light0position = light0Node->getAbsolutePosition();
        vec3 light0color(0.66f + fabs(sinf(time * 0.1f)) * 0.4f,
              0.66f + fabs(sinf(time * 0.12f + 0.4f))*0.4f,
              0.66f + fabs(sinf(time * 0.18f + 0.989f)) * 0.4f);
        context->setUniform("light0position", light0position);
        context->setUniform("light0color", light0color);

        vec3 light1position = light1Node->getAbsolutePosition();
        vec3 light1color(0.66f + fabs(sinf(time * 0.1f)) * 0.4f,
              0.66f+fabs(sinf(time*0.12f+0.14f))*0.4f,
              0.66f+fabs(sinf(time*0.18f+0.389f))*0.4f);
        context->setUniform("light1position", light1position);
        context->setUniform("light1color", light1color);
```

```
            context->setUniform("cameraPosition", camPos);

            float cyl_radius = sqrt(30.0f);

            vec3 shadow_info[2] =
            {
              calculate_cyl_info(vec2(-30.0, -30.0), cyl_radius, vec2(light0position.x, light0position.y), 0.0),
              calculate_cyl_info(vec2( 30.0,  30.0), cyl_radius, vec2(light0position.x, light0position.y), 0.0)
            };

            context->setUniform("shadow_info[0]", shadow_info[0]);
            context->setUniform("shadow_info[1]", shadow_info[1]);


            traverseDraw(scene->getTree(0), context);
...
```

6. Example 4-63 shows the vertex shader code for the environment:

Example 4-63 Environment vertex shader lightshow.vert

```
...
uniform mat4 WORLD;
uniform mat4 INV_TRANS_WORLD;
uniform mat4 VIEW_PROJECTION;
uniform vec3 cameraPosition;

uniform vec3 light0position;
uniform vec3 light1position;
uniform vec3 light2position;

attribute vec4 POSITION;
attribute vec2 TEXCOORD0;
attribute vec3 NORMAL;
attribute vec3 BINORMAL;
attribute vec3 TANGENT;

varying vec2 outTexcoord;

varying vec3 oPos;
varying vec3 tViewDir;

/* lights */
varying vec3 oLight0Dir;
varying vec3 tLight0Dir;
varying vec3 oLight1Dir;
varying vec3 tLight1Dir;

void main()
{
    oPos = (WORLD * POSITION).xyz;
    gl_Position = VIEW_PROJECTION * vec4(oPos, 1);

    outTexcoord.xy = TEXCOORD0 * vec2(1,-1);

    mat3 m = mat3(INV_TRANS_WORLD[0].xyz, INV_TRANS_WORLD[1].xyz,
                  INV_TRANS_WORLD[2].xyz);
    mat3 tangentSpace = mat3(m * TANGENT, m * BINORMAL, m * NORMAL);

    vec3 oViewDir = oPos - cameraPosition;
```

```
    oLight0Dir = light0position - oPos;
    tLight0Dir = oLight0Dir * tangentSpace;

    oLight1Dir = light1position - oPos;
    tLight1Dir = oLight1Dir * tangentSpace;

    tViewDir = oViewDir * tangentSpace;
}
```

7. Example 4-64 shows the fragment shader code for the environment:

Example 4-64 Environment fragment shader lightshow.frag

```
...
varying vec3 oPos;
varying vec3 tViewDir;
varying vec2 outTexcoord;

varying vec3 oLight0Dir;
varying vec3 tLight0Dir;
varying vec3 oLight1Dir;
varying vec3 tLight1Dir;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D glossMap;

uniform vec3 cameraPosition;
uniform vec3 light0color;
uniform vec3 light1color;
uniform vec3 light2color;
uniform vec3 light0position;

const float pi = 3.1415;

uniform vec3 shadow_info[2];
const float soft_shadow_eps = 0.015;

// calculate how the light source intersects the column
float intersectRayCylinder(vec2 ray_origin, vec3 shadow_info)
{
    const float scale_factor = (0.5/pi);

    float offset     = shadow_info.x;
    float light_dist = shadow_info.y;
    float limit = shadow_info.z;

    vec2 light_pixel = ray_origin.xy - light0position.xy;
    float angle = fract(atan(light_pixel.y, light_pixel.x)*scale_factor - offset);
    float adjusted_angle = abs(angle - 0.5);

    return smoothstep(limit, limit+soft_shadow_eps, adjusted_angle) *
                      float(light_dist < length(light_pixel));
}

// is this point in shadow
float getShadow(vec2 oPos)
{
    float shadow = 0.0;
    shadow = max(shadow, intersectRayCylinder(oPos, shadow_info[0]));
    shadow = max(shadow, intersectRayCylinder(oPos, shadow_info[1]));
```

```
    shadow = 1.0 - shadow;

    shadow = max(0.0, shadow);
    shadow = min(1.0, shadow);

    return shadow;
}

vec2 calcLight(vec3 tLightDir, vec3 oLightDir, vec3 oPos, vec3 tNormal,
               vec3 tRefViewDir, vec2 texcoord)
{
    float lightDist = length(tLightDir);
    tLightDir = normalize(tLightDir);

    float l_dot_n = max(dot(tNormal,     tLightDir), 0.0);
    float h_dot_n = max(dot(tRefViewDir, tLightDir), 0.0);

    float attenuation = max(0.0, 120.0 - lightDist) / 90.0;
    attenuation *= attenuation;

    // use the shadow value to calculate the diffuse and specular light
    float shadow = getShadow(oPos.xy) * attenuation;
    float diffuse = l_dot_n * shadow;
    float specular = pow(h_dot_n, 50.0) * shadow;

    return vec2(diffuse, specular);
}

void main(void)
{
    vec2 texcoord = outTexcoord.xy;
    vec3 tNormal = normalize((2.0*texture2D(normalMap, outTexcoord.xy).xyz)-1.0);

    vec3 tViewDirNorm = normalize(tViewDir);
    vec3 tRefViewDir  = reflect(tViewDirNorm, tNormal);

    vec3 diffuseLight = vec3(0.0, 0,0);
    vec3 specularLight = vec3(0,0,0);

    {
        vec3 lightColor = light0color;
        vec2 diffuseSpecular = calcLight(tLight0Dir, oLight0Dir, oPos, tNormal,
                                         tRefViewDir, texcoord);
        diffuseLight.xyz += lightColor * diffuseSpecular.x;
        specularLight.xyz += lightColor * diffuseSpecular.y * texture2D(glossMap,
                             texcoord).x;
    }

    vec3 texDiffuse = texture2D(diffuseMap, outTexcoord.xy).xyz;
    gl_FragColor = vec4(diffuseLight*texDiffuse + specularLight,1);
}
```

8. Example 4-65 shows the code that sets the uniforms for the flare and calls `drawFlare()`:

Example 4-65 Drawing the flare

```
...

            glDepthMask(GL_FALSE);
            glEnable(GL_BLEND);
            glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

```
            // specify flare.vert and flare.frag as the active shaders
            context->setProgram(flarePrg);
            context->setUniformMatrix("VIEW", view);
            context->setUniformMatrix("PROJECTION", proj);
            context->setUniformSampler("colorTex", flareTex);
            // call drawFlare with the light position, color, and flare size
            drawFlare(context, light0position, light0color, 10);
            glDepthMask(GL_TRUE);
            glDisable(GL_BLEND);

            frameCount++;
        }while (context->update());

    catch(Exception& e)
    {
        printf("An exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

9. Example 4-66 shows the vertex shader code for the flare:

Example 4-66 Flare vertex shader flare.vert

```
...
uniform mat4 VIEW;
uniform mat4 PROJECTION;
uniform float size;
uniform vec3 position;

attribute vec2 TEXCOORD0;

varying vec2 outTexcoord;

void main(void)
{
    // calculate the location of the flare in the world view
    vec4 p = VIEW * vec4(position, 1.0);
    // increase the size of the flare
    // shift by -size to +size depending on value of TEXCOORD0
    p.xy += (TEXCOORD0 * 2.0 - 1.0) * size;
    // calculate the display position
    gl_Position =  PROJECTION * p;
    // forward the text coordinate to the fragment shader
    outTexcoord = TEXCOORD0;
}
```

10. Example 4-67 shows the fragment shader code for the flare:

Example 4-67 Flare fragment shader flare.frag

```
...
uniform sampler2D colorTex;
uniform vec3 color;

void main(void)
{
```

```
    // look up color in texture map and multiply by the color uniform
    gl_FragColor =  vec4(texture2D(colorTex, outTexcoord).xyz * 2.0 * color,1);
}
```

11. Rebuild the project and run it. See Figure 4-23:

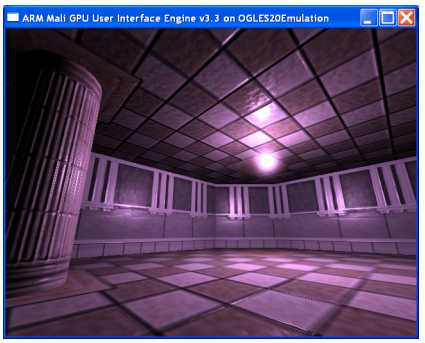

Figure 4-23 The Lighting image with moving camera and object

The asset contains:

- all of the shaders
- the environment bitmaps
- the path for the moving flare
- the camera path
- the lighting sources.

The asset file is organized in nodes. The application calls the `traverseDraw()` and `drawFlare()` functions to calculate the current positions and draw the environment and flare.

12. Modify the flare code in `main.cpp` as shown in Example 4-68 to make the flare oscillate in size:

Example 4-68 Modifying the flare

```
...
            glDepthMask(GL_FALSE);
            glEnable(GL_BLEND);
            glBlendFunc(GL_SRC_ALPHA, GL_ONE);
            // specify flare.vert and flare.frag as the active shaders
            context->setProgram(flarePrg);
            context->setUniformMatrix("VIEW", view);
            context->setUniformMatrix("PROJECTION", proj);
            context->setUniformSampler("colorTex", flareTex);
            // call drawFlare with the light position, color, and flare size
            // drawFlare(context, light0position, light0color, 10);
            // vary the size of the flare based on the frame number
            drawFlare(context, light0position, light0color, (5+3*sin(frameCount*.020)));
            glDepthMask(GL_TRUE);
            glDisable(GL_BLEND);

            frameCount++;
        }while (context->update());

    catch(Exception& e)
    {
```

```
        printf("An exception was thrown:\n%s\n", e.getMessage().getCharString() );
    }
}
```

13. Rebuild and run the application. The flare now varies in size.

Chapter 5
Tutorial on the Lotion User Interface Classes

This chapter provides information about the user interface classes provided with the Lotion example code. It contains the following sections:

5.1 Overview of the Lotion source code

This section describes the code organization and execution sequence:

- *Directory organization*
- *Overview of initialization in main.cpp* on page 5-6
- *Interaction between the Lotion classes* on page 5-7.

5.1.1 Directory organization

This section describes the files and directories that contain the source code for the lotion example.

Lotion directory

The following files are in the `lotion` directory:

- `applet.cpp` and `applet.h` are for applet instantiation and drawing. See also *applets subdirectory* on page 5-3.
- `calibration_applet.h` contains code for calibrating touch screens. The user taps on each of the corner of the screen and the software adjusts the display coordinates.
- `application.cpp` and `application.h` to provide the user interface and manage applet selection and display. These are used by `main.cpp`.
- `common.h` contains some enumerations and utility code.
- `double_transition.h` contains code for fading one image in while another image is faded out. provide the user interface and manage applet selection and display.
- `etc_bitmap_demo_loader.cpp` and `etc_bitmap_demo_loader.h` load a bitmap asset from a proxy.
- `etc_texture_demo_loader.cpp` and `etc_texture_demo_loader.h` load a pkm asset from a proxy.
- `lotion.h` contains the `#include` statements for the header files.
- `main.cpp` reads the command-line parameters and initializes the application.
- `radio_button_group.cpp` and `radio_button_group.h` add or remove radio buttons from a widget. The widget and theme code implement the radio button events.
- `render_target.cpp` and `render_target.h` manage the render buffers.
- `theme.h` contains the enumeration of the applets and the declaration of the virtual `Theme` class.
- `transition.cpp` and `transition.h` describe which special transitions are available. No special transitions are enabled in this example.
- `widget.cpp` and `widget.h` manage the creation and update of widgets.
- `widget_group.cpp` and `widget_group.h` groups applets with the application.

3rdparty subdirectory

The `3rdparty` subdirectory contains open-source third-party libraries:

etcpack This subdirectory contains the code that supports the Ericsson Texture Compression Codec standard.

freetype2 This subdirectory contains files from the FreeType project, and is for font display.

applets subdirectory

The `applets` subdirectory contains the user applets that run in the lotion environment:

browser A shell application that can be extended to imitate a browser.

buzzword This is an interactive graphics display. The viewer can change how the graphic is displayed.

dummy A shell application

lightshow This is an animation of a moving flare and camera. The viewer can select the rendering parameters.

photo This is a photo album applet.

planet This is a demonstration of different shader effects.

presentation This is a slideshow presentation with animated transitions between slides.

slideshow_presentation
This is a slideshow presentation with timed transitions between slides.

data subdirectory

The `data` subdirectory contains code and data that is used by the applications:

applets This directory contains shaders and bitmaps for the applets.

shared This directory contains compressed shaders.

themes This directory contains user interface themes for the applets.

transitions This directory contains files that manage the transition from one page to another.

widgets This directory contains user interface components such as buttons and labels.

double_transitions subdirectory

The `double_transitions` subdirectory contains code that manages different rendered layers:

page This directory contains code that switches between pages.

transition_wrapper
This directory contains code that assigns the fade-in an fade-out transitions.

multi_touch subdirectory

The `multi_touch` subdirectory contains code for input devices:

keyboard_device.cpp
The KeyboardDevice class assigns a keyboard to a context.

mouse_device.cpp

The MouseDevice class assigns a mouse and observer to a context.

mouse_keyboard_device.cpp

The MouseKeyboardDevice class assigns a keyboard to a context. Key presses are interpreted as mouse events.

multi_touch

multi_touch.h contains the declarations for the virtual Multitouch class.

render_engine subdirectory

The `render_engine` subdirectory contains code that manages renders the layer to the drawing window:

bitmap2d_blur

The Bitmap2Blur class blurs an image.

bitmap2d_collage

The Bitmap2DCollage class creates a collage from multiple images.

etc_bitmap_compression

These files uses Mali Texture Compression to write an image to a stream.

font The Font class manages writing a text string to a context.

hash_map The HashMap class creates hash maps. Values can be retrieved by specifying their key.

instance The Instance class is used with the InstanceGeometry class to assign primitive shapes to a layer.

instance_geometry

The InstanceGeometry class is used with the Instance class to position primitive shapes on a layer.

layer The Layer class is used with the RenderEngine class to display graphic layers.

quad The Quad class is used with the Layer class to display graphic primitives.

rectangle The Rectangle class defines the size and location of a rectangle.

render_engine

The RenderEngine class, with the Layer and Instance classes, manage the graphics overlays.

text The Text class describes the text string and bounding rectangle for a text block that will be rendered to a layer.

themes subdirectory

The `themes` subdirectory contains code that provides alternative ways to construct the user interface:

blue The Blue class inherits from Theme and provides user-interface controls such as buttons, check boxes, radio buttons, labels, panels, and sliders.

minimal The Minimal class inherits from Theme and provides very basic graphic elements. Some of the controls in the Blue theme are used.

robot The Robot class inherits from Theme and provides animated user-interface controls.

The `lotion` directory contains the `theme.h` file that contains the pure virtual declarations for the Theme class.

transitions subdirectory

The `transitions` subdirectory contains code that provides alternative ways to switch between different images:

cube The Cube class defines the size and location of a cube and associated transition actions.

grid The Grid class defines the size and location of a grid and associated transition actions.

linear_stretch

The LinearStretch class defines transition actions based on distorting a shape.

page The Page class defines the size and location of a page and associated transition actions.

puzzle The Puzzle class defines the size and location of a puzzle shape and associated transition actions.

slime The Slime class defines the size and location of a irregular shape and associated transition actions.

widgets subdirectory

The `widgets` subdirectory contains code that provides user-interface controls and manages the background for the user applications:

skybox The Skybox class defines the size and location of a texture cube and associated texturing actions.

background

The Background class is an empty placeholder.

button The Button class defines the text and enable state for a button.

checkbox

The Checkbox class inherits from Button defines the checked and enabled state of a checkbox widget.

composite_frame

The CompositeFrame class is a widget that contains multiple quads.

label

The Label class is a widget for displaying text.

custom_label

The CustomLabel class inherits from Label and defines how the label is displayed.

fps_counter The FpsCounter class is a virtual definition that has a single public virtual method that returns the frame rate.

radio_button

The RadioButton class inherits from Button defines the checked and enabled state of a radio button widget.

slider The Slider class defines the virtual interface methods for a slider widget.

5.1.2 Overview of initialization in main.cpp

Figure 5-1 shows the startup and run sequence in `main.cpp`:

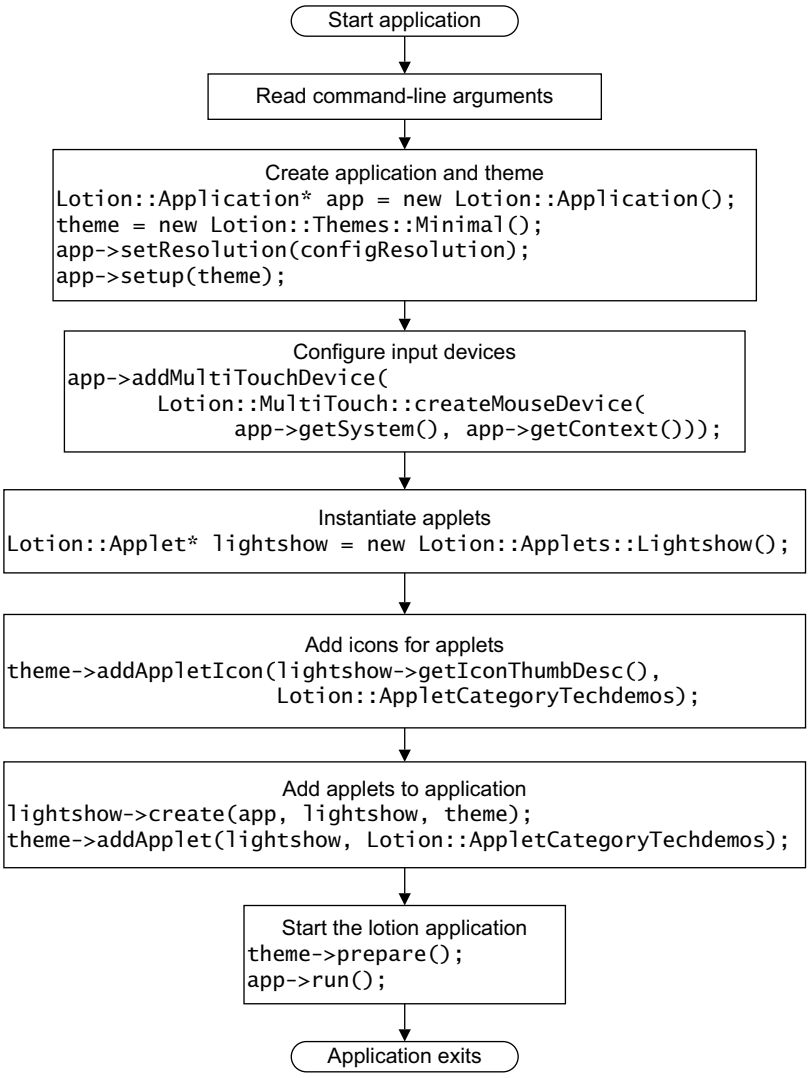

Figure 5-1 Lotion main.cpp execution flow

After initializing the application, the code in `main.cpp` prepares the theme and calls the `run()` method in the application. Figure 5-2 on page 5-7 shows the `run()` code within `application.cpp`:

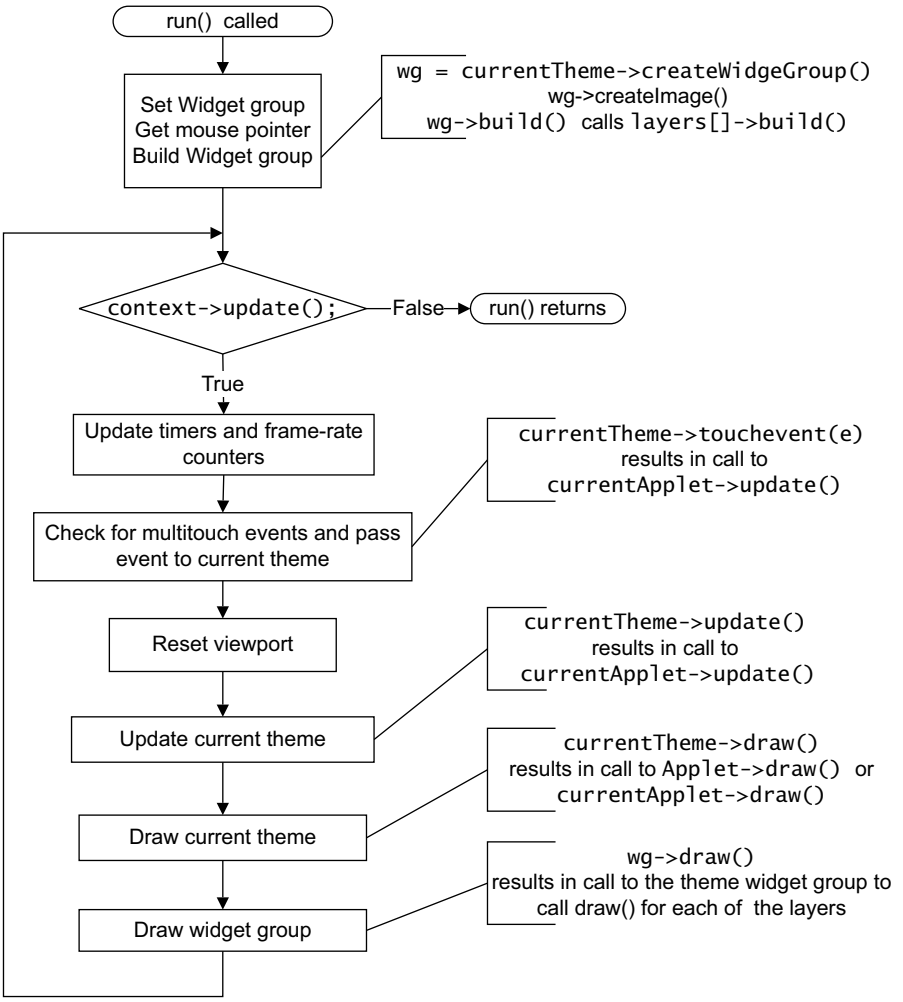

Figure 5-2 Application run() method

5.1.3 Interaction between the Lotion classes

A simplified view of the interaction during startup is shown in Figure 5-3 on page 5-8:

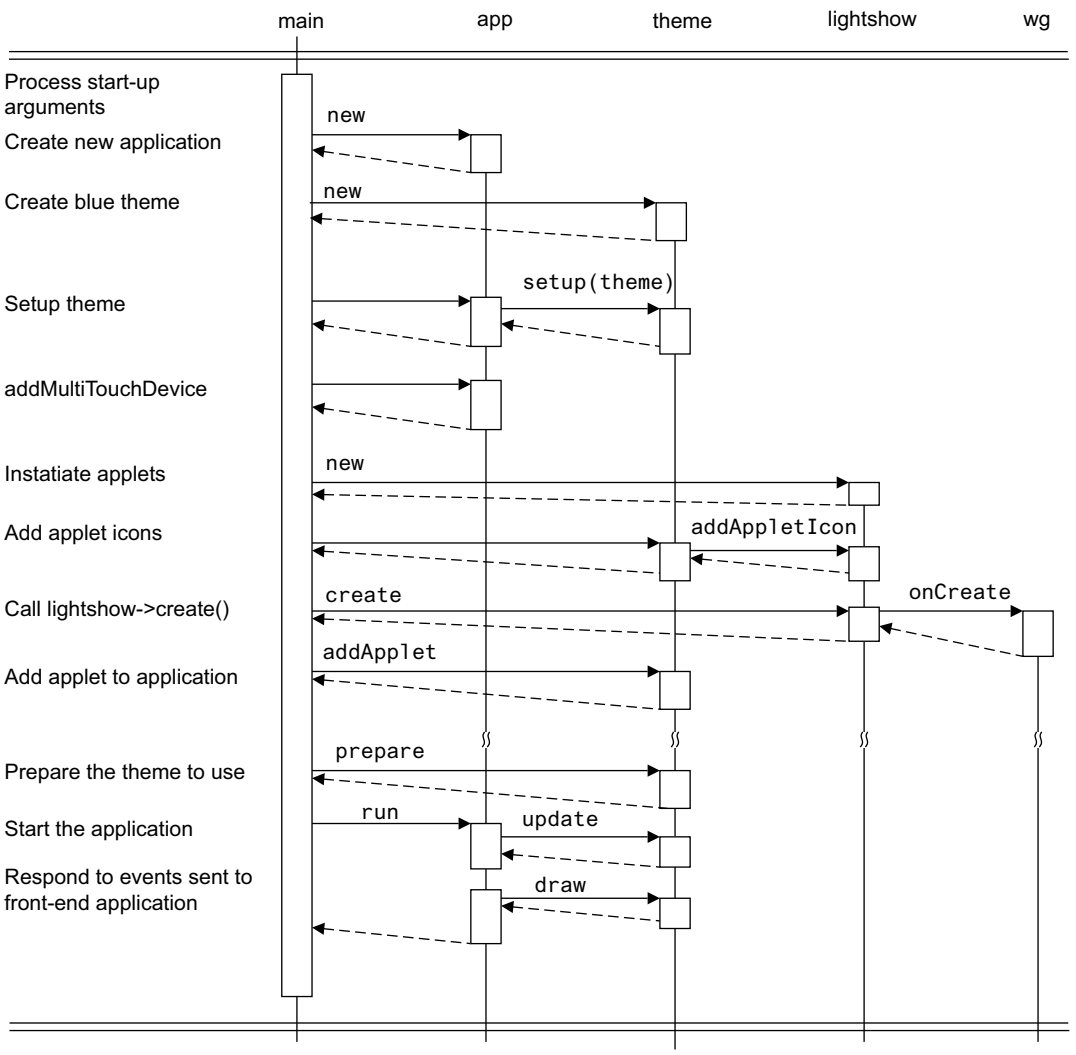

Figure 5-3 Startup and creation of applets

In Figure 5-3;

- `main` is the `main.cpp` code in the lotion directory
- `app` is an application created at startup by `main()`
- `theme` is a Theme object that uses code from the `themes` directory. The Blue theme is the default theme. It also creates a widget group, but that is not shown in Figure 5-3.
- `lightshow` is the reference to the lightshow applet. Only the lightshow applet is shown, but a similar instantiation sequence applies to all of applets.
- `wg` is a widget group created by the lightshow applet. This code calls the code in a Layer object to render its image to the display.
- If the exit control is clicked in the theme, `run()` exits and the lotion application ends.

Figure 5-4 on page 5-9 shows the control flow related to selecting and running the lightshow applet. The theme passes user events to the lightshow widget which then calls the lighshow application to process the user action.

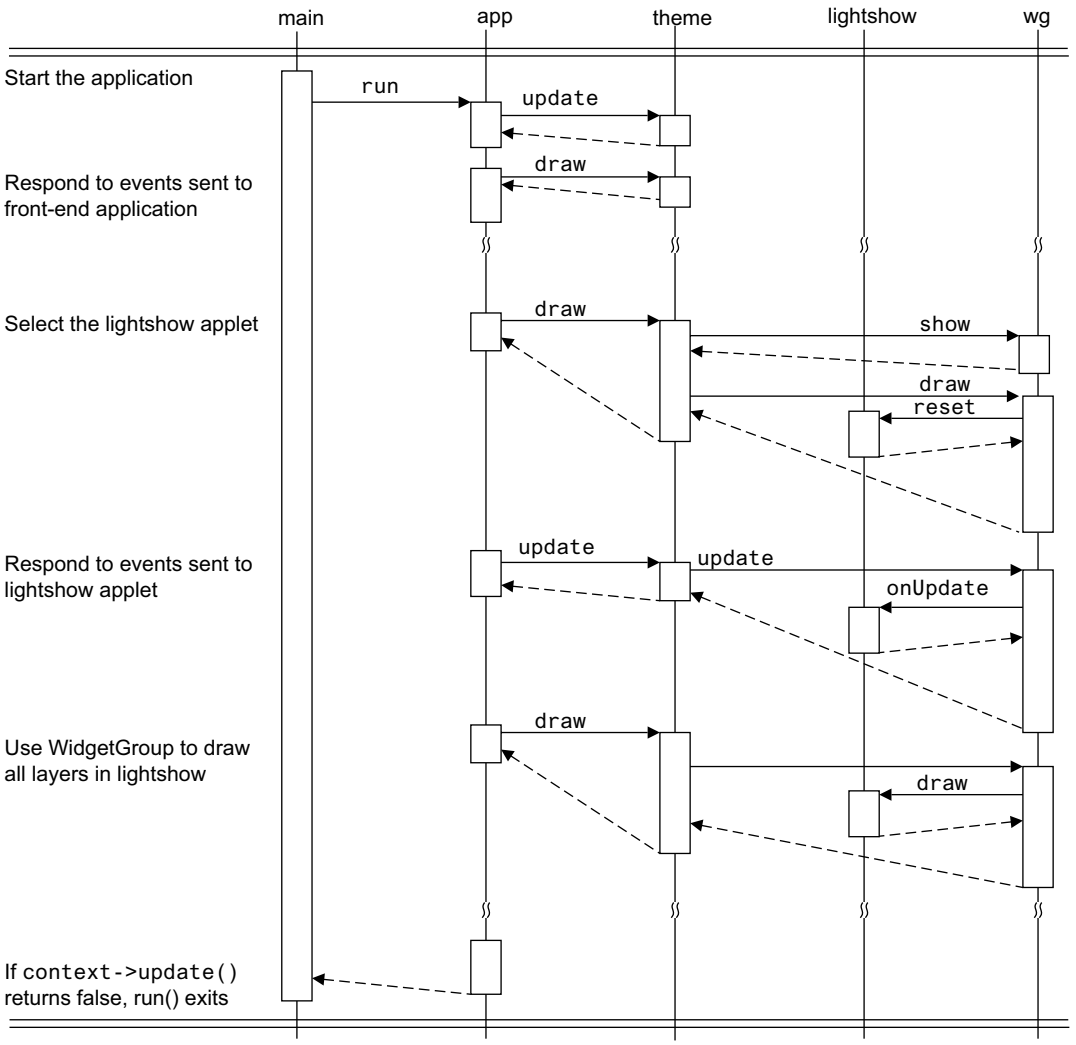

Figure 5-4 Control flow between the lotion application and the lightshow applet

5.2 The lotion main.cpp file

This section describes the logic flow in the `main.cpp` file:

- *Includes and enumerations in main.cpp*
- *Convenience functions* on page 5-11
- *Evaluating the start-up arguments* on page 5-12
- *Initializing the applets* on page 5-15
- *Starting the application* on page 5-17.

5.2.1 Includes and enumerations in main.cpp

The `main.cpp` file starts with include statements and enumerations as shown in Example 5-1:

Example 5-1 Includes and enumerations in main.cpp

```
#include <stdio.h>
#include <stdexcept>
#include <string.h>
#include <stdlib.h>
#ifdef MDE_OS_WIN32
#include <direct.h>
#else
#include <unistd.h>
#endif
#include <limits.h>
#include "lotion.h"

#include "applets/lightshow/lightshow.h"
#include "applets/browser/browser.h"
#include "applets/slideshow_presentation/slideshow_presentation.h"

enum applets {
    APPLET_PHOTO_BROWSER = 1 << 0,
    APPLET_PRESENTATION = 1 << 3,
    APPLET_SLIDESHOW_PRESENTATION = 1 << 4,
    APPLET_BUZZWORD = 1 << 5,
    APPLET_PLANET = 1 << 6,
    APPLET_LIGHTSHOW = 1 << 7,
    APPLET_BROWSER = 1 << 9,

    APPLETS_ALL = 0xFFFF,
    APPLETS_DEFAULT = APPLETS_ALL & ~APPLET_BROWSER
};

enum themes {
    THEME_BLUE = 1, THEME_ROBOT = 2, THEME_MINIMAL = 3
};

enum input_modes {
    INPUT_MODE_MOUSE = 1, INPUT_MODE_TOUCH = 2
};

struct input_descriptor {
    input_modes mode;
    MDE::String arg;
};
```

5.2.2 Convenience functions

Some convenience functions are defined as shown in Example 5-2:

Example 5-2 Convenience functions

```
static void cdToExecutable() {
    char buf[PATH_MAX + 1];
#ifdef MDE_OS_WIN32
    if(GetModuleFileName(GetModuleHandle(NULL), buf, sizeof(buf)) == 0)
    {
        MDE_GENERAL_ERROR("Can not find location of executing application");
    }
    char * enddir = strrchr(buf, '\\');
#else
    if (readlink("/proc/self/exe", buf, sizeof(buf) - 1) == -1)
    {
        MDE_GENERAL_ERROR("Can not find location of executing application");
    }
    char * enddir = strrchr(buf, '/');
#endif
    *enddir = 0;
    chdir(buf);
}

static void errorExit(const char * fmt, const char * arg)
{
    printf(fmt, arg);
#ifdef MDE_OS_WIN32
    // Pause so that we don't lose the message on exit.
    printf("Press RETURN to exit ...");
    getchar();
#endif
    exit(-1);
}

static void printHelp() {
    printf(
            "SYNTAX: lotion [-resolution arg] [-depth arg] [-theme arg] [-applets args] [-input args]\n\n");

    printf("-resolution\n");
    printf("    800x480\n");
    printf("    800x600\n\n");

    printf("-depth\n");
    printf("    bits per pixel\n\n");

    printf("-theme\n");
    printf("    blue\n");
    printf("    robot\n");
    printf("    minimal\n");

    printf("-applets\n");
    printf("    all\n");
    printf("    default (all)\n");
    printf("    lightshow\n");
    printf("    planet\n");
    printf("    buzzword\n");
    printf("    photo\n");
    printf("    presentation\n");
    printf("    slideshow\n");
    printf("    browser\n\n");
```

```
    printf("-input\n");
    printf("    mouse\n");
    printf("    touch <filename>\n");

}

static void argErrorExit(const char * fmt, const char * printfArg) {
    printHelp();
    errorExit(fmt, printfArg);
}
```

5.2.3 Evaluating the start-up arguments

The first part of `main()` evaluates the passed arguments:

1. Entering `resolution` on the command line specifies a new resolution to replace the default resolution. See Example 5-3:

Example 5-3 Setting the resolution

```
int main(int argc, char* argv[]) {
    try {
        // Run from the directory containing the executable,
        // so that we can find the data wherever it is run from
        cdToExecutable();

        // parse commandline args

        //bool configStandalone = false;
        int configApplets = 0;
        bool configAppletsAdded = false;
        themes configTheme = THEME_BLUE;
        MDE::Array<input_descriptor> configInputs;
        MDE::vec2i configResolution(800, 480);
        unsigned int configDepth = 0; // Use platform default config depth.

        for (int i = 1; i < argc; i++) {
            if (strcmp(argv[i], "-resolution") == 0) {
                if (i + 1 == argc) {
                    argErrorExit("No argument specified for -resolution switch",
                                 NULL);
                } else if (sscanf(argv[i + 1],
                           "%dx%d", &configResolution.x,
                            &configResolution.y) != 2) {
                      argErrorExit("Invalid -resolution: %s\n", argv[i + 1]);
                }

                i++;
            } else if (strcmp(argv[i], "-depth") == 0) {
...
```

2. Entering `depth` on the command line specifies the bits per pixel. See Example 5-4 on page 5-13:

Example 5-4 Setting the depth

```
...
            } else if (strcmp(argv[i], "-depth") == 0) {
                if (i + 1 == argc) {
                    argErrorExit("No argument specified for -depth switch",
                            NULL);
                } else if (sscanf(argv[i + 1], "%d", &configDepth) != 1) {
                    argErrorExit("Invalid -depth: %s\n", argv[i + 1]);
                }
                i++;
            } else if (strcmp(argv[i], "-theme") == 0) {
...
```

3. Entering `theme` on the command line specifies which theme to use. See Example 5-5:

Example 5-5 Setting the theme

```
...
            } else if (strcmp(argv[i], "-theme") == 0) {
                if (i + 1 == argc) {
                    argErrorExit("No argument specified for -theme switch", NULL);
                } else if (strcmp(argv[i + 1], "blue") == 0) {
                    configTheme = THEME_BLUE;
                } else if (strcmp(argv[i + 1], "robot") == 0) {
                    configTheme = THEME_ROBOT;
                } else if (strcmp(argv[i + 1], "minimal") == 0) {
                    configTheme = THEME_MINIMAL;
                } else {
                    argErrorExit(
                            "Unknown argument specified for -theme switch: %s",
                            argv[i + 1]);
                }

                i++;
            } else if (strcmp(argv[i], "-applets") == 0) {
...
```

4. Entering `applets` on the command line specifies which applets to load. See Example 5-6:

Example 5-6 Setting the applets

```
...
            } else if (strcmp(argv[i], "-applets") == 0) {
                while (i + 1 < argc) {
                    if (argv[i + 1][0] == '-') {
                        break;
                    } else if (strcmp(argv[i + 1], "all") == 0) {
                        configApplets |= APPLETS_ALL;
                    } else if (strcmp(argv[i + 1], "default") == 0) {
                        configApplets |= APPLETS_DEFAULT;
                    } else if (strcmp(argv[i + 1], "lightshow") == 0) {
                        configApplets |= APPLET_LIGHTSHOW;
                    } else if (strcmp(argv[i + 1], "planet") == 0) {
                        configApplets |= APPLET_PLANET;
                    } else if (strcmp(argv[i + 1], "buzzword") == 0) {
                        configApplets |= APPLET_BUZZWORD;
                    } else if (strcmp(argv[i + 1], "photo") == 0) {
```

```
                        configApplets |= APPLET_PHOTO_BROWSER;
                    } else if (strcmp(argv[i + 1], "presentation") == 0) {
                        configApplets |= APPLET_PRESENTATION;
                    } else if (strcmp(argv[i + 1], "slideshow") == 0) {
                        configApplets |= APPLET_SLIDESHOW_PRESENTATION;
                    } else if (strcmp(argv[i + 1], "browser") == 0) {
                        configApplets |= APPLET_BROWSER;
                    } else {
                        argErrorExit("Unknown applet: %s\n", argv[i + 1]);
                    }

                    configAppletsAdded = true;
                    i++;
                }
                if (!configAppletsAdded) {
                    argErrorExit("No arguments specified for -applets switch\n",
                                NULL);
                }
            } else if (strcmp(argv[i], "-input") == 0) {
...
```

5. The `input` command sets the mouse, touchscreen, and keyboard options. See Example 5-7:

Example 5-7 Setting the input devices

```
...
            } else if (strcmp(argv[i], "-input") == 0) {
                input_descriptor d;

                if (argc < i + 2 || argv[i + 1][0] == '-') {
                    argErrorExit("Too few arguments specified for -input\n", NULL);
                }

                if (strcmp(argv[i + 1], "mouse") == 0) {
                    d.mode = INPUT_MODE_MOUSE;
                } else if (strcmp(argv[i + 1], "touch") == 0) {
                    d.mode = INPUT_MODE_TOUCH;
                } else {
                    argErrorExit("Unknown argument specified for -input: %s\n",
                                argv[i + 1]);
                }
                i++;
                if (argc > i + 1 && argv[i + 1][0] != '-') {
                    d.arg = argv[i + 1];
                    i++;
                } else {
                    d.arg = "";
                }
                configInputs.add(d);
            } else if (strcmp(argv[i], "-help") == 0) {
...
```

6. If `help` is entered on the command line, the command options are listed. See Example 5-8 on page 5-15:

Example 5-8 Displaying command-line options

```
...
            } else if (strcmp(argv[i], "-help") == 0) {
                printHelp();

#ifdef MDE_OS_WIN32
                // Pause so that we don't lose the message on exit.
                printf("Press RETURN to exit ...");
                getchar();
#endif
                return 0;

            } else {
                argErrorExit("Warning: Ignoring unknown argument: %s\n", argv[i]);
            }
        }
...
```

5.2.4 Initializing the applets

The applets and context are initialized:

1. Create the theme and set the resolution. See Example 5-9:

Example 5-9 Initializing the applets

```
...
        if (!configAppletsAdded) {
            configApplets = APPLETS_DEFAULT;
        }

        Lotion::Application* app = new Lotion::Application();
        Lotion::Theme* theme = 0;

        if (configTheme == THEME_BLUE)
            theme = new Lotion::Themes::Blue();
        else if (configTheme == THEME_ROBOT)
            theme = new Lotion::Themes::Robot();
        else if (configTheme == THEME_MINIMAL)
            theme = new Lotion::Themes::Minimal();

        if (theme == 0) {
            argErrorExit("Error: No theme specified!\n", NULL);
        }
        app->setResolution(configResolution);
        app->setup(theme);
...
```

2. Configure the input device handler. See Example 5-10:

Example 5-10 Initializing the applets

```
...
        if (configInputs.getSize() == 0) {
```

```
            // We need an input device, assume a mouse on the default device.
            printf("No input device specified, using default mouse device\n");
            input_descriptor d = { INPUT_MODE_MOUSE, "" };
            configInputs.add(d);
        }

        for (int i = 0; i < configInputs.getSize(); i++) {
            if (configInputs[i].mode == INPUT_MODE_MOUSE) {
                app->addMultiTouchDevice(Lotion::MultiTouch::createMouseDevice(
                        app->getSystem(), app->getContext()));
            } else if (configInputs[i].mode == INPUT_MODE_TOUCH) {
#ifdef MDE_OS_WIN32
                // ...
#else
                app->addMultiTouchDevice(
                        Lotion::MultiTouch::createLinuxInputDevice(
                                app->getContext(), configInputs[i].arg));
#endif
            }
        }
...
```

Note

Touch screens are not supported on Windows.

3. Instantiate the applets and add the associated icons to the theme. See Example 5-11:

Example 5-11 Instantiate the applets

```
...
        // create instances of the individual applets
        Lotion::Applet* photo = new Lotion::Applets::Photo();
        Lotion::Applet* presentation = new Lotion::Applets::Presentation();
        Lotion::Applet* slideshow_presentation =
                new Lotion::Applets::SlideshowPresentation();
        Lotion::Applet* buzzword = new Lotion::Applets::Buzzword();
        Lotion::Applet* planet = new Lotion::Applets::Planet();
        Lotion::Applet* lightshow = new Lotion::Applets::LightShow();
        Lotion::Applet* browser = new Lotion::Applets::Browser();

        // add icons for the applets
        theme->addAppletIcon(photo->getIconThumbDesc(),
                Lotion::AppletCategoryPhotos);
        theme->addAppletIcon(presentation->getIconThumbDesc(),
                Lotion::AppletCategoryPresentations);
        theme->addAppletIcon(slideshow_presentation->getIconThumbDesc(),
                Lotion::AppletCategoryPresentations);
        theme->addAppletIcon(buzzword->getIconThumbDesc(),
                Lotion::AppletCategoryTechdemos);
        theme->addAppletIcon(planet->getIconThumbDesc(),
                Lotion::AppletCategoryTechdemos);
        theme->addAppletIcon(lightshow->getIconThumbDesc(),
                Lotion::AppletCategoryTechdemos);
        theme->addAppletIcon(browser->getIconThumbDesc(),
                Lotion::AppletCategoryTechdemos);
...
```

4. Create the applet and add it to the theme. See Example 5-12 on page 5-17:

Example 5-12 Creating the applets

```
...
        // create the applet and add it to the theme
        if (configApplets & APPLET_PHOTO_BROWSER) {
            photo->create(app, photo, theme);
            theme->addApplet(photo, Lotion::AppletCategoryPhotos);
        }
        if (configApplets & APPLET_PRESENTATION) {
            presentation->create(app, presentation, theme);
            theme->addApplet(presentation, Lotion::AppletCategoryPresentations);
        }
        if (configApplets & APPLET_SLIDESHOW_PRESENTATION) {
            slideshow_presentation->create(app, slideshow_presentation, theme);
            theme->addApplet(slideshow_presentation,
                             Lotion::AppletCategoryPresentations);
        }
        if (configApplets & APPLET_BUZZWORD) {
            buzzword->create(app, buzzword, theme);
            theme->addApplet(buzzword, Lotion::AppletCategoryTechdemos);
        }
        if (configApplets & APPLET_PLANET) {
            planet->create(app, planet, theme);
            theme->addApplet(planet, Lotion::AppletCategoryTechdemos);
        }
        if (configApplets & APPLET_LIGHTSHOW) {
            lightshow->create(app, lightshow, theme);
            theme->addApplet(lightshow, Lotion::AppletCategoryTechdemos);
        }
        if (configApplets & APPLET_BROWSER) {
            browser->create(app, browser, theme);
            theme->addApplet(browser, Lotion::AppletCategoryBrowser);
        }

        // Calibration - MUST BE THE LAST APPLET ADDED
        Lotion::CalibrationApplet* calibration =
                new Lotion::CalibrationApplet();
        calibration->create(app, calibration, theme);
        theme->addApplet(calibration, Lotion::AppletCategoryTechdemos);

        theme->prepare();
...
```

5.2.5 Starting the application

The application is started shown in Example 5-13. Any exceptions are trapped.

Example 5-13 Starting the application

```
...
       app->run();
    } catch (MDE::Exception e) {
        argErrorExit("%s\n", e.getMessage().getCharString());
    } catch (std::exception e) {
        argErrorExit("Standard Library Exception?\n", NULL);
    } catch (...) {
        argErrorExit("Non MDE Exception\n", NULL);
```

```
    }
    return 0;
}
```

5.3 The application.cpp file

The application.cpp file contains the code that implements the Application class:

1. The application.h header file contains the Application class definition. See Example 5-15:

Example 5-14 Application header file

```
class Application
{
    Theme* currentTheme;
    MDE::System* system;
    MDE::Context* context;
    MDE::FileSystem* fileSystem;
    MDE::Proxy* proxy;
    MDE::Timer* timer;
    MDE::Array<MultiTouch::Device*> mtDevices;
    MDE::vec2i resolution;
    RenderEngine* renderEngine;
    MultiTouch::Device* calibratingDevice;
    MDE::vec4i viewport;
    bool nativeResolution;

    public:
    Application();
    RenderEngine* getRenderEngine();
    MDE::Context* getContext();
    MDE::System* getSystem();
    void setResolution(MDE::vec2i resolution);
    void addMultiTouchDevice(MultiTouch::Device* mtDevice);
    void setup(Theme* theme);
    void run();
    void runApplet(Applet* applet);
    MDE::Proxy* getProxy();
    Theme* getTheme();
    float getTime();
    void resetViewport();
};
```

2. The constructor assigns values for the viewing area. See Example 5-15:

Example 5-15 Application constructor

```
Application::Application()
{
    nativeResolution = true;
    viewport = MDE::vec4i(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
    resolution = MDE::vec2i(WINDOW_WIDTH, WINDOW_HEIGHT);
    calibratingDevice = 0;
}
```

The constants WINDOW_HEIGHT and WINDOW_WIDTH are 480 and 800

3. Initialization code in main.cpp calls the setup methods. See Example 5-16 on page 5-20:

Example 5-16 Application setup methods

```
// called from main.cpp
void Application::setup(Theme* theme)
{
    system = MDE::create_system_backend();
    context = system->createContext(resolution.x, resolution.y);
    fileSystem = system->createFileSystem("data/");
    timer = system->createTimer();
    proxy = new MDE::Proxy(fileSystem, context);
    proxy->addPriorityLoader(new EtcTextureDemoLoader());
    proxy->addPriorityLoader(new EtcBitmapDemoLoader());
    renderEngine = new RenderEngine(proxy, MDE::vec2i(WINDOW_WIDTH, WINDOW_HEIGHT));

    this->currentTheme = theme;
    printf("Creating theme...\n");
    if (currentTheme)
        currentTheme->create(this, 0, 0);
    printf("Theme created successfully\n");
}

// called from main.cpp
void Application::addMultiTouchDevice(MultiTouch::Device* mtDevice)
{
    mtDevices.add(mtDevice);
}
```

4. The applets and themes use the access methods. See Example 5-17:

Example 5-17 Application access methods

```
// used in applet.cpp, robot.cpp and widget.cpp
RenderEngine* Application::getRenderEngine()
{
    return renderEngine;
}

// used in applet.cpp and blue.cpp
MDE::Proxy* Application::getProxy()
{
    return this->proxy;
}

// used in applet.cpp and blue.cpp
float Application::getTime() {
    return timer->getTime();
}

// the widget getTheme() method returns the result of getApplication->getTheme()
Theme* Application::getTheme()
{
    return currentTheme;
}

// used by main.cpp to set up the multitouch device and by the applets
MDE::Context* Application::getContext()
{
    return context;
}
```

```
// used by main.cpp to set up the multitouch device
MDE::System* Application::getSystem()
{
    return system;
}
```

5. The `run()` function performs initialization. See Example 5-18:

Example 5-18 Application run methods

```
// called from run() if not using native resolution
void Application::resetViewport()
{
    glViewport(viewport.x, viewport.y, viewport.z, viewport.w);
}

// called from run() if not using native resolution
void Application::setResolution(MDE::vec2i resolution)
{
    this->resolution = resolution;
    nativeResolution = resolution.x == WINDOW_WIDTH && resolution.y == WINDOW_HEIGHT;

    viewport.x = 0;
    viewport.y = resolution.y - WINDOW_HEIGHT;
    viewport.z = WINDOW_WIDTH;
    viewport.w = WINDOW_HEIGHT;
}

void Application::runApplet(Applet* applet) {
    currentTheme->runApplet(applet);
}

// main.cpp calls app->run()
void Application::run() {
#ifdef DRAW_INPUT_POSITION
    printf("creating mouse pointer\n");

    if (!currentTheme)
    {
        throw MDE::Exception("Application::run: currentTheme not set");
    }

    WidgetGroup* wg = currentTheme->createWidgetGroup(currentTheme, "input_pointer");
    Widgets::Image* inputPointerWidget = wg->createImage(0);

    inputPointerWidget->setImage(inputPointerWidget->getBitmap2DAsset(
                                                "themes/blue/widgets/mouse_widget.png"));
    inputPointerWidget->setSize(MDE::vec2i(32, 32));
    wg->build();
    MDE::vec2i inputPosition;
#endif

    timer->reset();

    while (context->update())
    {
        //printf("mainloop\n");

        float time = timer->getTime();
        float fps = 0.0f;
```

```
            static float lastFpsTime = 0.0f;
            static int frameCount = 0;

            float fpsTime = time - lastFpsTime;
            if (fpsTime > 1.0f) {
            fps = float(frameCount) / fpsTime;
            lastFpsTime = time;
            frameCount = 0;

            static float accFps = 0.0f;
            static float accFpsCount = 0.0f;
            accFps += fps;
            accFpsCount += 1.0;
            float averageFps = accFps / accFpsCount;

            printf("Time: %.2f\tFPS: %.2f (%.2f ms)\tAverage FPS: %.2f (%.2f ms)\n",
                           time, fps, 1000.0f / fps, averageFps, 1000.0f / averageFps);
        }

        frameCount++;

        for (int i = 0; i < mtDevices.getSize(); i++)
        {
            mtDevices[i]->update();

            while (mtDevices[i]->hasNextEvent())
            {
                MultiTouch::Event e = mtDevices[i]->getNextEvent();

                if (calibratingDevice == 0)
                {
                    if (mtDevices[i]->needsCalibration())
                    {
                        calibratingDevice = mtDevices[i];
                        currentTheme->runCalibration();
                    }
                }

#ifdef DRAW_INPUT_POSITION
                inputPosition = e.position;
#endif

                if (currentTheme)
                    currentTheme->touchEvent(e);
            }
        }

        if (!nativeResolution)
        {
            glViewport(0, 0, resolution.x, resolution.y);
        }

        glClearColor(0, 0, 0, 0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
        glDisable(GL_CULL_FACE);

        if (!nativeResolution)
        {
           resetViewport();
        }

        renderEngine->setTime(time);
```

```
#ifdef DRAW_INPUT_POSITION
    //printf ("rendering mouse pointer\n");
    inputPointerWidget->setPosition(inputPosition - MDE::vec2i(7, 7));
    wg->setVisible(false);
#endif

    if (currentTheme)
    {
        currentTheme->update();
        currentTheme->draw();
    }

#ifdef DRAW_INPUT_POSITION
    glDisable(GL_DEPTH_TEST);
    wg->setVisible(true);
    wg->draw();
#endif
}
```

5.4 Modifications to lightshow.cpp

This section describes the modifications required to convert the standalone lightshow application to an applet that runs in the lotion interface:

- *New lightshow.h file*
- *Modified startup code in lightshow.cpp* on page 5-26
- *Changes to traverseDraw()* on page 5-27
- *Utility functions for lighting* on page 5-28
- *Widget initialization and event handling* on page 5-29
- *Modified draw loop* on page 5-30.

The standalone version of the application is described in *Lightshow animation with moving sprite and camera* on page 4-47.

5.4.1 New lightshow.h file

A new `lightshow.h` file is required because the lightshow applet is used by the lotion `main.cpp` code. Example 5-19 lists the new header file:

Example 5-19 New lightshow.h code

```
#ifndef __LIGHTSHOW_APPLET_H__
#define __LIGHTSHOW_APPLET_H__

#include "../../lotion.h"
#include "../../applet.h"
#include "../../application.h"
#include "../../theme.h"

namespace Lotion    // was namespace MDE in the original lightshow app
{
    namespace Applets
    {
        // lightshow is now an applet class
        class LightShow: public Applet
        {
            MDE::Buffer* flareTexcoord; // this was in the original lightshow app
            GLint flareTexcoordsLoc;    // this was in the original lightshow app

            bool useWireframe;
            bool styggShader;
            int lightCount;

            MDE::SceneAsset* scene;
            MDE::Program* flarePrg;
            MDE::Program* prg_gl2;
            MDE::Program* prg_gl1;

            MDE::NodeAsset* camNode;
            MDE::NodeAsset* targetNode;
            MDE::NodeAsset* light0Node;
            MDE::NodeAsset* light1Node;

            MDE::Texture* flareTex;

            // references to functions that were in the original app
            void prepareFlares(MDE::Context* context, MDE::Program* p);
            void drawFlare(MDE::Context* context, MDE::vec3 position,
                           MDE::vec3 color, float size);
```

```
            void traverseTransform(MDE::Tree<MDE::NodeAsset*>* node,
                                   MDE::mat4 parentTransform, float time);
            void applyMaterial(MDE::MaterialAsset* m, MDE::Context* context);
            void traverseDraw(MDE::Tree<MDE::NodeAsset*>* node,
                              MDE::Context* context);
            float fract(float v);
            MDE::vec2 angles_for_light(MDE::vec2 cyl_pos,
                                       float cyl_radius, MDE::vec2 light_position);
            MDE::vec3 calculate_cyl_info(MDE::vec2 cyl_pos, float cyl_radius,
                                         MDE::vec2 light_position, float light_radius);

            float time;

            // new app must use widgets and radio buttons
            WidgetGroup* wg;
            Widget* exitButton;

            Widgets::Panel* panel;

            RadioButtonGroup* glVersion;
            Widgets::CustomLabel* settingsGlLbl;
            Widgets::RadioButton* gl11Chk;
            Widgets::RadioButton* gl20Chk;

            RadioButtonGroup* renderMode;
            Widgets::CustomLabel* renderLbl;
            Widgets::RadioButton* solidChk;
            Widgets::RadioButton* wireframeChk;

        public:
            // code related to widget creation and associated icons
            virtual String getName() {
                 return "LightShow Demo";
            }
            virtual String getDescription() {
                 return "Light Show Applet.";
            };
            virtual String getIconThumbDesc() {
                 return "applets/icons/lightshow_small.png";
            }
            virtual String getIconFullDesc() {
                 return "applets/icons/lightshow_correct.png";
            }

            // initialize the lightshow proxy, context, and assets
            virtual void onCreate(Widget* w);

            // respond to changed drawing options
            virtual void onValueChanged(Widget* w);

            virtual void draw();
        };
    }
}

#endif
```

5.4.2 Modified startup code in lightshow.cpp

The original lighshow.cpp file in `...\example_applications\lightshow` is not the same as the one in `...\lotion\applets\lightshow`:

- the standalone application is converted to a be a class
- additional code is added to interface with the theme and application classes.

The modified lightshow.cpp file uses the header file and the Lotion namespaces as shown in :

Example 5-20 Namespace for lightshow

```
// add reference to new header file
#include "lightshow.h"

// remove low-level includes

using namespace MDE;

namespace Lotion // use Lotion namespace
{
    namespace Applets // use Applets namespace
    {
     ...
    }
}
```

The first part of `lightshow.cpp` file is similar to the original, except that some widget code has been added as shown in Example 5-21:

Example 5-21 Functions in lightshow.cpp

```
// same as original except now class method
void LightShow::prepareFlares(Context* context, MDE::Program* p)
{
    float texcoords[] = { 0, 0, 1, 0, 1, 1, 0, 1 };
    flareTexcoord = context->createBuffer(GL_ARRAY_BUFFER,
                  sizeof(texcoords), 8);
    flareTexcoord->setData(0, sizeof(texcoords), texcoords);
    flareTexcoordsLoc = p->getAttribLocation(TEXCOORD0);
}

// same as original except now class method
void LightShow::drawFlare(Context* context, vec3 position, vec3 color, float size)
{
    context->setUniform("position", position);
    context->setUniform("color", color);
    context->setUniform("size", size);

    glEnableVertexAttribArray(flareTexcoordsLoc);
    glBindBuffer(GL_ARRAY_BUFFER, flareTexcoord->getHandle());
    glVertexAttribPointer(flareTexcoordsLoc, 2, GL_FLOAT, GL_FALSE, 0, 0);

    unsigned short indices[] = { 0, 1, 2, 0, 2, 3 };

    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, indices);

    glDisableVertexAttribArray(flareTexcoordsLoc);
}
```

```
// same as original except now class method
void LightShow::traverseTransform(Tree<NodeAsset*>* node,
                  mat4 parentTransform, float time)
{
    if (!node)
        return;
    traverseTransform(node->getNextSibling(), parentTransform, time);
    if (node->data) {
        mat4 localTransform = node->data->sampleLocalTransform(time);
        parentTransform = localTransform * parentTransform;
        node->data->setAbsoluteTransform(parentTransform);
    }
    traverseTransform(node->getFirstChild(), parentTransform, time);
}

// same as original except now class method
void LightShow::applyMaterial(MaterialAsset* m, Context* context)
{
    if (m) {
        if (m->commonMaps.diffuse)
            context->setUniformSampler("diffuseMap",
                       m->commonMaps.diffuse->getTexture());

        if (m->commonMaps.bump) {
            Texture* t = m->commonMaps.bump->getTexture();
            context->setUniformSampler("normalMap", t);
        }

        if (m->commonMaps.spec_level)
            context->setUniformSampler("glossMap",
                       m->commonMaps.spec_level->getTexture());
    }
}
```

5.4.3 Changes to traverseDraw()

The `traverseDraw()` function is the same except for the namespace and the addition of a loop through the batches. See Example 5-22:

Example 5-22 traverseDraw() function

```
// now class method and display options configurable by radio buttons
void LightShow::traverseDraw(Tree<NodeAsset*>* node, Context* context) {
    if (!node)
        return;
    traverseDraw(node->getNextSibling(), context);
    if (node->data)
    {
        mat4 world = node->data->getAbsoluteTransform();
        context->setUniformMatrix("WORLD", world);

        mat4 inv_trans_world = world.transposed();
        inv_trans_world.invert(inv_trans_world);
        context->setUniformMatrix("INV_TRANS_WORLD", inv_trans_world);

        // new for applet, loop through batches
        for (unsigned int i = 0; i < node->data->getBatchCount(); i++)
        {
            Batch b = node->data->getBatch(i);
```

```
            applyMaterial(b.material, context);
            if (useWireframe) {
            for (unsigned int j = 0; j < b.geometry->getVertexBufferCount(); j++)
            {
                if (b.geometry->getVertexBuffer(j))
                    context->setVertexBuffer(j, b.geometry->getVertexBuffer(j));
            }
            if (b.geometry->getVertexDeclaration())
                context->setVertexDeclaration(b.geometry-> getVertexDeclaration());
            if (b.geometry->getIndexBuffer())
                context->drawElements(GL_LINE_STRIP, 0,
                          b.geometry->getPrimitiveCount() * 3,
                          b.geometry->getIndexBuffer());
            }
            else
            {
            b.geometry->draw();
            }
        }
    }
    traverseDraw(node->getFirstChild(), context);
}
```

5.4.4 Utility functions for lighting

`LightShow()`, `angles_for_lights()`, and `calculate_cyl_info()` are identical to the original except for the namespace.See Example 5-23:

Example 5-23 Utility functions for lighting

```
// same as original except now class method
float LightShow::fract(float v) {
    return v - floor(v);
}

// same as original except now class method
vec2 LightShow::angles_for_light(vec2 cyl_pos, float cyl_radius, vec2 light_position) {
    vec2 light_cyl = cyl_pos - light_position;
    float light_dist = light_cyl.length();
    float scale_factor = (0.5f / 3.141532f);
    float cyl_center_angle = atan2(light_cyl.y, light_cyl.x) * scale_factor;
    float cyl_radius_angle = sin(cyl_radius / light_dist) * scale_factor;
    return vec2(cyl_center_angle - cyl_radius_angle,
                cyl_center_angle + cyl_radius_angle);
}

vec3 LightShow::calculate_cyl_info(vec2 cyl_pos, float cyl_radius,
                                   vec2 light_position, float light_radius) {
    vec2 light_cyl = cyl_pos - light_position;
    vec2 light_sideway_vec = vec2(light_cyl.y, -light_cyl.x).normalized();
    float light_dist = light_cyl.length();

    vec2 angles = angles_for_light(cyl_pos, cyl_radius, light_position);
    float offset = fract((angles.x + angles.y) / 2.0);
    float limit = 0.5 - fract(angles.y - offset);
    vec3 shadow_info(offset, light_dist, limit);
    return shadow_info;
}
```

5.4.5 Widget initialization and event handling

New methods are added to respond to widget creation or change. See Example 5-24:

Example 5-24 Widget initialization and event handling

```
void LightShow::onCreate(Widget* w) {
    static const int NORMAL_FONT_SIZE=21;
    if (w != this)
        return;

    styggShader = false;
    useWireframe = false;

    Context* context = getApplication()->getContext();
    Proxy* proxy = getApplication()->getProxy();

    scene = proxy->getSceneAsset(
         "applets/lightshow/lightshow_remake.MBA{applets/lightshow/}");
    flarePrg = proxy->getProgram(
         "applets/lightshow/flare.vert;applets/lightshow/flare.frag");
    prg_gl2 = proxy->getProgram(
         "applets/lightshow/lightshow.vert;applets/lightshow/lightshow.frag");
    prg_gl1 = proxy->getProgram(
          "applets/lightshow/lightshow_gl1.vert;applets/lightshow/lightshow_gl1.frag");

    prepareFlares(context, flarePrg);

    camNode = (NodeAsset*) scene->getAsset(Asset::TYPE_NODE, "Camera01-node");
    targetNode = (NodeAsset*) scene->getAsset(Asset::TYPE_NODE,
                      "Camera01.Target-node");
    light0Node = (NodeAsset*) scene->getAsset(Asset::TYPE_NODE, "Omni01-node");
    light1Node = (NodeAsset*) scene->getAsset(Asset::TYPE_NODE, "Omni02-node");

    flareTex = proxy->getTexture2D("applets/lightshow/flare.png");
    time = 0;

    wg = createWidgetGroup("lightshow_wg");

    // add the user controls to switch between shading effects
    panel = wg->createPanel(this);
    panel->setSize(MDE::vec2i(180, 200));
    panel->setPosition(MDE::vec2i(0, 0));

    settingsGlLbl = wg->createCustomLabel(this);
    settingsGlLbl->build(NORMAL_FONT_SIZE, 16);
    settingsGlLbl->setPosition(vec2i(10, 0));
    settingsGlLbl->setColor(vec4(1.f, 1.f, 1.f, 1.f));
    settingsGlLbl->setTextAlign(TEXT_ALIGN_LEFT | TEXT_ALIGN_TOP);
    settingsGlLbl->setText("GL Version");

    gl11Chk = wg->createRadioButton(this);
    gl11Chk->setText("GLES 1.1");
    gl11Chk->setPosition(vec2i(10, 25));

    gl20Chk = wg->createRadioButton(this);
    gl20Chk->setText("GLES 2.0");
    gl20Chk->setPosition(vec2i(10, 50));

    glVersion = new RadioButtonGroup();
    glVersion->add(gl11Chk);
    glVersion->add(gl20Chk);
```

```
    renderLbl = wg->createCustomLabel(this);
    renderLbl->build(NORMAL_FONT_SIZE, 16);
    renderLbl->setPosition(vec2i(10, 90));
    renderLbl->setColor(vec4(1.f, 1.f, 1.f, 1.f));
    renderLbl->setTextAlign(TEXT_ALIGN_LEFT | TEXT_ALIGN_TOP);
    renderLbl->setText("Rendermode");

    solidChk = wg->createRadioButton(this);
    solidChk->setText("Solid");
    solidChk->setPosition(vec2i(10, 115));

    wireframeChk = wg->createRadioButton(this);
    wireframeChk->setText("Wireframe");
    wireframeChk->setPosition(vec2i(10, 140));

    renderMode = new RadioButtonGroup();
    renderMode->add(solidChk);
    renderMode->add(wireframeChk);

    exitButton = wg->createExitButton(this);
    exitButton->setPosition(MDE::vec2i(130, 150));

    solidChk->check();
    gl20Chk->check();

    setPosition(MDE::vec2i(0, 0));
    setSize(MDE::vec2i(800, 480));
}

void LightShow::onValueChanged(Widget* w) {
    useWireframe = wireframeChk->isChecked();
    styggShader = gl11Chk->isChecked();
}
```

5.4.6 Modified draw loop

The code in the original draw loop is moved to a `draw()` method.

The FPS calculation has been removed. See Example 5-25:

Example 5-25 Modified draw functionality in lightshow.cpp

```
void LightShow::draw() {
    Context* context = getApplication()->getContext();

    float deltat = getTime() * 0.3f - time;
    time = getTime() * 0.3f;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    float scenetime = time;
    while (scenetime > 10)
        scenetime -= 10;

    MDE::Program* prg;
    if (styggShader)
        prg = prg_gl1;
    else
        prg = prg_gl2;
```

```
    mat4 rootTransform = mat4::identity();
    traverseTransform(scene->getTree(0), rootTransform, scenetime);
    vec3 camPos = camNode->getAbsolutePosition();
    vec3 targetPos = targetNode->getAbsolutePosition();

    mat4 view = mat4::lookAt(camPos, targetPos, vec3(0, 0, 1));
    mat4 proj = mat4::perspective(80, 1.33333f, 1, 500);

    context->setProgram(prg);
    context->setUniformMatrix("VIEW_PROJECTION", proj * view);

    vec3 light0position = light0Node->getAbsolutePosition();
    vec3 light0color(0.66f + fabs(sinf(time * 0.1f)) * 0.4f,
                     0.66f + fabs(sinf(time * 0.12f + 0.4f)) * 0.4f,
                     0.66f + fabs(sinf(time * 0.18f + 0.989f)) * 0.4f);
    context->setUniform("light0position", light0position);
    context->setUniform("light0color", light0color);

    vec3 light1position = light1Node->getAbsolutePosition();
    vec3 light1color(0.66f + fabs(sinf(time * 0.1f)) * 0.4f,
                     0.66f + fabs(sinf(time * 0.12f + 0.14f)) * 0.4f,
                     0.66f + fabs(sinf(time * 0.18f + 0.389f)) * 0.4f);
    context->setUniform("light1position", light1position);
    context->setUniform("light1color", light1color);

    context->setUniform("cameraPosition", camPos);

    float cyl_radius = sqrt(30.0f);

    vec3 shadow_info[2] =
     { calculate_cyl_info(vec2(-30.0, -30.0), cyl_radius,
                          vec2(light0position.x,light0position.y), 0.0),
     calculate_cyl_info(vec2(30.0, 30.0), cyl_radius,
                          vec2(light0position.x, light0position.y), 0.0) };

    context->setUniform("shadow_info[0]", shadow_info[0]);
    context->setUniform("shadow_info[1]", shadow_info[1]);

    glEnable(GL_DEPTH_TEST);
    traverseDraw(scene->getTree(0), context);

    glDepthMask(GL_FALSE);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    context->setProgram(flarePrg);
    context->setUniformMatrix("VIEW", view);
    context->setUniformMatrix("PROJECTION", proj);
    context->setUniformSampler("colorTex", flareTex);
    drawFlare(context, light0position, light0color, 10);
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);

    // new code to make the widget visible and draw the image
    wg->setVisible(true);
    Applet::draw();
}
```

5.5 The Theme class and blue.cpp

The Theme class:

- displays the controls and applet icons
- manages user input
- runs an applet if it is selected
- is the parent class for the Blue, Minimal, and Robot classes.

There are several themes available for use in the application. This section uses example code from `lotion\themes\blue.cpp`:

5.5.1 Theme namespace and includes

The Theme methods use the `Themes`, `Lotion`, and `MDE` namespaces. See Example 5-27:

Example 5-26 Theme namespace and includes

```
#include "blue.h"
#include "../../application.h"
#include "blue_widget_group.h"
#include "../../transitions/puzzle.h"
#include "../../transitions/slime.h"
#include "../../transitions/cube.h"

using namespace MDE;

namespace Lotion
{
    namespace Themes
    {
      ...
    }
}
```

5.5.2 Theme onCreate() method

Example 5-27 shows the start of the `blue.cpp` file and the `onCreate()` method:

Example 5-27 Theme onCreate() method

```
...
void Blue::onCreate(Widget* w)
{
    printf("Blue: Loading shader\n");
    state = StateBrowsing;
```

```
setPosition(MDE::vec2i(0, 0));
setSize(MDE::vec2i(8000, 4800));
font = getFont("fonts/orbitron-light.ttf", 24);
prog = getProgram("themes/blue/shader.vert;themes/blue/shader.frag");

// Background
printf("Blue: Loading background\n");
backgroundLayer = createLayer(prog, 1, 0, 0,
    Lotion::LayerModeRGBA, "blue_bg");
backgroundLayer->createInstances(1, backgroundInstances);
backgroundInstances[0]->createQuads(1, false, backgroundQuads);
backgroundQuads[0]->set2DRectangle(vec2(0,0), getResolution(), 0);
backgroundQuads[0]->setTexture(0, getBitmap("themes/blue/backdrop.png"),
                               BitmapComponentsRGBA_PremultiplyAlpha);
backgroundLayer->setFilter(GL_NEAREST, GL_NEAREST);

// Logo
printf("Blue: Loading logo\n");
backgroundInstances[0]->createQuads(1, true, backgroundQuads);
backgroundQuads[1]->set2DRectangle(vec2(2,2), vec2(126, 46), 0);
backgroundQuads[1]->setTexture(0, getBitmap("themes/blue/mali_logo.png"),
                                        BitmapComponentsRGBA_PremultiplyAlpha);

// Icons
printf("Blue: Loading icons: shader\n");
Program menuProg =
        getProgram("themes/blue/menu.vert;themes/blue/menu.frag");
iconLayer = createLayer(menuProg, 4, 0, 0, LayerModeRGBCompressed,
                        "blue_icons");
iconLayer->setFilter(GL_NEAREST, GL_NEAREST);
iconLayer->createInstances(iconCount, iconInstances);

printf("Blue: Loading icons: text shader\n");
Program textProg =
             getProgram("themes/blue/text.vert;themes/blue/text.frag");
printf("Blue: Loading icons: creating layer\n");
textLayer = createLayer(textProg, 1, 0, 0, LayerModeRGBA, "blue_text");
printf("Blue: Loading icons: creating layer instances\n");
textLayer->createInstances(iconCount, iconTextInstances);
textLayer->setFilter(GL_NEAREST, GL_NEAREST);

for (int i = 0; i < iconCount; i++)
{
    printf("Blue: Loading icons: Icon %d\n", i);
    iconInstances[i]->createQuads(4, true, iconQuads);

    float add_refl = 0.f;
    float refl_dist = 30.f;
    // Icon
    iconQuads[i*4+0]->set2DRectangle(vec2(i * 265.f+16, 105.f + add_refl),
                     vec2(240.f, 136.f), 0);
    iconQuads[i*4+0]->setTexture(0, getBitmap("themes/blue/empty.png"),
                          BitmapComponentsRGB);
    iconQuads[i*4+0]->setTexture(1, getBitmap("themes/blue/empty.png"),
                          BitmapComponentsRGB);
    // Icon Reflection
    iconQuads[i*4+1]->set2DRectangle(
              vec2(i * 265+16, 87+162*2+refl_dist+ add_refl),
              vec2(240, -136), 0);
    iconQuads[i*4+1]->setTexture(0, getBitmap("themes/blue/empty.png"),
                          BitmapComponentsRGB);
    iconQuads[i*4+1]->setTexture(1, getBitmap("themes/blue/empty.png"),
                          BitmapComponentsRGB);
```

```
        // Frame
        iconQuads[i*4+2]->set2DRectangle(vec2(i * 265.f+7.f, 96.f+ add_refl),
                         vec2(258.f, 154.f), 0);
        iconQuads[i*4+2]->setTexture(0, getBitmap("themes/blue/frame.png"),
                             BitmapComponentsRGB_PremultiplyAlpha);
        iconQuads[i*4+2]->setTexture(1, getBitmap("themes/blue/frame.png"),
                             BitmapComponentsAAA);
        // Frame Reflection
        iconQuads[i*4+3]->set2DRectangle(
                vec2(i * 265+7, 96+162*2+refl_dist+ add_refl),
                vec2(258, -154), 0);
        iconQuads[i*4+3]->setTexture(0, getBitmap("themes/blue/frame.png"),
                             BitmapComponentsRGB_PremultiplyAlpha);
        iconQuads[i*4+3]->setTexture(1, getBitmap("themes/blue/frame.png"),
                             BitmapComponentsAAA);
    }

    // placing the reflection stopper into the text layer
    printf("Blue: Loading reflection stopper\n");
    textLayer->createInstances(2, iconTextInstances);
    iconTextInstances[iconTextInstances.getSize()-1]->createQuads(2, true,
                                                                   iconQuads);
    Quad* refStopper = iconQuads[iconQuads.getSize()-2];
    refStopper->setTexture(0, getBitmap("themes/blue/reflection_stopper.png"),
                                          BitmapComponentsRGBA_PremultiplyAlpha);
    refStopper->set2DRectangle(vec2(0,0), getResolution(), 0);
    Quad* bottomBar = iconQuads[iconQuads.getSize()-1];
    bottomBar->setTexture(0, getBitmap("themes/blue/bottom_bar.png"),
                                         BitmapComponentsRGBA_PremultiplyAlpha);
    bottomBar->set2DRectangle(vec2(0,480-50), vec2(800,46), 0);

    currentApplet = 0;
    position = 0;
    velocity = 0;

    printf("Blue: Loading transitions\n");
    transition = new Transitions::Slime(getApplication()->getProxy(), getResolution());
    printf("Blue: Init complete\n");
}
...
```

5.5.3 Theme update methods

The methods shown in Example 5-28 handle updates:

Example 5-28 Theme update methods

```
...
void Blue::onUpdate(Widget* w)
{
    if (w != this) return;
    float time = getApplication()->getTime();
    static float simTime = 0.0f;
    while (simTime < time)
    {
       if (state == StateAppletRunning)
       {
           velocity = 0;
       }
```

```
        if (state == StateCenteringTargetApplet)
        {
           if (fabs(position-targetApplet) > 0.002f)
               velocity += ((targetApplet-position)*500.f-velocity)*0.1f;
                 else state = StateBrowsing;
                 velocity *= 0.65f;
        }
        else if (state == StateBrowsing)
        {
             if (!isPressed)
             {
                 float closest = floorf(position+0.5f);
                 velocity += (closest-position);
             }
             velocity *= 0.97f;
        }
        position += velocity * 0.001;
        simTime += 0.01f;
    }
    if (position < 0) position = 0;
    if (position > 16) position = 16;

    for (int i = 0; i < iconInstances.getSize(); i++)
    {
        mat4 t;
        t = mat4::translation(int((-position+1)* 265.0f), 0, 0);

        if ((state == StateOpeningTargetApplet)||(state == StateAppletClosing))
        {
            float animTime = getTime()-openingStartTime;
            if (animTime > 0.6f)
            {
                animTime = 0.6f;
            }
            if (state == StateAppletClosing)
            {
                animTime = 0.6f-animTime;
            }

            // Fade neighbour icons left/right
            if (i < floorf(position+0.5f))
            {
                t = mat4::translation(-animTime*animTime*1500.0f, 0, 0) * t;
            }
            else if (i > floorf(position+0.5f))
            {
                t = mat4::translation(+animTime*animTime*1500.0f, 0, 0) * t;
            }
        }

        iconInstances[i]->setTransform(t);
        iconTextInstances[i]->setTransform(t);
     }
}

void Blue::update()
{
    onUpdate(this);
    if (currentApplet) currentApplet->update();
}
...
```

5.5.4 Handling user interface events in the Theme class

The methods in Example 5-29 react to user input:

Example 5-29 Handling user interface events in the Theme class

```
...
void Blue::onTouchDrag(Widget* w, MultiTouch::Event e)
{
    if (w != this) return;
    if (state == StateBrowsing)
    {
        velocity -= float(e.position.x-lastMouseX) * 0.25f;
    }
    lastMouseX = (float)e.position.x;
}

void Blue::onTouchPress(Widget* w, MultiTouch::Event e)
{
    if (w != this) return;
    lastMouseX = (float)e.position.x;
    lastMouseTime = getTime();
    isPressed = true;
}

void Blue::onTouchRelease(Widget* w, MultiTouch::Event e)
{
    isPressed = false;
}

void Blue::onTouchTap(Widget* w, MultiTouch::Event e)
{
    if (w != this) return;

    if( state == StateBrowsing )
    {
        targetApplet = 0;
        state = StateOpeningTargetApplet;
    }
}
...
```

5.5.5 Theme addApplet() method

Example 5-30 shows the `addApplet()` method:

Example 5-30 Theme addApplet() method

```
...
void Blue::addApplet(Applet* applet, AppletCategory category)
{
    applet->hide();
    if (applet->showIconInMenu())
    {
        iconQuads[applets.getSize()*4+0]->setTexture(0, applet->getIconBitmap(),
                    BitmapComponentsRGB_PremultiplyAlpha);
        iconQuads[applets.getSize()*4+0]->setTexture(1,
getBitmap("themes/blue/empty.png"),
```

```
                        BitmapComponentsAAA);
            iconQuads[applets.getSize()*4+1]->
                        setTexture(0, applet->getIconBitmap(),
                           BitmapComponentsRGB_PremultiplyAlpha);
            iconQuads[applets.getSize()*4+1]->
               setTexture(1, getBitmap("themes/blue/empty.png"),
                                 BitmapComponentsAAA);
            appletIconTextures.add(applet->getIconTexture());
        }
        applets.add(applet);
        printf("Minimal: adding applet %s\n", applet->getName().getCharString());
}

void Blue::addAppletIcon(MDE::String desc, AppletCategory category)
{
    iconQuads[0]->getLayer()->addTexture(getBitmap(desc),
                                          BitmapComponentsRGB_PremultiplyAlpha);
}
...
```

5.5.6 Theme createWidgetGroup() method

Example 5-27 on page 5-32 shows the code to create a widget group for the applet:

Example 5-31 Theme createWidgetGroup() method

```
...

WidgetGroup* Blue::createWidgetGroup(Applet* applet, String name)
{
    return new BlueWidgetGroup(getApplication(), applet, name);
}
...
```

5.5.7 Theme runApplet() and closeApplet() methods

Example 5-32 shows the `runApplet()` and `closeApplet()` methods:

Example 5-32 Theme runApplet() and closeApplet() methods

```
...
void Blue::runApplet(Applet* applet)
{
    hide();
    this->currentApplet = applet;
    currentApplet->show();
    currentApplet->reset();
}

void Blue::closeApplet()
{
    // printf("Blue: closing applet %s\n",
    // currentApplet->getName().getCharString());
    if (currentApplet) currentApplet->hide();
    this->currentApplet = 0;
    state = StateAppletClosing;
```

```
    show();
}
...
```

5.5.8 Theme draw() method

Example 5-33 shows the `draw()` method:

Example 5-33 Theme draw() method

```
...
void Blue::draw()
{
    //printf("Blue:draw()\n");

    if (currentApplet)
    {
         currentApplet->draw();
    }
    else
    {
        Applet::draw();

        if (state == StateOpeningTargetApplet)
        {
            // printf("Blue:draw():OpeningTargetApplet (%d)... ",
            // targetApplet);
            if (targetApplet < applets.getSize())
            {
               // printf("%s\n",
               // applets[targetApplet]->getName().getCharString());
                state = StateAppletRunning;
                        runApplet(applets[targetApplet]);
            }
            else
            {
            //    printf(" out-of-range, no applet selected\n");
            }
        }
        if (state == StateAppletClosing)
        {
            state = StateBrowsing;
        }
    }
}
...
```

5.6 Running the lotion application

This section describes the steps to run the lotion application and select the lightshow applet:

1. If you have not done so already, build the lotion project.

   Run the lotion application.

2. A command-line window is displayed that displays the log messages. The application starts building the applet source into applets and loading them. See Figure 5-5:

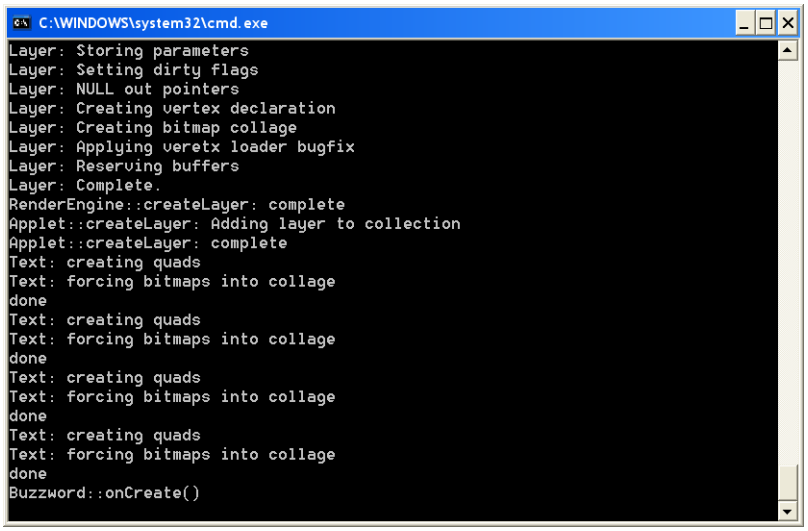

Figure 5-5 Lotion startup messages

3. The lotion graphics window is displayed. See Figure 5-6:

Figure 5-6 lotion window

4. Click and hold the cursor over a blank part of the window. Move the cursor to the left, to simulate swiping a touchpad, until the lightshow icon is visible. See Figure 5-7 on page 5-40:

Figure 5-7 lotion window

5. Click the lightshow icon. The lightshow applet starts as shown in Figure 5-8:

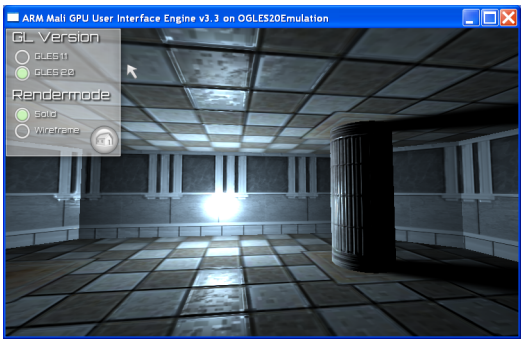

Figure 5-8 Lightshow applet in normal mode

6. Click the **gles1.1** radio button to switch the display to a simulated OpenGL ES 1.1 mode. The image detail is reduced as shown in Figure 5-9:

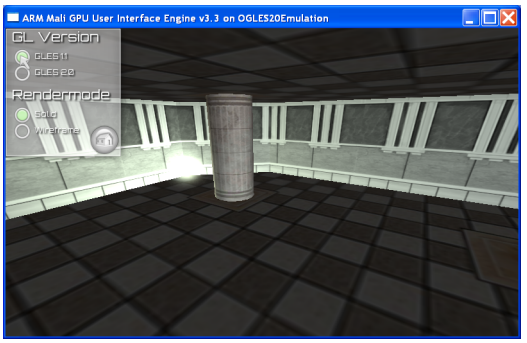

Figure 5-9 Lightshow applet running in OpenGL ES 1.1 mode

7. Click the **wireframe** radio buttons display only the outlines of the primitive shapes. See Figure 5-10 on page 5-41:

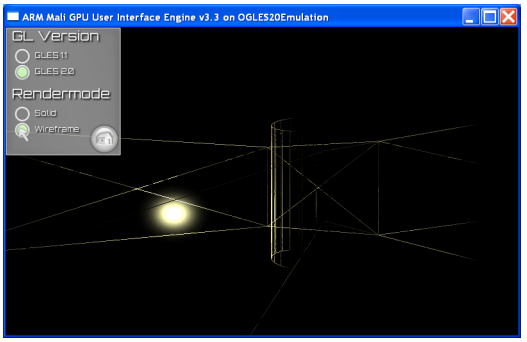

Figure 5-10 Lightshow applet running in wireframe mode

Chapter 6
Tutorial on Constructing Custom Shaders

This chapter describes how to use the Mali GPU User Interface Engine and the MDE library to create custom vertex and fragment shaders. It contains the following sections:

6.1 Overview of the graphics pipeline

Figure 6-1 shows the processing flow for a typical CPU and OpenGL ES 2.0 GPU combination:

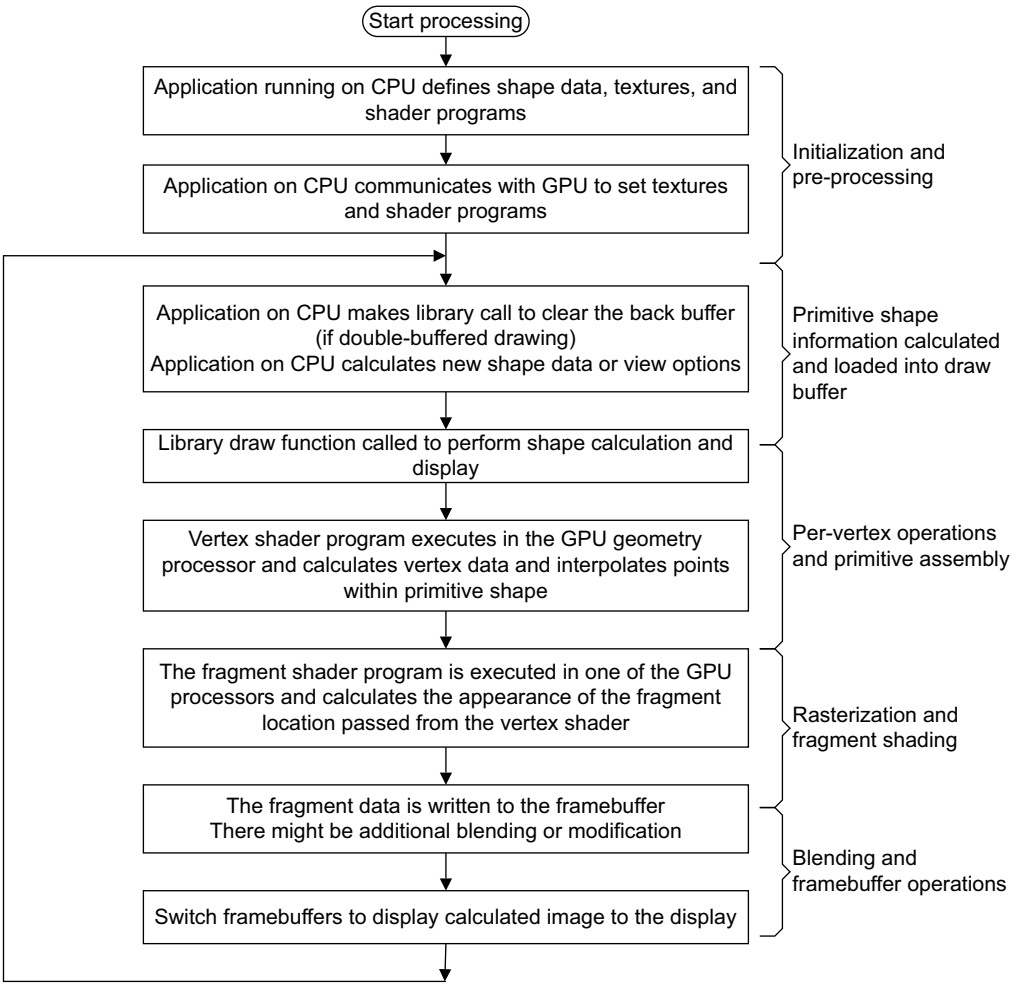

Figure 6-1 Processing flow with GPU shaders

The primary function of the vertex shader is to calculate the position of a point on the primitive shape by interpolation from the vertex locations. This is often called the transformation stage because it typically generates a pixel location based on matrix operations on a 3D shape. It can also:

- read `uniform` and `attribute` values passed to the vertex shader by the application
- create new `varying` constants to pass to the fragment shader
- use matrix transformations to modify the interpolated position
- calculate tangents and normals that will be used by the fragment shader to adjust lighting and color
- calculate the color for a point based on its distance from the vertices and output the intermediate color to the fragment shader.
- modify global variables that determine the position of the fragment.

The fragment shader outputs `gl_FragColor` which is the global color value for the fragment. The shader might just assign a fixed color value to all fragments in the shape but it can also:

- read `uniform` values passed to the fragment shader by the application
- use a texture bitmap to locate the bitmap pixel to use as the color for the fragment
- use the `varying` output values from the vertex shader to modify the fragment color
- use local variables or functions to simplify complex calculations
- use matrix transformations to calculate lighting effects to apply
- use a stencil to return fragment color for points that fit the specified condition.

Figure 6-2 shows the data paths between the application, shader programs, and global context:

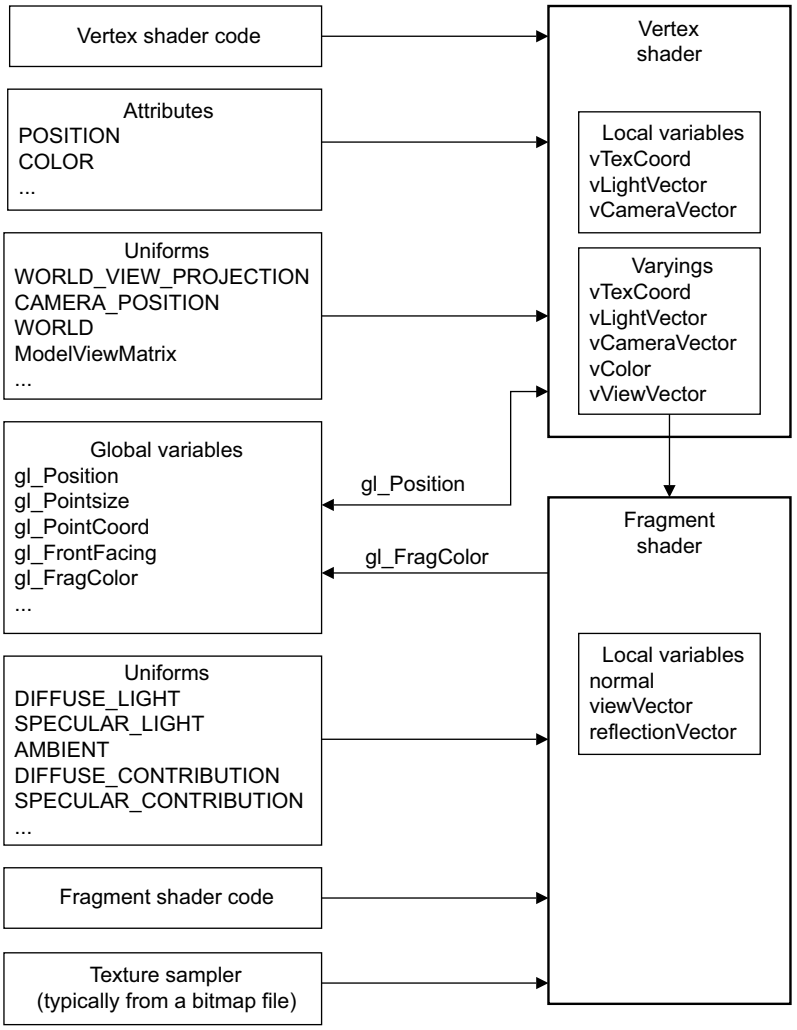

Figure 6-2 Simplified view of shader data flow

The fragment shader might also use matrix transformations to calculate the fragment color based on all of the lighting sources.

To draw a geometric object, for example a chair:

1. All vertices associated with the chair must be known. For 3D graphics, the vertices are represented by x, y, and z coordinates. (If not explicitly specified, the value of w is assumed to be 1 for points in 3D space.)

2. The vertices must be grouped into geometric primitives.

3. The order to draw the primitives must be determined. Breaking down a complex shape into a sequence of primitive shapes is called modeling.

   The chair is now modeled in its own 3D coordinate system. The next steps position that chair in a larger environment and determine how it looks from the camera.

4. Specify where the camera is located in 3D space, where the camera is pointing, where is the top of the camera, and the viewing angle for the camera.

5. A viewing transform calculates where the viewing frustum is in world coordinates.

6. A projection transform maps the vertex to the display space (clip coordinates).

7. Modify the result to produce the perspective effects.

8. The vertex shader uses the interpolated position on the primitive to map the transformed and clipped coordinates to the display window.

9. The fragment shader converts fragments to colored pixels in the frame buffer.

6.2 Data resources used by the shaders

Shader processors can use the following resources to create the graphic:

Programs The vertex and fragment shader processors in the GPU execute programs that are downloaded from the application running on the CPU. The shader code can be either *OpenGL ES Shading Language* (ESSL) source code or pre-compiled binary objects. Different shader code can be used for different shapes, but the code does not change within the primitive drawing stage.

Textures The fragment shaders in the GPU can use a texture, typically from a bitmap, to fill a primitive shape. Multiple textures can be defined and used for different shape primitives.

Global variables

The vertex and fragment shader in the GPU can read global variables associated with the graphics context. Some globals are specific to the vertex shader or fragment shader or can only be written to by one of the shaders. The most commonly used globals are:

- `gl_Position` is only visible to the vertex shader. The shader sets it to the calculated vertex position..
- `gl_FragColor` is only visible to the fragment shader. The shader can set it to the fragment color or execute the `discard` statement to discard the fragment. If set, the fragment color is used in the framebuffer.

Global constants and built-in functions

The library provides global constants and functions that can be used by the shaders such as `gl_MaxVertexAttribs` or `sin()`.

Uniform constants

The uniform constants are global values that apply to the primitive shape. They are read-only, and are either standard OpenGL ES variables or application-defined variables.

Attribute constants

Attribute variables are values passed to the vertex shader on a per-vertex basis. They are read-only, and are either standard OpenGL ES variables or application-defined variables.

Varying variables

Varying variables are values passed to the fragment shader from the vertex shader. They are read-only in the fragment shader. If an vertex shader attribute variable must be used by the fragment shader, the vertex shader must copy the attribute into a new `varying` variable that is shared between the processors.

For points on the face that are not vertices, the GPU calculates the varying value based on the distance of the point from the vertices and the values at the vertices.

Local variables and functions

Each shader can define local variables and functions to hold intermediate values and simplify calculations.

Preprocessor directives

The shader code can contain directives such as `#ifdef` statements to instruct the compiler to use specified declarations for the actual environment. The source code can also contain `//` or `/* */` to identify comment text.

For more information on shader code, see the ESSL specifications.

6.3 Passing data as uniforms

The application defines matrices and vectors, that represent the required transformations and the data for the graphics objects, and pass these values to the shader for further modification and rendering.

Read-only parameters that are passed to shaders are called *uniforms* and are passed by member functions of the `context` class.

6.3.1 Vertex buffer objects

It is efficient to use *Vertex Buffer Objects* (VBO) to pass data for simple shapes to the shaders:

1. The data values, typically the vertices of primitive shapes, is defined in the `VertexBuffer` array as shown in Example 6-1:

Example 6-1 Defining the VertexData array and passing it to the shader

```
// Set up the geometry of the cube
GLfloat vertexData[] = {
            // FRONT
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f, //x,y,z location and u,v texture location
            0.5f, -0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            // BACK
            -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // LEFT
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
            -0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
            -0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // RIGHT
            0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 1.0f,
            // TOP
            -0.5f,  0.5f,  0.5f, 0.0f, 0.0f,
            0.5f,  0.5f,  0.5f, 1.0f, 0.0f,
            -0.5f,  0.5f, -0.5f, 0.0f, 1.0f,
            0.5f,  0.5f, -0.5f, 1.0f, 1.0f,
            // BOTTOM
            -0.5f, -0.5f,  0.5f, 0.0f, 0.0f,
            -0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
            0.5f, -0.5f,  0.5f, 0.0f, 1.0f,
            0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
}
        Managed<Buffer> vertexBuffer =
            context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData),
                      sizeof(GLfloat)*5);
        vertexBuffer->setData(0, sizeof(vertexData), vertexData);
        context->setVertexBuffer(0, vertexBuffer);
```

2. The type of data that is passed is defined in the VertexElement array as shown in Example 6-2 on page 6-8:

Example 6-2 Defining the VertexElement array and passing it to the shader

```
VertexElement elements[2];
elements[0].components = 3;
elements[0].offset = 0;
elements[0].semantic = POSITION;
elements[0].stream = 0;
elements[0].type = GL_FLOAT;
elements[1].components = 2;
elements[1].offset = 3*sizeof(GLfloat);
elements[1].semantic = TEXCOORD0;
elements[1].stream = 0;
elements[1].type = GL_FLOAT;

Managed<VertexDeclaration> vertexDeclaration =
context->createVertexDeclaration(elements, 2);

context->setVertexDeclaration(vertexDeclaration);
```

3. Example 6-3 shows the `WORLD_VIEW_PROJECTION` uniform in the vertex shader:

Example 6-3 Using the VertexBuffer in the vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;
attribute vec4 POSITION;
attribute vec2 TEXCOORD0;

varying vec2 vTexCoord;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    vTexCoord = TEXCOORD0;
}
```

4. The values for `gl_Position` and `vTexCoord` are calculated from the values in the vertex buffer element that is being drawn.

   `vTexCoord` is passed to the fragment shader. The shader uses the texture coordinate together with the fragment sampler uniform to calculate the color value for the pixel as shown in Example 6-4:

Example 6-4 Using the texture sampler in the fragment shader

```
uniform sampler2D texture;
varying vec2 vTexCoord;

void main(void)
{
    gl_FragColor = texture2D(texture, vTexCoord);
}
```

6.3.2 Using asset files

The Vector Buffer Object is appropriate for simple shapes, but applications often use complex shapes or shape data produced by a third-party tool.

Using files in Mali Binary Format improves efficiency and separates graphic creation from graphic display:

1. To load asset files, create a Proxy object for the directory containing the assets. See Example 6-5:

Example 6-5 Creating a proxy

```
Managed<System> system = create_system_backend();
Managed<FileSystem> filesystem = system->createFileSystem("data/");
Proxy proxy(filesystem, context);
```

2. Example 6-6 shows the code that loads the assets from the directory:

Example 6-6 Loading assets from a proxy

```
// load the scene from the asset file
SceneAsset* scene = proxy.getSceneAsset("superpot.MBA");
// load the shader programs
Program* program = proxy.getProgramAsset("default.vert;default.frag")->getProgram();
context->setProgram(program);
// Access parts of the scene by name or id
const char* exName = "Teapot01-mesh";
GeometryAsset* mesh = (GeometryAsset*)scene->getAsset(Asset::TYPE_GEOMETRY, exName);
```

3. The draw loop reads the geometry assets from the file and updates the context as shown in Example 6-7:

Example 6-7 Redrawing geometries from the asset

```
for(unsigned int i = 0; i < scene->getAssetCount(Asset::TYPE_GEOMETRY); i++)
{
    // set the drawing options
    ...
    // calculate the camera position, world, view, and perspective projection
    ...

    // set the uniform values to pass to the shaders
    context->setUniformMatrix("WORLD", 4, 1, world);
    context->setUniformMatrix("VIEW", 4, 1, view);
    context->setUniformMatrix("PROJECTION", 4, 1, proj);
    context->setUniformMatrix("PROJ_VIEW", 4, 1, proj*view);

    // tell the scene asset to draw itself
    ((GeometryAsset*)scene->getAsset(Asset::TYPE_GEOMETRY, i))->draw();

    // update the context to switch the buffers and display the drawing
    context->update();
}
```

For more information on the Mali Binary Format, see the *Mali GPU Binary Asset Exporter User Guide*.

6.3.3 Locations and view transformations as vectors and matrices

Use the `setUniform()` and `setUniformMatrix()` functions to pass a vector or matrix to the shader program.

The code in Example 6-8 passes vectors representing the light and camera position:

Example 6-8 Passing a vector uniform to the shader

```
float angleRad = 23.0f/(180*3.15149265);
vec3 camPos=vec3(75.0f * cos(angleRad), 75.0f * sin(angleRad), 0.0f);
vec3 lightPosition(75.0f, 75.0f, 0.0f);

context->setUniform("LIGHT_POSITION", lightPosition);
context->setUniform("CAMERA_POSITION", camPos);

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glEnable(GL_DEPTH_TEST);

// calculate the camera position
vec3 camPos(60.0f, 0.0f, 0.0f);
context->setUniform("CAM_POS", camPos);
// the world is formed from time-based rotations
mat4 world = mat4::rotation(t*10.0f, vec3(1,1,0))*mat4::rotation(t, vec3(1,0,1));
// the view is constructed from the camera location and viewing orientations
mat4 view = mat4::lookAt( camPos, vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 0.0f, 1.0f) );
// the perspective transform is specified by field-of-view, aspect ratio,
// near plane, and far plane
mat4 proj = mat4::perspective(270.0f, 4.0f/3.0f, 1.0f, 500.0f);

// combine the calculated matrices to a world-view-projection matrix
wvpMatrix = proj * view * world;
// set the uniform values to pass to the shaders
context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);
context->setUniformMatrix("WORLD", world.toMat3());
```

The vertex shader reads the vertex values and calculates the lighting as shown in Example 6-9:

Example 6-9 Using a matrix uniform in the vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;
uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;

attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vNormal;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
```

```
    vec3 pos = WORLD * POSITION.xyz;

    // No point in normalizing before the fragment shader as the normalization will
    // be offset by the interpolation anyway.
    vNormal = WORLD * NORMAL;
    vLightVector = LIGHT_POSITION - pos;
    vCameraVector = CAMERA_POSITION - pos;
}
```

`POSITION` is a element offset into a Vertex Buffer Object. The `attribute POSITION` passed to the shader is an interpolated value based on the vertex positions.

The calculated vectors `vLightVector` and `vCameraVector` are passed to the fragment shader.

6.3.4 Textures

A texture is typically based on a bit-mapped graphic. The code in Example 6-10 loads a texture from a bitmap and passes a sampler to the fragment shader:

Example 6-10 Passing a texture sampler to the fragment shader

```
Program *texturingProgram =
proxy.getProgram("shaders/texturing.vert;shaders/texturing.frag");
Texture2D *texture = proxy.getTexture2D("data/rock_t.png");
texture->setFilterMode(GL_NEAREST, GL_NEAREST);

context->setProgram(texturingProgram);
context->setUniformSampler("texture", static_cast<Texture*>(texture));
```

The uniform is passed to the shader as shown in Example 6-11:

Example 6-11 Using the texture data in the fragment shader

```
uniform sampler2D texture;
varying vec2 vTexCoord;
void main(void)
{
    gl_FragColor = texture2D(texture, vTexCoord);
}
```

`vTexCoord` is a calculated texture coordinate passed from the vertex shader. `gl_FragColor` is the output color that is rendered to the display.

6.4 Minimal shader programs

The shaders are not required to use all of the available uniforms and attributes. Example 6-12 shows a simple vertex shader that assigns the interpolated value in the `POSITION` attribute to the global position variable `gl_Position`:

Example 6-12 Identity vertex shader

```
attribute vec4 POSITION;
void main(void)
{
   gl_Position = POSITION;
};
```

Example 6-13 shows a simple fragment shader that always outputs red as the fragment color:

Example 6-13 Fixed-color fragment shader

```
precision mediump float;
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
};
```

6.4.1 Adding vertex coloring

The vertex shader can define a `varying col` and use it to pass the interpolated vertex color, `COLOR`, to the fragment shader. The `varying col` variable is shared with the fragment shader. See Example 6-14:

Example 6-14 Vertex shader passes interpolated color to fragment shader

```
attribute vec4 POSITION;
attribute vec4 COLOR;
varying vec4 col;
void main(void)
{
    col = COLOR;
    gl_Position = POSITION;
};
```

Example 6-15 shows that the actual assignment to the `gl_FragColor` global color variable is done in the fragment shader:

Example 6-15 Assignment of the color in the fragment shader

```
precision mediump float;
varying vec4 col;
void main(void)
```

```
{
    gl_FragColor = col;
};
```

6.5 Assigning the shader programs

There are the following ways to assign the shader code to a shader:

- *In-line source code*
- *External shader source code*
- *Precompiled shader programs* on page 6-15.

6.5.1 In-line source code

Example 6-16 shows:

- how to define shaders source code in text strings
- using `createShader()` to compile the shader source code
- using `createProgram()` to create a program from the object code
- using `setProgram()` to assign the program to the context.

Example 6-16 Specifying the attributes for the vertex source elements

```
...
    const char* vertexsource =
        "attribute vec4 POSITION; \
        attribute vec4 COLOR;\
        varying vec4 col;\
        void main(void)\
        {\
        col = COLOR;\
        gl_Position = POSITION;\
        }";

    const char* fragmentsource =
        "precision mediump float; \
        varying vec4 col;\
        void main(void)\
        {\
        gl_FragColor = col;\
        }";
    Managed<Shader> vertexshader =
            context->createShader(GL_VERTEX_SHADER, vertexsource);
    Managed<Shader> fragmentshader =
            context->createShader(GL_FRAGMENT_SHADER, fragmentsource);
    Managed<Program> program = context->createProgram(vertexshader, fragmentshader);
    context->setProgram(program);
...
```

6.5.2 External shader source code

Example 6-17 shows loading the shader source code from two files:

Example 6-17 Loading the hello_world shaders

```
...
    Managed<FileSystem> filesystem = system->createFileSystem(".");
    Proxy proxy(filesystem, context);
    Program *helloWorldProgram =
```

```
            proxy.getProgram("shaders/hello_world.vert;shaders/hello_world.frag");
    context->setProgram(helloWorldProgram);
...
```

6.5.3 Precompiled shader programs

The Mali GPU Offline Shader Compiler is a command line tool that translates vertex shaders and fragment shaders written in the OpenGL ES Shading Language (ESSL) into binary vertex shaders and binary fragment shaders that you can link and run on the Mali GPU.

Using the Mali GPU Offline Shader Compiler is optional. You can use the compiler to provide binary shaders which can be distributed without the source. However, using compiled shaders also reduces portability, because a particular binary shader can only be used on one type of GPU.

6.6 Filling a shape with a bitmap texture

The examples in *Minimal shader programs* on page 6-12 use fixed or interpolated colors for the fragment color. To map a bitmap onto the shape:

1. Example 6-18 shows the code to load the bitmap and create a texture:

Example 6-18 Creating a texture from a bitmap file

```
    Managed<FileSystem> filesystem = system->createFileSystem(".");
    Proxy proxy(filesystem, context);
...
    Program *texturingProgram =
        proxy.getProgram("shaders/texturing.vert;shaders/texturing.frag");
    Texture2D *texture = proxy.getTexture2D("data/rock_t.png");
    texture->setFilterMode(GL_NEAREST, GL_NEAREST);
...
```

2. The semantic entry in the `VertexElement` array must specify `TEXTCOORD0` as shown in Example 6-19:

Example 6-19 Modifying the VertexElement array to use a texture

```
...
    Managed<Buffer> vertexBuffer =
        context->createBuffer(GL_ARRAY_BUFFER, sizeof(vertexData), sizeof(GLfloat)*5);
    vertexBuffer->setData(0, sizeof(vertexData), vertexData);
    context->setVertexBuffer(0, vertexBuffer);

    VertexElement elements[2];
    elements[0].components = 3;
    elements[0].offset = 0;
    elements[0].semantic = POSITION;
    elements[0].stream = 0;
    elements[0].type = GL_FLOAT;
    elements[1].components = 2;
    elements[1].offset = 3*sizeof(GLfloat);
    elements[1].semantic = TEXCOORD0;
    elements[1].stream = 0;
    elements[1].type = GL_FLOAT;

    Managed<VertexDeclaration> vertexDeclaration =
        context->createVertexDeclaration(elements, 2);
    context->setVertexDeclaration(vertexDeclaration);
...
```

3. Example 6-20 shows the use of `setProgram()` to connect the texturing program from Example 6-18 to the context:

Example 6-20 Create a texture program and a uniform sampler to use the bitmap

```
...
    context->setProgram(texturingProgram);
    context->setUniformSampler("texture", static_cast<Texture*>(texture));
...
```

4. The vertex shader listed in Example 6-21 performs the following actions:
   - It calculates the output position, `gl_Position`, based on a world view projection and the current value of `POSITION`.

     This position calculation in Example 6-21 is a simple matrix multiplication to enable viewing the object from different angles, but more complex transformations can be performed in the vertex shader.
   - It assigns the current of `TEXCOORD0` to the varying `vTexCoord`.

     The fragment shader does not have access to `TEXCOORD0`, so the vertex shader must create a variable to pass the value to the fragment shader. In Example 6-21, the vertex coordinate is just directly copied, but the vertex shader can also modify the coordinate before passing it to the fragment shader.

Example 6-21 Calculating gl_Position in the vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;

attribute vec4 POSITION;
attribute vec2 TEXCOORD0;

varying vec2 vTexCoord;

void main(void)
{
    // calculate the actual position based on the interpolated POSITION attribute
    // and the uniform WORLD_VIEW_PROJECTION
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    // the texture coordinate is interpolated as passed as the TEXTCOORD0 attribute
    vTexCoord = TEXCOORD0;
}
```

5. When the fragment shader receives the texture coordinate from the vertex shader, it assigns a value from the bitmap sampler to `gl_FragColor` as shown in Example 6-22:

Example 6-22 Fragment shader code to read the texture value

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

// texture uniform was set up by the application
uniform sampler2D texture;
// the texture coordinate was passed from the vertex shader
varying vec2 vTexCoord;

void main(void)
{
    // assign color by looking at the specified coordinate in the texture
    gl_FragColor = texture2D(texture, vTexCoord);
}
```

6.7 Performing matrix transformations in the shaders

This section describes how the matrix and vector values defined in the draw loop are modified and rendered in the shaders.

6.7.1 Example of a simple world-view transformation

The application code in Example 6-23 calculates the camera position, camera target, camera up vector, world, projection, view, and world view projection:

Example 6-23 Simple projection matrices in the application code

```
static const float PI = 3.14159265;

/**
* define a function that calculates the camera position given an angle and a radius.
* This function enables the camera to rotate around the scene object.
*/
inline vec3 calculateCamPos(float radius, float angle)
{
    float angleRad = angle / 180 * PI;
    return vec3(radius * cos(angleRad), 2.0f, radius * sin(angleRad));
}

int main(int argc, char * argv[])
{
...
        // Initialize the demo engine classes and load the shader code
         ...

        // Set up the geometry of the cube by specifying values for the vertex buffer.
         ...

        // The camera position, target and up vector used for the view matrix
        float cameraAngle = 45.0f;
        vec3 camPos = calculateCamPos(4.0f, cameraAngle);
        vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
        vec3 upVector = vec3(0.0f, 1.0f, 0.0f);

        // Set up the view and projection matrices and multiply them to get the
        // modelviewprojection matrix.

        // No requirement to change the position, rotation, or
        // scaling of the cube we don't need a world matrix.
        mat4 world = mat4::identity();

        // projection matrix for matrix
        mat4 proj = mat4::perspective(30.0f, 1.0f, 0.1f, 20.0f);

        // use lookAt to create the view from camera to target
        mat4 view = mat4::lookAt(camPos, camTarget, upVector);

        // modify the view to use perspective
        mat4 wvpMatrix = proj * view;
...
```

The application code in Example 6-24 on page 6-19 uses `lookAt()` and `rotate()` to modify the world-view projection matrix and pass it to the vertex shader:

Example 6-24 Transformations in the draw loop

```
...
        // execute the draw loop
        glEnable(GL_DEPTH_TEST);
        do
        {
            // use the keyboard to change the image
            if (keyboard->getSpecialKeyState(Keyboard::KEY_SPACE))
            {
            // Rotate camera
               if (keyboard->getSpecialKeyState(Keyboard::KEY_LEFT))
                {
                    // Move camera (by rotating it) to create a new camera angle
                    camPos = calculateCamPos(4.0f, ++cameraAngle);
                    // create a transformation matrix based on the new relationship
                    // camera has changed, so view must be updated
                    view = mat4::lookAt(camPos, camTarget, upVector);
                }
                ...
            }
            else
            {
            // Rotate object
                if (keyboard->getSpecialKeyState(Keyboard::KEY_RIGHT))
                {
                    // rotate the object itself about the specified vector
                    // this changes object orientation, but camera does not move
                    world *= mat4::rotation(1.0f, vec3(0.0f, 1.0f, 0.0f));
                }
                ...
            }

            // calculate a new world view projection based on the camera, view, and
            // the world containing the object
            wvpMatrix = proj * view * world;

            // pass the calculated matrix as a uniform to the vertex shader
            // the first parameter is the name of the uniform variable
            // the second parameter is the actual matrix data
            context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);

            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

            // Draw the object
            context->drawArrays(GL_TRIANGLE_STRIP, 0, 2);
            ...
        }while ( context->update() );
}
```

The vertex shader code in Example 6-25 calculates the position of a fragment based on the interpolated `POSITION` value and the `WORLD_VIEW_PROJECTION` uniform:

Example 6-25 Simple world view projection calculation in vertex shader

```
attribute vec4 POSITION;            // position of a point on the primitive shape
attribute vec4 COLOR;               // interpolated color from vertex colors
uniform mat4 WORLD_VIEW_PROJECTION; // transformation matrix passed as a uniform
varying vec4 vColor                 // parameter to pass to the fragment shader
```

```
void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    vColor = COLOR;                  // fragment shader assigns this to gl_FragColor
}
```

6.7.2 Example showing calculation of specular reflections

The application code in Example 6-23 on page 6-18 uses an interpolated vertex color for the fragment. The example in Example 6-26 calculates the light based on the vectors that specify the different light sources:

Example 6-26 Context initialization in lighting application code

```
...
    // define constants for the light colors
static const vec3 AMBIENT_LIGHT = vec3 (0.1f, 0.1f, 0.1f);
static const vec3 DIFFUSE_LIGHT = vec3(0.5f, 0.5f, 0.5f);
static const vec3 SPECULAR_LIGHT = vec3(0.5f, 0.5f, 0.5f);
static const vec4 COLOR = vec4(1.0f, 0.0f, 0.0f, 1.0f);
...
    // define the calculateCamPos that returns a vec3 given an angle and radius
    // enables the camera to rotate about the scene object
...
    // initialize the context and MDE classes
...
    // load the shader code
...
    // Loading scene from a Mali Binary Asset file
    SceneAsset* scene = proxy.getSceneAsset("cube.mba");
    // get the asset from the mba file and assign it to the cube_asset variable
    GeometryAsset* cube_asset =
        static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));

    Texture2D *diffuseTexture =
        proxy.getTexture2DAsset("rock_t.png")->getTexture2D();

    Texture2D *normalMap =
        proxy.getTexture2DAsset("rock_n.png")->getTexture2D();

    // Initialize the camera position, target, and up vector
    float cameraAngle = 0.0f;
    vec3 camPos = calculateCamPos(75.0f, cameraAngle);
    vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
    vec3 upVector = vec3(0.0f, 0.0f, 1.0f);

    // Set up the view, projection and world matrices and multiply them to get the
    // modelviewprojection and modelview matrices.
    mat4 view = mat4::lookAt(camPos, camTarget, upVector);
    mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 500.0f);
    mat4 world = mat4::identity();

    // create the transformation matrix
    mat4 wvpMatrix = proj * view * world;

    // define the light position
    vec3 LightPosition(75.f, 75.f, 0.0f)
...
```

Lighting calculation in the vertex shader

The vertex shader code in Example 6-27 calculates the position of a fragment based on the interpolated POSITION and lighting vectors from the transformations:

Example 6-27 Specular lighting calculation in vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;
// the light at each location depends on the relative angles between light position,
point location, camera angle, and lighting colors
uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;
// interpolated values for this point on primitive shape
attribute vec4 POSITION;
attribute vec3 NORMAL;
// light values to pass to fragment shader
varying vec3 vDiffuse;
varying vec3 vSpecular;

void main(void)
{
    // calculate the position based on the transformation
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    // vector variables used to calculate lighting
    vec3 normal = normalize(WORLD * NORMAL);
    vec3 pos = WORLD * POSITION.xyz;
    // determine the angle from the light to the shape position
    vec3 lightVector = normalize(LIGHT_POSITION - pos);
    // calculate a multiplier for the diffuse light
    float nDotL = max(dot(normal, lightVector), 0.0);
    vDiffuse = DIFFUSE_LIGHT * nDotL;

    // No point in calculating specular reflections from the backside of vertices.
    float specPow = 0.0;
    if (nDotL > 0.0)
    {
        normalize the camera position and calculate the light reflection
        vec3 cameraVector = normalize(CAMERA_POSITION - pos);
        vec3 reflectVector = reflect(-cameraVector, normal);
        // determine the amount of reflected light
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 4.0);
    }
    // assign the specular light based on the specular uniform and
    // the computed multiplier
    vSpecular = SPECULAR_LIGHT * specPow;
}
```

As shown in Example 6-28, the fragment shader simply combines the light sources:

Example 6-28 Fragment shader adds specular light to background light

```
uniform vec3 AMBIENT_LIGHT;
uniform vec4 COLOR;

varying vec3 vDiffuse;  // calculated diffuse component
varying vec3 vSpecular; // calculated specular component based on light position
```

```
void main(void)
{
    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * COLOR +
                   vec4(vDiffuse, 1.0)      * COLOR +
                   vec4(vSpecular, 1.0);
}
```

Lighting calculation in the fragment shader

The vertex shader code in Example 6-29 passes more of the calculation to the fragment shader:

Example 6-29 Specular lighting calculation in vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;

uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;

attribute vec4 POSITION;
attribute vec3 NORMAL;

varying vec3 vNormal;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;

    vec3 pos = WORLD * POSITION.xyz;

    // No point in normalizing before the fragment shader as the normalization will
    // be offset by the interpolation anyway.
    vNormal = WORLD * NORMAL;
    vLightVector = LIGHT_POSITION - pos;
    vCameraVector = CAMERA_POSITION - pos;
}
```

The fragment shader in Example 6-30 calculates the reflection vectors before combining the light sources:

Example 6-30 Fragment shader adds specular light to background light

```
uniform vec3 AMBIENT_LIGHT;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;
uniform vec4 COLOR;

varying vec3 vNormal;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    vec3 normal = normalize(vNormal);
```

```
    vec3 lightVector = normalize(vLightVector);

    float nDotL = max(dot(normal, lightVector), 0.0);
    vec3 diffuse = DIFFUSE_LIGHT * nDotL;

    float specPow = 0.0;
    if (nDotL > 0.0)
    {
        vec3 cameraVector = normalize(vCameraVector);

        vec3 reflectVector = reflect(-cameraVector, normal);
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 16.0);
    }

    vec3 specular = SPECULAR_LIGHT * specPow;

    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * COLOR +
                   vec4(diffuse, 1.0)       * COLOR +
                   vec4(specular, 1.0);
}
```

6.7.3 Example showing calculation of bump-mapped reflections

The application code in Example 6-29 on page 6-22 uses an interpolated vertex color for the fragment. To use normal and diffuse texture samplers:

1. The example in Example 6-31 uses normal and diffuse texture samplers from bitmaps to modify the reflections:

Example 6-31 Context initialization in bump map application code

```
    // initialize the context and MDE classes
...
    // load the shader code
...
    // Loading scene from a Mali Binary Asset file eliminates the requirement to
    // construct an arrow of primitive shapes
    SceneAsset* scene = proxy.getSceneAsset("cube.mba");
    // get the asset from the mba file and assign it to the cube_asset variable
    GeometryAsset* cube_asset =
        static_cast<GeometryAsset*>(scene->getAsset(Asset::TYPE_GEOMETRY, 0));

    // this bitmap creates a texture used with the diffuse light calculation
    Texture2D *diffuseTexture =
        proxy.getTexture2DAsset("rock_t.png")->getTexture2D();

    // this bitmap creates a texture used with the specular light calculation
    Texture2D *normalMap =
        proxy.getTexture2DAsset("rock_n.png")->getTexture2D();

    // Initialize the camera position, target, and up vector
    float cameraAngle = 0.0f;
    vec3 camPos = calculateCamPos(75.0f, cameraAngle);
    vec3 camTarget = vec3(0.0f, 0.0f, 0.0f);
    vec3 upVector = vec3(0.0f, 0.0f, 1.0f);

    // Set up the view, projection and world matrices and multiply them to get the
    // modelviewprojection and modelview matrices.
    mat4 view = mat4::lookAt(camPos, camTarget, upVector);
    mat4 proj = mat4::perspective(60.0f, 4.0f/3.0f, 1.0f, 500.0f);
```

```
    mat4 world = mat4::identity();

    // create the transformation matrix
    mat4 wvpMatrix = proj * view * world;

    // define a light position that will be used with specular reflections
    vec3 lightPosition(75.0f, 75.0f, 0.0f);

...
    // Initialize the uniforms
    context->setUniform("LIGHT_POSITION", lightPosition);
    context->setUniform("CAMERA_POSITION", camPos);
    context->setUniform("AMBIENT_LIGHT", AMBIENT_LIGHT);
    context->setUniform("DIFFUSE_LIGHT", DIFFUSE_LIGHT);
    context->setUniform("SPECULAR_LIGHT", SPECULAR_LIGHT);
    context->setUniformSampler("diffuseTexture", diffuseTexture);
    context->setUniformSampler("normalMap", normalMap);
...
```

2. The vertex shader code in Example 6-32 is similar to the code in Example 6-27 on page 6-21, except that the `WORLD` uniform is also passed to the shader:

Example 6-32 Lighting initialization in bump map application code

```
...
    // Enable depth test and start drawing loop
    glEnable(GL_DEPTH_TEST);
    do
    {
        // use keyboard input to rotate the camera or the object
...
        wvpMatrix = proj * view * world;
        // set uniforms that have changed since the last draw
        context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);
        context->setUniformMatrix("WORLD", world.toMat3());

        // Clear the screen and draw cube_asset with lighting
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );

        // call the assets own draw method to render the primitives
        cube_asset->draw();
    }while ( context->update() );
...
```

3. The vertex shader code in Example 6-33 calculates the position of a fragment based on the interpolated `POSITION` value and a world view projection:

Example 6-33 Lighting calculation in bump map vertex shader

```
uniform mat4 WORLD_VIEW_PROJECTION;
uniform mat3 WORLD;

uniform vec3 LIGHT_POSITION;
uniform vec3 CAMERA_POSITION;

attribute vec4 POSITION;
attribute vec3 NORMAL;
attribute vec3 TANGENT;
```

```
attribute vec3 BINORMAL;
attribute vec3 TEXCOORD0;

varying vec2 vTexCoord;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
    vTexCoord = TEXCOORD0.st;

    vec3 pos = WORLD * POSITION.xyz;
    mat3 TBN = WORLD * mat3(normalize(TANGENT), normalize(BINORMAL),
                        normalize(NORMAL));

    // No point in normalizing before the fragment shader as the normalization will
    // be offset by the interpolation anyway.
    vLightVector = (LIGHT_POSITION - pos) * TBN;
    vCameraVector = (CAMERA_POSITION - pos) * TBN;
}
```

4. Example 6-34 shows the fragment shader combining the light sources:

Example 6-34 Summing light values in bump map fragment shader

```
uniform mat3 M_MATRIX;
uniform sampler2D diffuseTexture;
uniform sampler2D normalMap;
uniform vec3 AMBIENT_LIGHT;
uniform vec3 DIFFUSE_LIGHT;
uniform vec3 SPECULAR_LIGHT;

uniform float DIFFUSE_CONTRIBUTION;
uniform float SPECULAR_CONTRIBUTION;
uniform float AMBIENT;

varying vec2 vTexCoord;
varying vec3 vLightVector;
varying vec3 vCameraVector;

void main(void)
{
    // read the normal value for this point from the texture
    vec3 normal = normalize(texture2D(normalMap, vTexCoord).rgb * 2.0 - 1.0);
    // read the color for this point from the texture
    vec4 color = texture2D(diffuseTexture, vTexCoord);

    // calculate the intensity of the diffuse light reflected off the point
    vec3 lightVector = normalize(vLightVector);
    float nDotL = max(dot(normal, lightVector), 0.0);
    vec3 diffuse = DIFFUSE_LIGHT * nDotL;

    // calculate the multiplier for the specular reflection
    float specPow = 0.0; // was 0
    // if the nDotL value is less than 0, use 0.0 for the specular multiplier
    if (nDotL > 0.0)
    {
        vec3 cameraVector = normalize(vCameraVector);
        // what is the reflection based on the normal and the camera view
        vec3 reflectVector = reflect(-cameraVector, normal);
```

```
        specPow = pow(max(dot(reflectVector, lightVector), 0.0), 4.0);
    }
    // calculate the reflected light
    vec3 specular = SPECULAR_LIGHT * specPow;
    // add the light values for the point together and assign the result to the color
    gl_FragColor = vec4(AMBIENT_LIGHT, 1.0) * color + // was 1.0
                   vec4(diffuse, 1.0) * color + // was 1.0
                   vec4(specular, 1.0); //was 1.0
}
```

Appendix A
Matrix and Vector Operations

This chapter provides an overview of matrix and vector operations and describes the matrix and vector functions in the MDE library. It contains the following section:

A.1 Matrix and vector functions in the MDE library

The following matrix functions in the MDE library simplify transformations on matrices:

`matn determinant()`

Returns the determinate for the *n*x*n* identity matrix. *n* is in the range 2 to 4.

`matn identity()`

Returns a *n*x*n* identity matrix. *n* is in the range 2 to 4.

`matn invert()`

Attempts to calculate the inverse of the matrix. *n* is in the range 2 to 4.

```
mat4 matrix = mat4::scale(2.0f, 2.0f, 2.0f);
mat4 invertedMatrix;
bool isInvertable = matrix.invert(invertedMatrix);
```

`mat4 lookAt()`

Creates a 4x4 lookAt transform matrix, constructed from the given camera position, camera target, and camera up vector.

```
mat4 lookAtMatrix = mat4::lookAt(vec3(100.0f, 100.0f, 100.0f), //camera
                                 vec3(0.0f, 0.0f, 0.0f),       // target
                                 vec3(0.0f, 1.0f, 0.0f));      // up
```

`matn operator float* ()`

Converts a matrix into an array of floats. This enables passing a mat4 as a parameter to a function that expects a float array:

```
mat4 matrix;
glUniformMatrix4fv(loc, 4, GL_FALSE, matrix);
```

`matn operator=(float* d)`

Returns a reference to the matrix element identified by the index.

```
mat4 matrix;
matrix[0] = 2.0f;
```

`float matn operator[] (int index)`

Returns a reference to the matrix element identified by the index.

```
mat4 matrix;
matrix[0] = 2.0f;
```

`float m[16]`

Returns an array of floats that contains the entries of the 4x4 matrix.

`matn operator*`

If the parameter is a *n*x*n* matrix, performs matrix multiplication between the matrix and another *n*x*n* matrix. This results in a new *n*x*n* matrix. *n* is in the range 2 to 4. For example:

```
mat4 m1, m2;
mat4 result = m1*m2;
```

Multiplying a 4x4 perspective projection matrix by the 4x4 matrix returned from a `lookAt()` call results in a world-view projection matrix.

If the parameter is a *n*-element vector, performs matrix multiplication between the matrix and the vector. This results in a new *n*-element vector. For example:

```
mat4 matrix;
vec4 vector;
vec4 result = matrix*vector;
```

`mat4 perspective()`

Creates a 4x4 perspective transform matrix, constructed from the given field of view, aspect angle, near plane, and far plane.

```
mat4 perspectiveMatrix = mat4::perspective(45.0f,     // field
                                           4.0f/3.0f, // aspect
                                           1.0f,      // near plane
                                           1000.0f);  // far plane
```

The perspective matrix is typically used with the camera view to create the world view:

```
mat4 world = mat4::identity();  // could be a translation or rotation
mat4 proj = mat4::perspective(30.0f, 1.0f, 0.1f, 20.0f);
mat4 view = mat4::lookAt(camPos, camTarget, upVector);
mat4 wvpMatrix = proj * view * world;
context->setUniformMatrix("WORLD_VIEW_PROJECTION", wvpMatrix);
```

The vertex shader uses the world view to transform each point:

```
attribute vec4 POSITION;      // position of a point on the shape
WORLD_VIEW_PROJECTION;        // transformation matrix passed
void main(void)
{
    gl_Position = WORLD_VIEW_PROJECTION * POSITION;
}
```

`mat`*`n`* `rotation()`

Creates a *n*x*n* rotation matrix, calculated based on the passed axis and angle parameters. *n* is in the range 2 to 4.

```
mat4 matrix = mat4::rotation(45.0,       // angle to rotate
                             vec3(1.0f, 0.0f, 0.0f)); // rotation axis
```

`mat`*`n`* `scale()`

Creates a *n*x*n* scale matrix. *n* is 3 or 4.

```
mat4 matrix = mat4::scale(2.0f, 2.0f, 2.0f);  // x, y, z scale factors
```

`mat4 toMat3()`

Creates a new matrix from the upper left 3x3 elements of the original matrix.

```
mat4 m1;
mat3 m2 = m1.toMat3();
```

`mat4 translation()`

Creates a 4x4 translation matrix. The parameters can be a vector or the x, y, and z translation values:

```
vec3 trans_vec = vec3(2.0f, 3.0f, 4.0f);
mat4 matrix = mat4::translation(2.0f, 3.0f, 4.0f);
mat4 matrix = mat4::translation(trans_vec);
```

The following vector functions in the MDE library simplify vector operations:

`vec`*`n`* `crossProduct()`

Returns the cross product of two vectors. For two 3D vectors, the cross product results in a vector that is perpendicular to both of the original vectors.

`vec`*`n`* `length()`

Returns the length of the vector.

`vec`*`n`* `scale()` Scale the vector by the elements of a vector or by a constant.

`vec`*n* `normalized()`

Returns the vector normalized to have a length of 1.0.

`vec`*n* `operator*`

If used with a vector, Performs *n*-component dot product between the vector and the parameter vector.

If used with a constant, Multiplies all vector components with the given scalar parameter, and returns new 4-component vector.

There are versions for 3 and 4-element vectors.

`vec`*n* `operator+`

Performs component-wise vector addition of the vector and the given parameter vector, and returns the result. There are versions for 3 and 4-element vectors.

Note

For a full list of MDE library functions and their parameters, open the `index.html` file in the *`install_dir`*`\Mali GPU Engine `*`x.y`*`\docs\html` directory.

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.