Arm® Guide for Unreal Engine 4

Version 1.1

Optimizing Mobile Gaming Graphics

arm

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
100959_0101_00_en

Arm® Guide for Unreal Engine 4

Optimizing Mobile Gaming Graphics

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	23 May 2017	Non-Confidential	First release of version 1.0
0101-00	16 November 2019	Non-Confidential	First release of version 1.1

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

Copyright © 2017, 2019 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

www.arm.com

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents
Arm® Guide for Unreal Engine 4 Optimizing Mobile Gaming Graphics

Preface

About this book 7
Feedback 9

Chapter 1 Introduction

1.1 About Arm® Mali™ GPUs 1-11
1.2 About optimization 1-12
1.3 About Unreal Engine 4 1-13
1.4 About Arm® tools 1-14
1.5 About the Circuit VR demo 1-15

Chapter 2 The optimization process

2.1 The optimization process 2-18
2.2 Take measurements 2-19
2.3 Locate the bottleneck 2-20
2.4 Determine the optimization 2-21
2.5 Apply the optimization 2-22
2.6 Verify the optimization 2-23
2.7 Repeat the optimization process 2-24
2.8 General optimization advice 2-25
2.9 Bottlenecks move between processors 2-28

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3 **Optimizations and optimization techniques**
3.1 Optimization best practices 3-31
3.2 Adaptive Scalable Texture Compression 3-35
3.3 Unreal Engine best practices 3-36
3.4 Cubemaps 3-40

Chapter 4 **Virtual reality optimization techniques**
4.1 Virtual reality development best practices 4-52
4.2 Monoscopic far field rendering for mobile VR 4-56
4.3 Understanding Multiview 4-58

Chapter 5 **Arm Mobile Studio**
5.1 Introduction to Arm Mobile Studio 5-63
5.2 Introduction to Streamline 5-64
5.3 Introduction to Graphics Analyzer 5-69

Appendix A **Revisions**
A.1 Revisions Appx-A-81

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm® Guide for Unreal Engine 4 Optimizing Mobile Gaming Graphics*.

It contains the following:

- *About this book* on page 7.
- *Feedback* on page 9.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this book

This book is designed to help you create applications and content that make the best use of Unreal Engine 4 on mobile platforms, especially those with Arm® Mali™ GPUs.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This guide is for beginner to intermediate software developers working with Unreal Engine 4.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter introduces Arm Guide for Unreal Engine.

Chapter 2 The optimization process
This chapter describes the optimization process.

Chapter 3 Optimizations and optimization techniques
This chapter describes optimizations and optimization techniques.

Chapter 4 Virtual reality optimization techniques
This chapter describes optimization techniques for *Virtual Reality* (VR) applications.

Chapter 5 Arm Mobile Studio
This chapter discusses Arm Mobile Studio and how the Graphics Analyzer and Streamline components can improve your development when using Unreal Engine 4.

Appendix A Revisions
This appendix describes the changes between released issues of this book.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes 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.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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>
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

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

Arm publications

Developer resources

Arm Developer, *https://developer.arm.com/graphics/*.
Arm Mobile Studio, *https://developer.arm.com/mobile-studio*.
Arm Mali Graphics Analyzer, *https://developer.arm.com/tools-and-software/graphics-and-gaming/armmobile-studio/components/graphics-analyzer*.
Arm Streamline Performance Analyzer, *https://developer.arm.com/tools-and-software/embedded/arm-development-studio/components/streamline-performance-analyzer*.
Midgard performance counters, *https://community.arm.com/graphics/b/blog/posts/mali-midgard-family-performance-counters*.
Bifrost performance counters, *https://community.arm.com/graphics/b/blog/posts/mali-bifrost-family-performance-counters*.
Circuit VR, *https://community.arm.com/graphics/b/blog/posts/circuit-vr-a-post-mortem-in-a-lilliputian-world*.

Other publications

Unreal Engine documentation, *https://docs.unrealengine.com/latest/INT*

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier 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 *Arm Guide for Unreal Engine 4 Optimizing Mobile Gaming Graphics*.
- The number 100959_0101_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 1
Introduction

This chapter introduces Arm Guide for Unreal Engine.

It contains the following sections:

- *1.1 About Arm® Mali™ GPUs* on page 1-11.
- *1.2 About optimization* on page 1-12.
- *1.3 About Unreal Engine 4* on page 1-13.
- *1.4 About Arm® tools* on page 1-14.
- *1.5 About the Circuit VR demo* on page 1-15.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.1 About Arm® Mali™ GPUs

Arm Mali™ GPUs are designed for mobile or embedded devices. Arm Mali GPUs are divided into the following families:

Bifrost GPUs

The Bifrost GPUs have unified shader cores that perform vertex, fragment, geometry, tessellation, and compute processing. They are used for graphics and compute applications with Vulkan, OpenGL ES 1.1 to OpenGL ES 3.2, and OpenCL 1.2 Full Profile.

Midgard GPUs

The Bifrost GPUs have unified shader cores that perform vertex, fragment, geometry, tessellation, and compute processing. They are used for graphics and compute applications with Vulkan, OpenGL ES 1.1 to OpenGL ES 3.2, and OpenCL 1.2 Full Profile.

Utgard GPUs

The Utgard GPUs have a vertex processor and one or more fragment processors. They are used for graphics only applications with OpenGL ES 1.1 and 2.0.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.2 About optimization

Graphics is about making things look good. Optimization is about making things look good with the least computational effort. This is especially important for mobile devices that keep power consumption low by limiting computing power and memory bandwidth.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.3 About Unreal Engine 4

Unreal Engine 4 is a suite of tools for creating games, simulations, and films. You can use it for anything from small mobile projects up to leading-edge games.

Unreal Engine provides everything you require to build games and other applications, without using external plug-ins or tools.

It provides built-in modular tools and customizable plug-ins for you to tailor the system to your own requirements.

Unreal Engine is widely used in the games industry and provides features including:

- Production ready.
- VR support.
- Blueprints for rapid prototyping.
- Full source for customization.
- Photo-realistic rendering.
- Sequencing and life-like animation.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.4 About Arm® tools

Arm tools provide you with ways to analyze the behavior of your applications and locate performance issues.

Arm Mobile Studio is a software suite that enables you to visualize performance data of your Android game or application. Identify bottlenecks with Arm Streamline, then use graphics application tracing with Graphics Analyzer to determine exactly where rendering defects occur.

Easy setup with non-root support

All of the tools that are included with Arm Mobile Studio are designed to work on a non-rooted Android device. This enables you to profile and debug your game without device modification.

Detect bottlenecks easily

Use Arm Streamline to visualize performance data from the system. Easy-to-use templates help you select the most appropriate set of counters for your target device, so you get the information that you need to optimize your application and filter out the rest of the data.

Optimize for mobile faster, including for 64-bit, quickly determining whether your performance bottleneck relates to the CPU or GPU using easy-to-interpret charts.

Where performance issues are related to the CPU, you can pinpoint the biggest problem areas in your code by drilling down through threads and functions calls, right down to line-byline source code analysis.

Identify graphics issues

Use Graphics Analyzer to view all the graphics API calls in your application. Easily trace OpenGL ES and Vulkan API calls and understand frame-by-frame the effect on application performance.

The tool captures state changes throughout the application, so you can easily identify rendering defects and trace back to the triggering API function call which caused the problem to occur.

Render the scene draw call by draw call, to see exactly how a frame in the application is composed. This enables you to easily pinpoint any graphical defects in the application and to spot any inefficient or redundant draw calls.

Accelerate shader programs

Use Mali Offline Compiler to compile all shaders and kernels from OpenGL ES, Vulkan and Open CL. Mali Offline Compiler performance reports provide easy visibility of the expected performance and the likely performance bottlenecks of your shader programs on any of the available Mali GPU targets.

Get Arm Mobile Studio

Download Arm Mobile Studio and find the learning resources and the tools that are available for Unreal Engine 4 at *https://developer.arm.com/ue4*. To read more about Arm Mobile Studio visit *https://developer.arm.com/mobile-studio*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.5 About the Circuit VR demo

Circuit VR is a demo produced by Arm using the Unreal Engine.

The development team made use of Mali Graphics Debugger and *Arm Streamline* on page 5-64 to analyze the demo and they used many of the techniques and optimizations in this guide to enhance the performance of the demo.

Designed specifically for virtual reality, Circuit VR miniaturizes you to the size of a processor chip, and teleports you to inside a smartphone. You can explore the insides of the device, as if you were there.

For more information, see: *https://community.arm.com/graphics/b/blog/posts/circuit-vr-a-post-mortem-in-a-lilliputian-world*.

The following image shows the insides of a smartphone:

Figure 1-1 Insides of a smartphone

The following image shows smart phone components:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

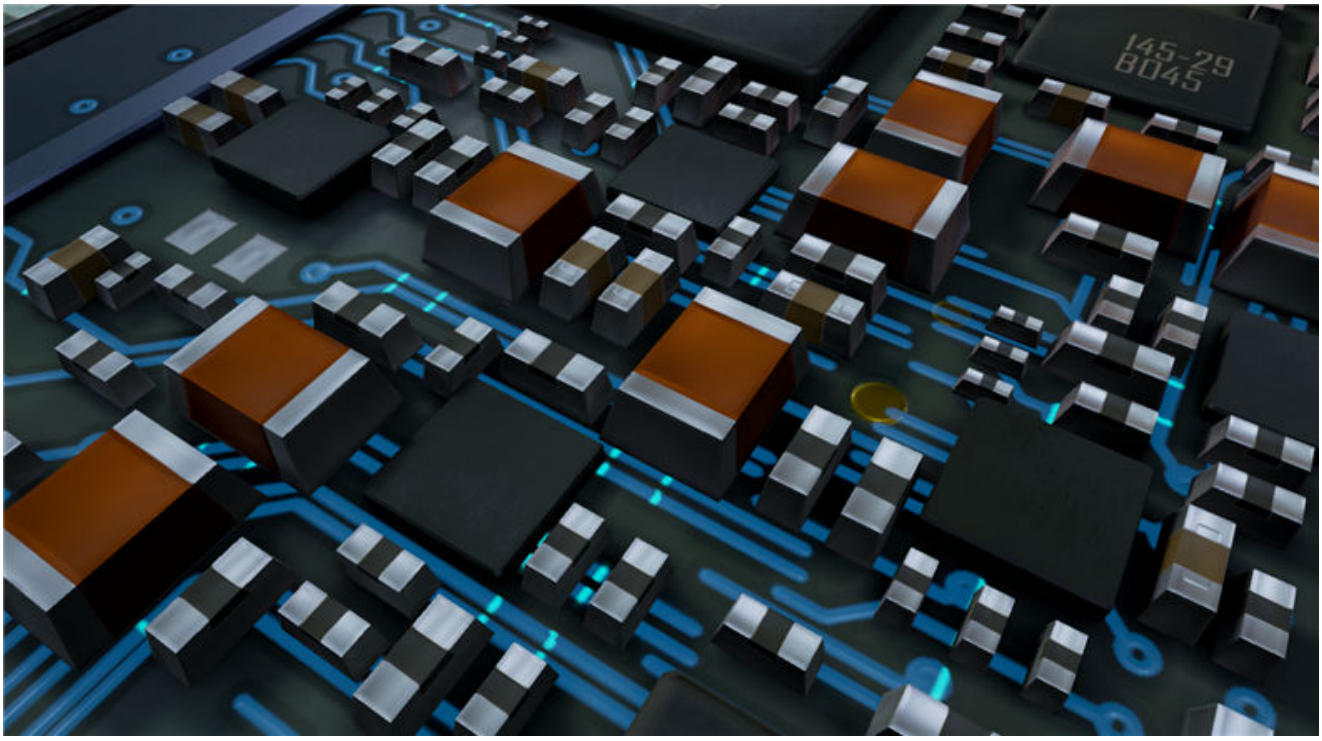

Figure 1-2 Smartphone components

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2
The optimization process

This chapter describes the optimization process.

It contains the following sections:

- *2.1 The optimization process* on page 2-18.
- *2.2 Take measurements* on page 2-19.
- *2.3 Locate the bottleneck* on page 2-20.
- *2.4 Determine the optimization* on page 2-21.
- *2.5 Apply the optimization* on page 2-22.
- *2.6 Verify the optimization* on page 2-23.
- *2.7 Repeat the optimization process* on page 2-24.
- *2.8 General optimization advice* on page 2-25.
- *2.9 Bottlenecks move between processors* on page 2-28.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1 The optimization process

Optimization is the process of taking an application and making it more efficient. For graphical applications, this typically means modifying the application to make it faster.

For example, a game with a low frame rate might appear jumpy. This gives a bad impression and can make a game difficult to play. You can use optimization to improve the frame rate of a game making it a better, smoother experience.

To optimize your code, use the optimization process. This is an iterative process that guides you through finding and removing performance problems.

The optimization process consists of the following steps:

1. Take measurements of your application with a profiler.
2. Analyze the data to locate the bottleneck.
3. Determine the relevant optimization to apply.
4. Verify that the optimization works.
5. If the performance is not acceptable return to step 1 and repeat the process.

The following is an example of the optimization process:

1. If you have a game that does not have the performance you require, you can use a profiler to take measurements.
2. Use the profiler to analyze the measurements so you can isolate and identify the source of the performance problem.
3. The problem with the game is, for example, rendering too many vertices.
4. Reduce the number of vertices in your meshes.
5. Execute the game again to ensure the optimization worked.

If you do this and the game still does not perform as expected, restart the process by profiling the application again to find out what else is causing problems.

Expect to repeat this process a number of times. Optimization is an iterative process where you might find performance problems in a number of different areas.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2 Take measurements

To start the optimization process, you first take measurements from your application. These measurements enable you to determine the problem areas.

Follow these rules when you are taking measurements to ensure that your measurements are accurate:

- Only take measurements from a hardware device with a Mali GPU. Only real hardware can provide accurate performance measurements.
- Ensure that you have VSYNC switched off when taking measurements. If it is enabled, results are likely to be inaccurate.

You can use Streamline to take readings from the Mali GPU counters and record data about the application while it is running. You can also gather performance information with other tools.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.3 Locate the bottleneck

To locate a bottleneck that reduces performance, you must analyze your measurements.

You can use tools to help you perform the analysis:

Streamline

Streamline on page 5-64 displays counters values from the Mali GPU and application processors as graphs on a timeline.

Other tools

You can also take performance measurements with other tools. The display of the measurements depends on the tool you use.

You can use the graphs and other data displays to locate a performance bottleneck. When you have located the bottleneck, you can:

- Take extra measurements to isolate the exact problem area.
- Apply one or more optimizations.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.4 Determine the optimization

The optimization to apply depends on the bottleneck. You might not find the exact cause of the bottleneck, but you can find out where it has the greatest impact.

Typically, the application is bound in one of the following areas:

Application code
Use Streamline Performance Analyzer to measure how much time is spent on the application processor. If your application takes 50ms to call SwapBuffers, your frame rate can never be higher than 20 frames per second.

Misuse of API
Graphics Analyzer enables you to check and analyze your API usage.

Use of blocking API calls
You can find blocking calls such as `glFinish()` and `glReadPixels()` with Graphics Analyzer.

Vertex processing
Check the `Vertex Load %` using the counter measurements in Streamline Performance Analyzer.

Triangle setup
You can measure this using the counter measurements in Streamline Performance Analyzer.

Fragment processing
Check the `Fragment Load %` using the counter measurements in Streamline Performance Analyzer.

Memory bandwidth
You can measure this using the counter measurements in Streamline Performance Analyzer.

For information on the Mali Midgard GPU performance counters, see

https://community.arm.com/developer/tools-software/graphics/b/blog/posts/mali-midgard-family-performance-counters.

For information on the Mali Bifrost GPU performance counters, see

https://community.arm.com/developer/tools-software/graphics/b/blog/posts/mali-bifrost-family-performance-counters.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.5 Apply the optimization

Applying the optimization might involve modifying application code and art assets.

For a list of optimizations and optimization techniques, see *Chapter 3 Optimizations and optimization techniques* on page 3-30.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.6 Verify the optimization

Verify the optimization by running the application again with the optimization applied.

To ensure successful testing, you must consider the following:

- Ensure you keep the testing environment the same.
- Only apply one optimization at a time. This ensures that you know the effect that a particular optimization has.
- Your testing sessions can take some considerable time because the device temperature can cause the GPU to reduce its clock speed. This reduces the performance of the game.
- It is best to use a relatively low GPU clock speed. This ensures that consistent performance is maintained even in prolonged gaming sessions.

Optimization might not always work as you expect. It is possible for an optimization to have little effect on application performance. This can mean the following:

- There are other bottlenecks in the application limiting performance.
- The measurements were misleading and the wrong optimization was applied. This can happen if the real bottleneck is difficult to measure.

If there is only a small difference to frame time, consider taking more measurements and analyzing them.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.7 Repeat the optimization process

An optimization process can reveal a series of different bottlenecks. You might have to go through the process several times to remove them all and get performance up to the required level.

You are likely to find new bottlenecks as you repeat the optimization process. As you optimize in one part of the system, you can find new bottlenecks in other areas.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.8 General optimization advice

General optimization advice includes experimenting with different approaches, using frames instead of frames per second for comparisons, and setting a computation budget and measuring against it.

This section contains the following subsections:

- *2.8.1 Experiment with different approaches* on page 2-25.
- *2.8.2 Use frame time instead of frames per second for comparisons* on page 2-25.
- *2.8.3 Set a computation budget and measure against it* on page 2-26.
- *2.8.4 Calculating a Fragment Shader budget* on page 2-27.

2.8.1 Experiment with different approaches

Different GPU implementations have different resources and might use various different of the Mali GPU drivers. These differences impact performance in different ways so it is important to experiment with different approaches to graphics programming and optimizations to achieve maximum performance.

Different applications can react to optimizations in very different ways. In one application a specific optimization might have a large impact on performance, whereas in another application it might have little or no impact. Profile on different devices to understand the optimizations that benefit multiple devices.

If you are optimizing, do not assume that all optimizations are always going to increase performance. A graphics pipeline consists of several components and different resources that can be the bottleneck. Optimizations that do not address the bottleneck have no effect until the current bottleneck has been resolved.

There are often trade-offs between optimizations, so experiment with different techniques to see what works best for your application.

2.8.2 Use frame time instead of frames per second for comparisons

Frames per second (FPS) is a simple and basic measurement of performance, but frame time is a better measure of optimization effectiveness.

Frame time is a linear measure, but frames per second is non-linear. Linear measurements make calculations easier.

The following figure shows frames per second plotted against frame time. This graph shows the non-linear nature of FPS measurements.

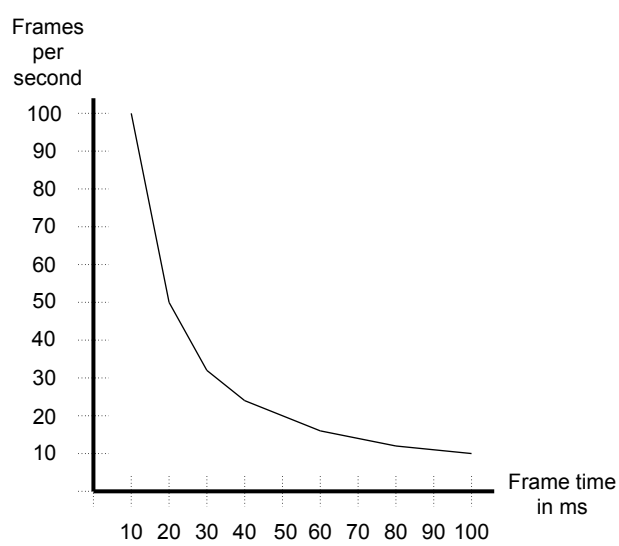

Figure 2-1 Frame time and FPS

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If you know the individual time changes corresponding to different optimizations, you can add the times together to get the total improvement.

If you are using FPS for measurements, you cannot add them together because their non-linear nature. Any attempt to add them gives an incorrect total.

The following table shows a series of comparisons between different FPS measurements A and B.

The FPS changes by a different amount for every measurement, but the frame time changes by the same amount every time.

For example, going from 100 FPS to 200 FPS involves a difference of 100 FPS or 5ms. However going from 20 FPS to 22.2 FPS is a difference of 2.2 FPS but this is also 5ms. The linear nature of frame time is easier to work with when you are measuring the impact of optimizations.

Table 2-1 Difference between frames per second and frame time

FPS change	FPS difference	Frame time difference
20 to 22.2	2.2	5ms
50 to 66.6	16.6	5ms
100 to 200	100	5ms

2.8.3 Set a computation budget and measure against it

There are maximum performance limits in processors that you cannot exceed. If you compare the computations that your application is doing against the maximum values, you can see if your application is trying to do too much.

It is useful to set a computation budget that you can measure against. The exact budget available depends on different factors such as:

- The type of GPU in your platform.
- The configuration of the GPU.
- Available memory bandwidth.
- Color depth.
- Image resolution.
- The required frame rate.

You can set a budget for:

Triangles
The maximum number of triangles per frame.

Application processor cycles
The time that is spent in application logic and in the driver, in clock cycles.

Vertex processing cycles
The average length of a Vertex shader available, in cycles.

Fragment processing cycles

The average length of a pixel shader available, in cycles.

Ensure that you take account of overdraw when calculating this. Overdraw is typically a factor of 2.5 times so divide the average length by 2.5.

Memory bandwidth
Memory bandwidth includes any data that is written to or read from memory. This includes:

- The number of bytes per pixel of texture data.
- Size of attribute data types.
- Number of vertices.
- Blitting.
- Writes to or reads from the framebuffer.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.8.4 Calculating a Fragment Shader budget

This describes how to calculate a fragment shader budget.

To calculate a fragment shader budget do the following:

1. Multiply the number of Mali GPU shader cores or fragment processors by the Mali GPU clock speed. This gives the theoretical maximum number of processing cycles per second.

   Multiply this by 0.8 to give a more realistic number of available fragment processing cycles per second. This is result A.

   Note

   Do not assume that the number of processing cycles available equals the number of processing instructions. The processors in some Mali GPUs can do many operations per cycle. The exact number varies depending on the Mali GPU you are using. To ensure good performance across a range of devices, calculate using the figures for the lowest end device you support.

2. Multiply the frame height by the frame width. This gives the number of pixels per frame.

   Multiply this by the required frame rate. This gives the number of pixels required per second.

   To take account of average overdraw, multiply this number by 2.5. This gives the number of fragments required per second. This is result B.
3. Divide the value of result A by the value of result B. This provides the approximate number of processing cycles available per pixel.

You do not have to make all your fragment shaders this long. For example, you can use longer, more complex shaders on objects closer to the camera and shorter, less complex shaders on more distant objects.

You can use the offline shader compiler to determine the number of cycles a shader requires.

For information about the offline shader compiler, see *https://developer.arm.com/graphics*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.9 Bottlenecks move between processors

This describes how bottlenecks move between processing stages of the graphics pipeline and the profile of an ideal application.

This section contains the following subsections:

- *2.9.1 How bottlenecks move between graphics pipeline processing stages* on page 2-28.
- *2.9.2 Ideal application profile* on page 2-29.

2.9.1 How bottlenecks move between graphics pipeline processing stages

The performance bottleneck in an application can move between the different processing stages as optimizations are applied. Readings from analysis tools can tell you where the bottleneck is likely to move to and if a processing stage is under-used.

Note

In Streamline, bottlenecks are directly visible in the graph display. The bottleneck is the busiest graph.

The following figure shows a bar graph of loads for different parts of a system running an application.

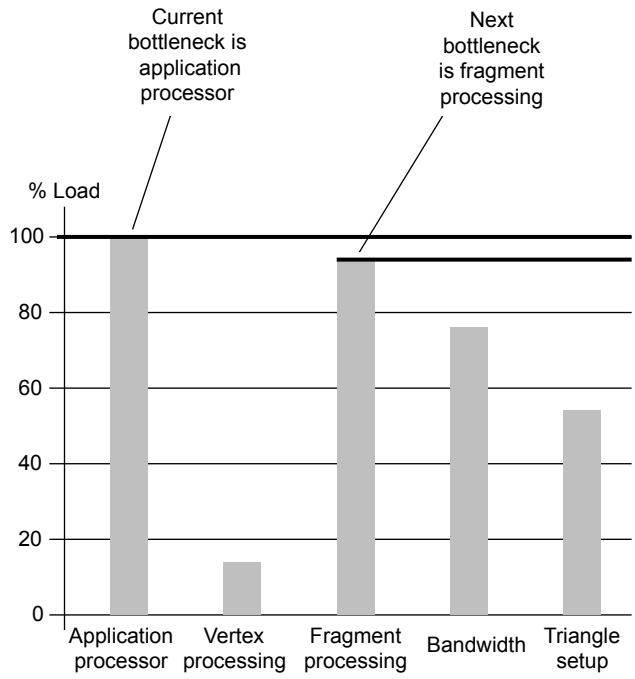

Figure 2-2 Load limitations of different system elements

Comparing the bars indicates the following:

- If you optimize the performance bottleneck, the next bottleneck is the component with second lowest bar. In this case, the bottleneck is the application processor and the next bottleneck is the fragment processing.
- The graph of the vertex processing has a much lower value than the others. This indicates that the bottleneck is not in the vertex processing. The large difference also indicates it is under-used.

  If a processor has spare processing capacity, consider if there are any processing operations that you can move to it. For example, you might be able to move operations from the application processor or fragment processing stage to the vertex processing stage.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.9.2 Ideal application profile

An ideal application is limited approximately equally by all components.

A bar graph such as shown in the following figure indicates the application is making good use of all components.

In this case, a single optimization is not likely to make a large impact on performance and you require multiple optimizations to give a higher and more stable frame rate.

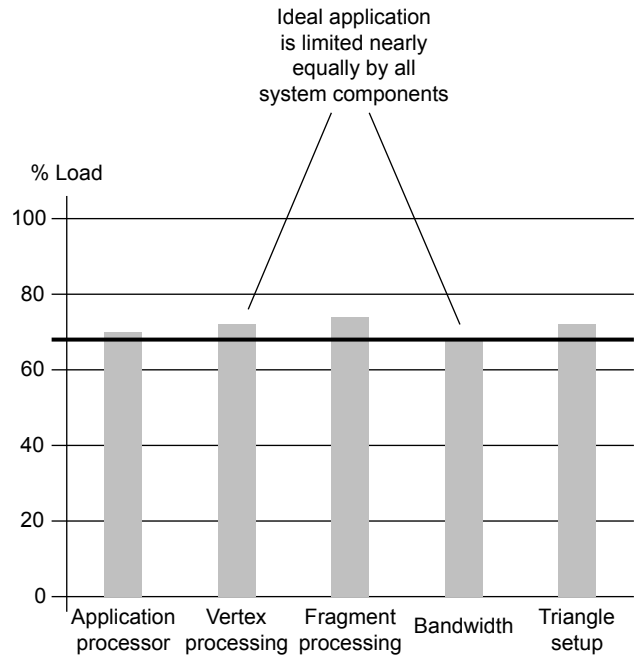

Figure 2-3 Ideal application equally limited

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3
Optimizations and optimization techniques

This chapter describes optimizations and optimization techniques.

It contains the following sections:

- *3.1 Optimization best practices* on page 3-31.
- *3.2 Adaptive Scalable Texture Compression* on page 3-35.
- *3.3 Unreal Engine best practices* on page 3-36.
- *3.4 Cubemaps* on page 3-40.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1 Optimization best practices

Developing a game for mobile requires a multi-faceted approach. Optimizing for mobile devices is more involved than optimizing for consoles or PCs because you must consider how to optimize your game across multiple performance-points.

This section contains the following subsections:

- *3.1.1 Consider device screen sizes* on page 3-31.
- *3.1.2 Consider battery life and heat dissipation* on page 3-31.
- *3.1.3 Pre-bake as much lighting as possible* on page 3-31.
- *3.1.4 Minimize the use of transparency* on page 3-31.
- *3.1.5 Test your game on real hardware* on page 3-31.
- *3.1.6 Disable high-end post processing features* on page 3-32.
- *3.1.7 Use anti-aliasing* on page 3-32.
- *3.1.8 Use mip-mapping* on page 3-33.
- *3.1.9 Android N sustained performance mode* on page 3-34.

3.1.1 Consider device screen sizes

Design assets to scale to different resolutions because mobile devices come in a large variety of resolutions.

Mobile devices come with a large variety of screen sizes, from 480 x 320, all the way up to 2560 x 1440. Therefore, it is important to ensure that you design your on-screen assets in a manner that they can scale to different resolutions efficiently.

3.1.2 Consider battery life and heat dissipation

Battery life and heat dissipation have a major impact on how you optimize your game.

Battery life and heat dissipation are major areas that require serious consideration. The more power your game requires to sustain its performance, the more power is drawn from the battery. The more power that is drawn, the hotter the phone becomes. It is therefore worth only hitting the maximum performance level in short, controlled, bursts.

3.1.3 Pre-bake as much lighting as possible

Real-time lighting is computationally expensive. Pre-bake lightmaps in advance to significantly reduce this cost.

Ensure that you pre-bake as much lighting and shadows as possible. Real-time lighting and shadows are computationally expensive and therefore have a negative impact on framerate and overall performance.

3.1.4 Minimize the use of transparency

Transparent objects are rendered on-screen by the GPU, even if their transparency is set to 100%.

High-quality transparency affects your rendering budget. If your game requires a constantly high framerate of 60 frames per second, such as first-person shooting or racing games, then double-check any scenes where the framerate drops suddenly. Scenes with multiple explosions occurring on-screen at the same time can have a dramatic impact on your rendering budget.

Note

The GPU still draws objects with 100% transparency. Therefore, adjust their visibility flag instead.

3.1.5 Test your game on real hardware

For real world performance metrics, test on the key mobile hardware you intend your game to run on.

The built-in emulators of your SDK are never as accurate as running your game on the actual hardware. While running multiple versions of an OS is much quicker in an emulator, you can only assess true

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

performance and playability on the target devices. High-end development workstations can hide any performance dips observed on real devices.

Testing the game on a device with the minimum specifications, and on multiple different devices up to the preferred specification, delivers a more accurate feed of data points to act on.

3.1.6 Disable high-end post processing features

High-end post processing features come at a computational cost that is too high for many mobile devices to handle efficiently.

Features, such as *Bloom* and *Depth-of-Field* (DOF), require substantial processing power to run, adding an extra cost that can reduce performance.

3.1.7 Use anti-aliasing

Multi-Sampling *Anti-Aliasing* (AA) is a technique for diminishing the step-like lines that you want to appear smooth on-screen.

Jaggy lines appear because the display device does not have a high enough resolution to represent the line smoothly. Anti-aliasing is a method to fool the eye into thinking the stepped-edge is smooth by placing blended pixels on either side of the hard line.

The following image shows an example with and without anti-aliasing applied:

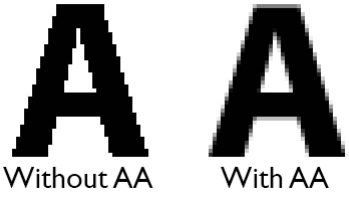

Figure 3-1 AA letter example

Multiple levels of AA are available on the Arm Mali GPU hardware. The options available with the Arm Mali GPU are:

4x MSAA
A pixel value is calculated by taking four samples around the source pixel.

16x MSAA
A pixel value is calculated by taking 16 samples around the source pixel.

The Arm Mali GPU architecture offers 4x MSAA at close to zero percent in performance penalty. With 16x MSAA, the textures are sampled at a much higher resolution yielding a much higher output quality, but as a side-effect there is a large penalty for utilizing it.

For most applications, 4x MSAA is sufficient. ARM always recommends using 4x MSAA because of the low computational cost.

The following image shows 4x and 16x AA examples.

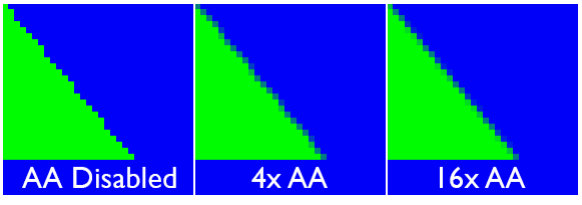

Figure 3-2 4x and 16x AA Example

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1.8 Use mip-mapping

Mip-mapping is a technique that can simultaneously improve image quality, reduce memory bandwidth usage, and potentially provide significant performance gains.

Mip-mapping has several potential benefits:

- Increases image quality.
- Increases texture cacheability.
- Reduces memory bandwidth usage.

When your application draws an object on screen, the image that is drawn can be at very different sizes, depending on the distance from the camera. It can range from filling the screen to being a small object in the distance.

If a single texture is used, the density that the texture is sampled at, or texture sampling density, is only correct when the object drawn is similar to the size of the texture. If the object size and the texture size do not match, the texture sampling density is incorrect. This produces artifacts that reduce image quality.

Mip-mapping gets around this problem by taking a high-resolution texture and scaling it to multiple smaller sizes known as mip-map levels. This requires about 33% more memory than a non mip-mapped texture. An image with several mip-map levels is shown in the following figure.

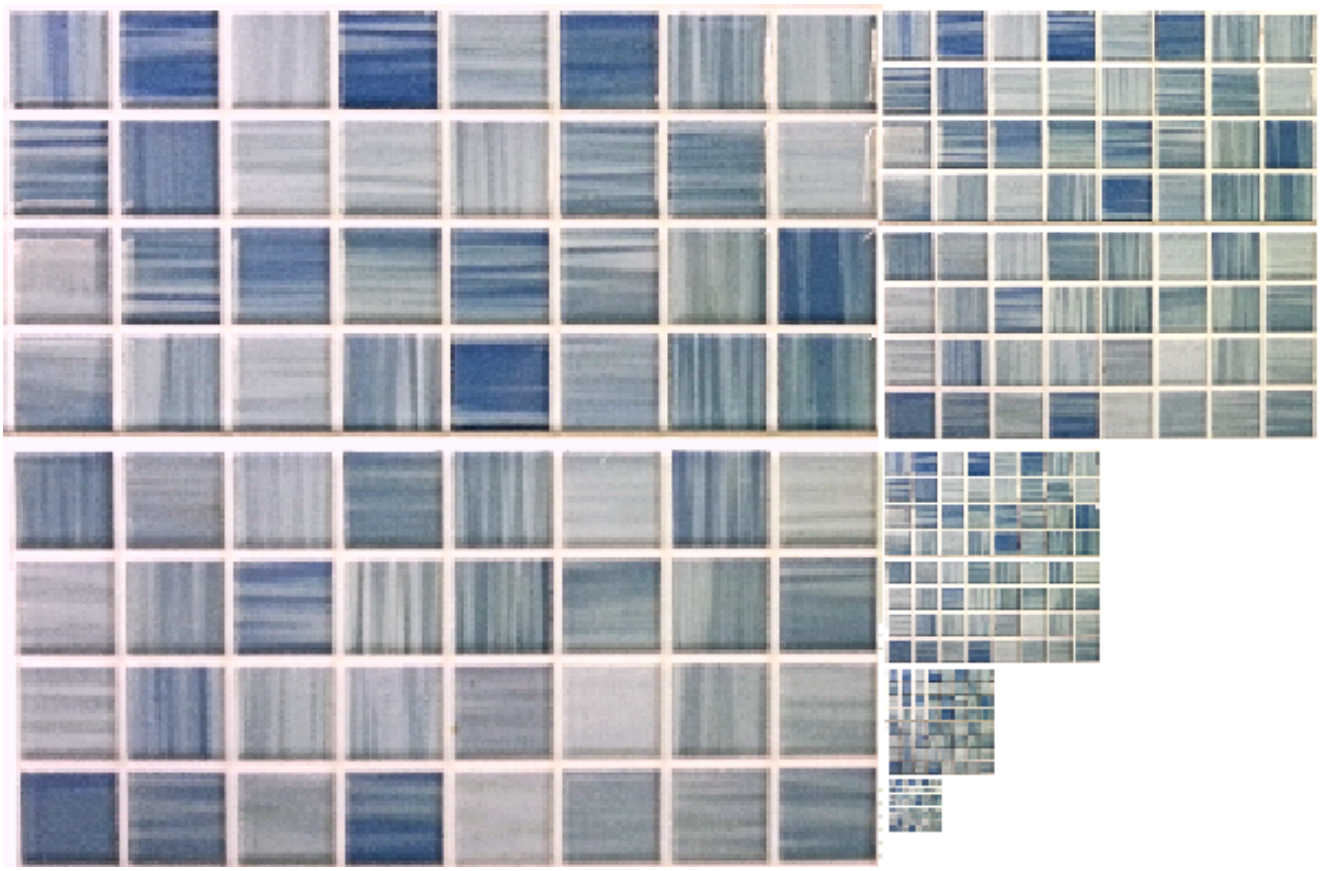

Figure 3-3 Image with mip-map levels

When an object is drawn with mip-mapping enabled, the texture with the closest size to the object is used. This means that the object always has a texture of a matching size to take samples from and the texture sampling density is therefore correct. This reduces image artifacts and produces a higher-quality image.

If the texture sampling density is correct, the texels that are sampled are close to one another in memory making the texture data more cacheable. Increased cacheability reduces memory bandwidth usage and increases performance.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Mip-mapping is an easy way to improve the performance of memory bandwidth limited applications.

Note

Some applications have shown very large performance gains with mip-mapping enabled.

Unreal Engine automatically generates mip-maps when you import a texture. You can edit the mip-map settings in the **Texture Editor**.

3.1.9 Android N sustained performance mode

You can set Android N to permanently run the processor at lower clock speeds, reducing battery drain.

Android N has a sustained performance mode API that is guaranteed to run indefinitely at a lower performance level. While a fast peak performance is attainable on mobile, you cannot sustain it indefinitely due to heat and battery drain concerns.

When profiling, use the sustained performance mode to have the lowest available budget. Optimizing for this mode enables the game to run indefinitely.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2 Adaptive Scalable Texture Compression

Adaptive Scalable Texture Compression (ASTC) produces high-quality compressed textures at low bit rates in many different image configurations.

This section contains the following subsections:

- *3.2.1 About textures* on page 3-35.
- *3.2.2 About ASTC* on page 3-35.

3.2.1 About textures

Textures describe the surface properties of objects.

In 3D graphics, the geometry describes the shape and location of objects. Shaders use textures to give flat geometry the appearance of surface detail.

Shaders use textures to color the surfaces of the object as well as add lighting, surface detail, reflectivity, and other properties. There are many different uses for textures and these have different requirements for numbers of color components and quality.

The textures used can be large, often several megabytes each. These have to be read every time the shaders run.

In a mobile device, memory bandwidth usage requires a lot of power so reducing it is important for both performance and battery life. Using texture compression saves memory bandwidth.

3.2.2 About ASTC

Compared to existing texture compression schemes, ASTC is a major step forward in terms of image quality and reduces both memory bandwidth and energy usage.

ASTC offers many features:

- ASTC is an OpenGL and OpenGL ES approved extension.
- Supports a large number of formats.
- ASTC is flexible, with bit rates from 8 bits per pixel down to less than 1 bit per pixel. This enables you to fine-tune the tradeoff of space against quality.
- Support for 1-4 color channels, together with modes for uncorrelated channels for use in mask textures and normal maps.
- Support for both *Low Dynamic Range* (LDR) and *High Dynamic Range* (HDR) images.
- Support for both 2D and 3D images.
- ASTC is not proprietary, you can use ASTC on any platform that supports it.
- You can choose any combination of features that suits your requirements.
- For the first time it is possible to use 3D textures to store, for example, discrete 3D object info.

Quality is better than existing texture compression formats for LDR images, and is comparable to other formats for HDR.

The variety of block sizes gives you many choices in how to compress a texture. For example, if an object is far from camera you can use a higher compression ratio because the visual quality is not important.

Alternatively, if the object is closer to camera the image quality is more important. You can use a lower compression ratio to produce better quality textures.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.3 Unreal Engine best practices

Unreal Engine has functions to help you optimize your game more efficiently and effectively.

Alongside the help that is already provided in this guide, you can make more performance and battery life gains by utilizing the full range of tools available to you in Unreal Engine.

This section contains the following subsections:

- *3.3.1 Automatic level of detail generation* on page 3-36.
- *3.3.2 Reduce drawcalls by merging actors* on page 3-36.
- *3.3.3 Unreal quality settings* on page 3-37.

3.3.1 Automatic level of detail generation

Unreal Engine enables you to automatically generate several *Level Of Details* (LODs) from your static meshes.

Using LODs reduces the polygon count in your graphics, because a different LOD is rendered depending on the distance of your mesh from the camera view point.

Rendering large meshes consumes a lot of memory and battery power, so it is important to render the images at the most efficient LOD level. Render to the lowest LOD that does not show any visual artifacts or compromise the visual quality. By rendering content to this optimal point rather than to the highest resolution, your game can reduce image rendering time and device energy usage.

The following image shows the Unreal Engine LOD settings.

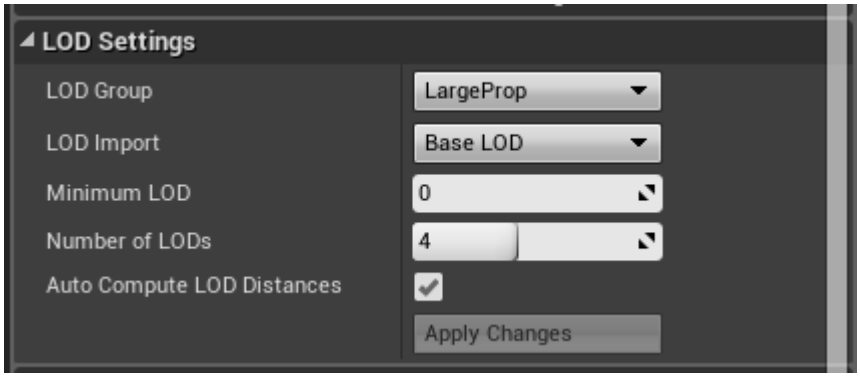

Figure 3-4 Unreal Engine LOD Settings

To make Unreal Engine automatically calculate the screen size to use for each LOD level, select **Auto Compute LOD Distances**.

3.3.2 Reduce drawcalls by merging actors

You can reduce the number of drawcalls that are made by using the *Merge Actors* tool in the *Developer Tools* menu.

When you merge actors that use various texture sizes, the packing algorithm evenly divides up your texture space, based on the number of textures. It then packs them into the set Texture Size.

The following image shows the *Merge Actors* tool:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

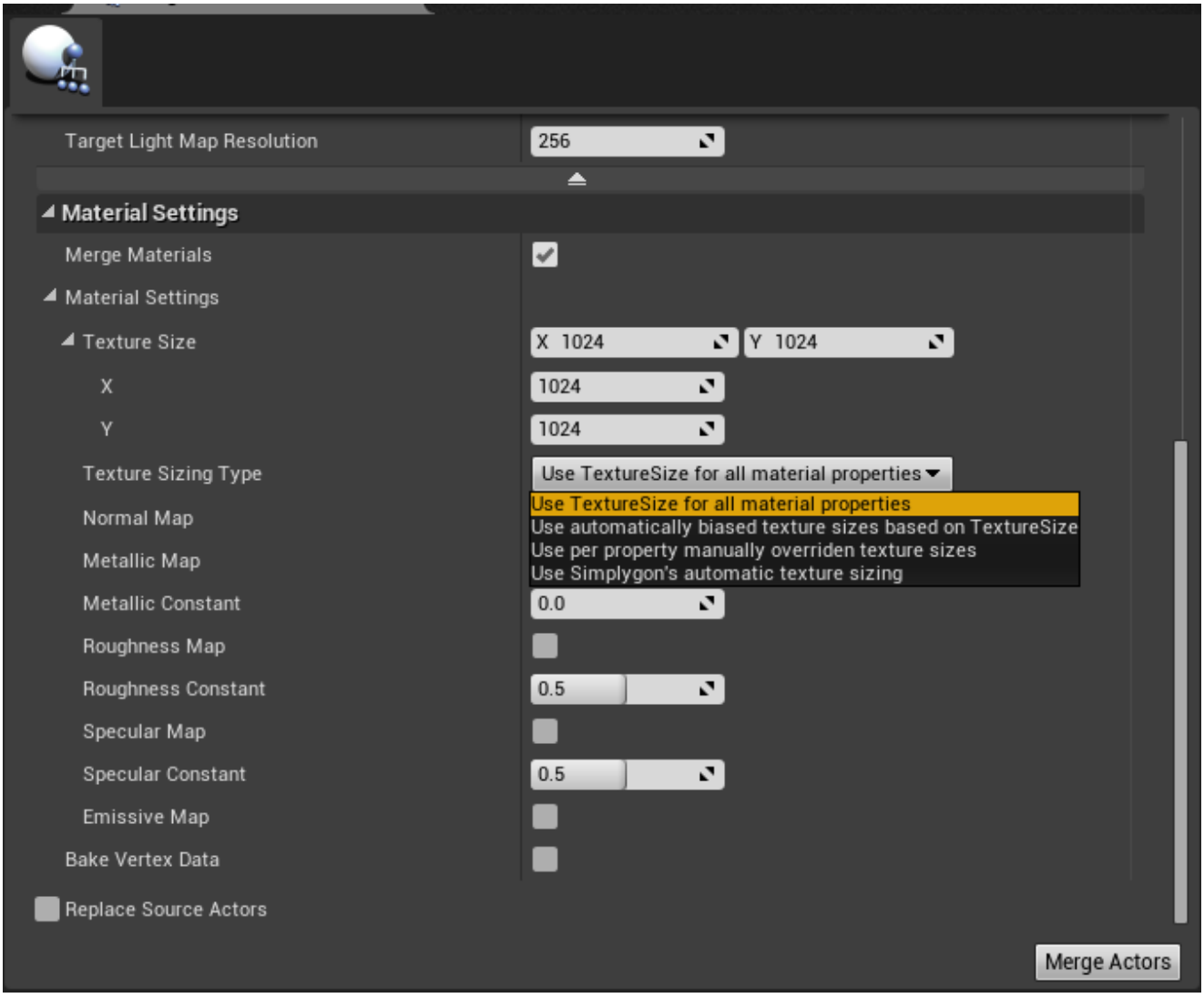

Figure 3-5 Merging actors

3.3.3 Unreal quality settings

There are a number of quality settings in Unreal Engine that enable you to optimize the quality of your game for your target device.

Mobile material quality settings

When building content that runs on low and high end mobile devices, you might run into issues with some features working on one set of devices, but not on another.

You also might encounter issues with your artwork looking correct on one device but completely different on another. While there are many ways to address problems like this, many of these ways are time and resource intensive and error prone. To address these issues, Unreal Engine has the material quality level system. This system enables you to build a single material that you can then use on a wide range of different devices, giving you full control over which devices use which features.

You can view what the different Material Quality level settings look like, inside of the Unreal Engine Editor. Going to the Main Toolbar and then go to **Settings** > **Material Quality Level** and select the quality level you want to preview.

Material quality level switch

You can set the material quality level on the console and in device profiles.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

You can set the material quality level in the following ways inside the Unreal Engine editor:

On the Console

In the editor, press the backtick key to open up the Unreal Engine console. Enter the following text followed by a number for the quality level:

```
r.MaterialQualityLevel
```

The quality levels are described in the following table.

Table 3-1 Material quality levels

Level	Quality
0	Low
1	High
2	Medium

In Device Profiles

You can adjust the material quality in **Device Profiles**:

From the main toolbar, go to **Windows** > **Developer Tools** and then click **Device Profiles**.

In the **Device Profiles** find the **Android_Low** profile and click on the first wrench icon to open the profile options.

In the **Console Variables** section under the **Rendering** option, click on the plus icon and in the input window that pops up, input `R.MaterialQualityLevel` and then press the Enter key to add the command to the list.

When the new entry has been added, change the value from the default value 1 to 0 so that when the project is viewed on lower end Android devices it uses the fastest possible material settings.

You can preview how your project looks when running on different hardware from the Unreal Engine editor. Go to on the Main Toolbar and select **Settings** > **Preview Rendering Level**. Select the preview level you want to use.

Quality switch material expression

Quality switch material expression enables you to define different levels of complexity within a single material.

For example, if you have a complex math operation or heavy texture reads, that work on higher end mobile devices but not on lower end mobile devices. You can use this node to specify simplified versions to display on lower end mobile devices that do not have the compute power to display higher end graphics.

To use the quality switch in your materials, you must:

1. Add the **Quality Switch Material Expression** node to the **Material Graph**.
2. Plug the output into any of the inputs on the **Main Material** node you want to change when the quality level is changed.

Mobile material options

There are a number of mobile material options available.

Open any **Material** and click on the **Main Material Node**, look in the **Details** Panel under the **Mobile** section, there are the following properties:

Fully Rough

Forces a material to be completely rough, saving several instructions and a texture sampler.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Use Lightmap Directionality
: Uses the lightmap directionality and per pixel normals. If disabled, the lightmaps are flatter but cheaper.

These properties can be enabled to help reduce the rendering cost of a material when viewed on less powerful mobile devices by completely eliminating the rendering path for those features. However, this is an all or nothing option because you either have these options enabled or disabled for this material on all devices.

Mobile material rendering overrides

You can override the rendering option a platform uses by adjusting the ES2 quality settings.

To locate the ES2 quality setting for both Android and iOS, go to **Project Settings** > **Platforms** and then select either Android ES2 Quality or iOS ES2 quality settings. To use the overrides, click on the **Enable Quality Override** option and then click the option you want to override.

When you have selected all of the options you want, press the **Update preview shaders** button to re-compile all materials to work with the options you have specified.

Mobile rendering options

There are a number of options that control how lighting is handled in a mobile project.

Mobile rendering options are located under the **Mobile** section of the **Settings** menu for your project.

To access your project settings, go to the Main Toolbar and then select **Edit** > **Project Settings**.

Locate the **Engine** section and then click on the **Rendering** category and look for **Mobile**.

Under the Mobile section there are a number of different options that you can enable or disable:

Mobile HDR
: If enabled, mobile renders are in full HDR. Disable this setting for games that do not require post-processing, or for better performance on slow devices.

Max dynamic point lights
: This sets the number of dynamic point lights supported on mobile devices. Setting this to 0 for games that do not require dynamic point lights reduces the number of shaders generated. Changing this setting requires restarting the editor.

Use shared dynamic point light shaders
: If enabled, the engine uses the same shader for the dynamic point lights hitting a surface. This is slightly slower but reduces the number of shaders generated. Changing this setting requires restarting the editor.

Enabled combined static and CSM shadowing
: This enables primitives to receive both static and CSM shadows from a stationary light. Disabling this frees a mobile texture sampler.

For more information, see *https://docs.unrealengine.com/latest/INT/Platforms/Mobile/Performance/index.html*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.4 Cubemaps

It is fundamental to have high-quality reflections while carefully balancing the available runtime resources. Cubemaps are a technique that let you do this.

Reflections are an important part of the visual world and therefore provide better immersion and credibility for your environment.

Generating reflections in real time is very computationally expensive. Precomputed reflections enable you to reduce the load at runtime, while their quality can be controlled by adjusting the resolution.

A cubemap is a 360-degree snapshot of the world that is taken from a specific spot, looking along each orthogonal axis. If you generate a cubemap looking from a reflective object, you can get the reflected color just by sampling the cubemap along the appropriate direction.

Unreal Engine has an existing native implementation of precomputed reflections with sphere reflection captures, however cubemaps produce higher resolution, and therefore better results.

This section contains the following subsections:

- *3.4.1 History of reflection implementations* on page 3-40.
- *3.4.2 Generating correct reflections with a local cubemap* on page 3-43.
- *3.4.3 Implementing reflections in Unreal Engine* on page 3-45.

3.4.1 History of reflection implementations

Graphics developers have always tried to find computationally cheap methods to implement reflections.

One of the first solutions is spherical mapping. This technique simulates reflections or lighting on objects without using expensive ray-tracing or lighting calculations.

The following image shows a scene with no mirror:

Figure 3-6 Starting scene, without a mirror

The following figure shows an environment map on a sphere:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

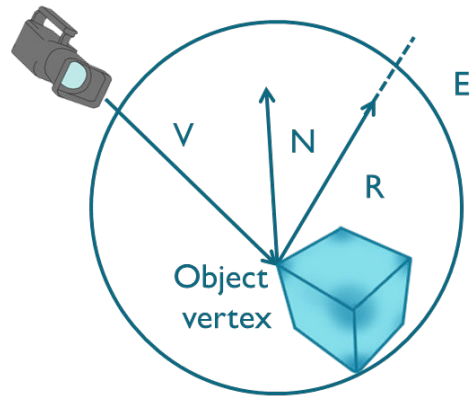

Figure 3-7 Environment map on a sphere

The following figure shows the equation for mapping a spherical surface into two dimensions:

The spherical surface is mapped into 2D:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

Figure 3-8 Spherical surface 2D mapping equation

This approach has several disadvantages, but the main problem is the distortions that occur when mapping a picture onto a sphere. In 1999, it became possible to use cubemaps with hardware acceleration. Cubemaps solved the problems of image distortions, viewpoint dependency and computational inefficiency related to spherical mapping.

The following figure shows an unfolded cube:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

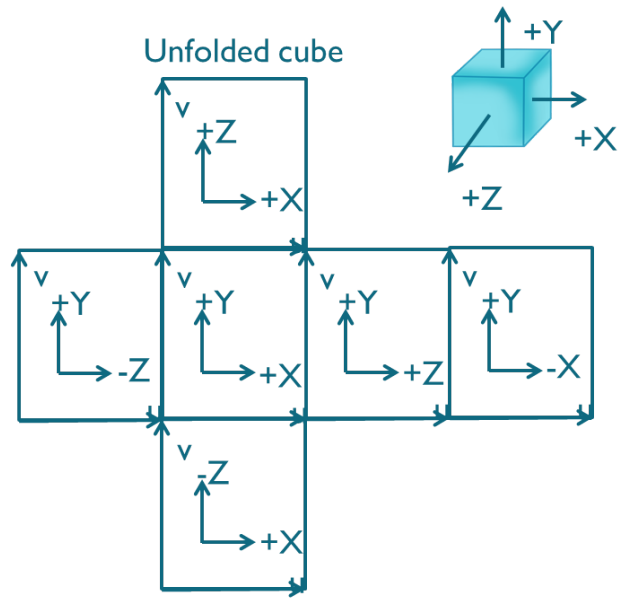

Figure 3-9 Unfolded cube

Cubemapping uses the six faces of a cube as the map shape. The environment is projected onto each side of a cube and stored as six square textures, or unfolded into six regions of a single texture. The cubemap is generated by rendering the scene from a given position with six different camera orientations with a 90 degree view frustum representing each a cube face. Source images are sampled directly. No distortion is introduced by resampling into an intermediate environment map.

The following figure shows infinite reflections:

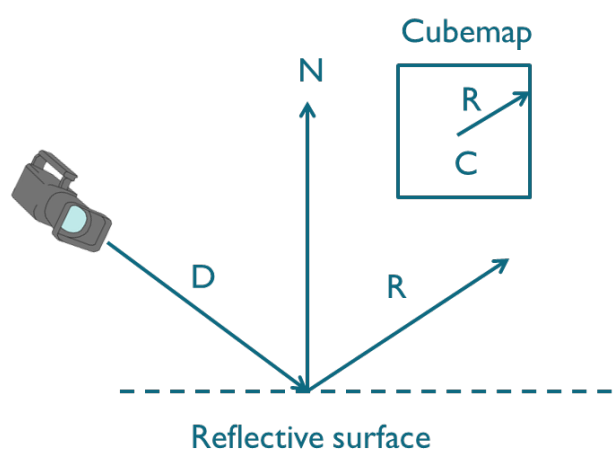

Figure 3-10 Infinite reflections

This approach can only reproduce reflections correctly from a distant environment where the cubemap position is not relevant. This simple and effective technique is mainly used in outdoor lighting, for example, to add reflections of the sky.

The following figure shows incorrect reflections:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 3-11 Incorrect reflections

If you use this technique in a local environment it produces inaccurate reflections. The reason why the reflections are incorrect is that there is no binding to the local geometry. For example, if you walk across a reflective floor looking at it from the same angle you always see the same reflection. The reflected vector is always the same and the expression always produces the same result. This is because the direction of the view vector does not change. In the real world reflections depend on both viewing angle and viewing position.

3.4.2 Generating correct reflections with a local cubemap

A solution to this problem involves binding to the local geometry in the procedure to calculate the reflection.

This solution is described in: *GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics* (1st edition, 2004).

The following figure shows a local correction using a bounding sphere:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

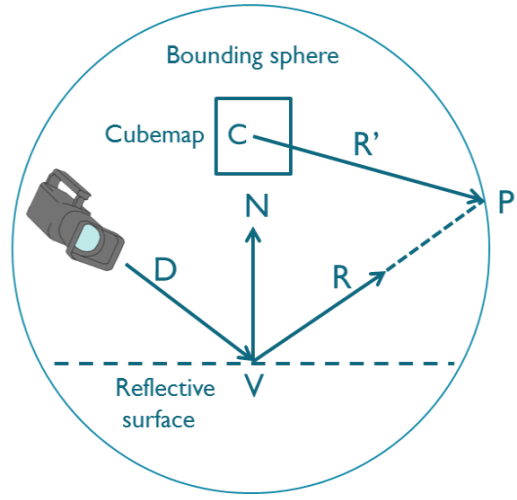

Figure 3-12 Local correction using a bounding sphere

A bounding sphere is used as a proxy volume that delimits the scene to be reflected. Instead of using the reflected vector `R` to fetch the texture from the cubemap a new vector `R'` is used. To build this new vector, you find the intersection point `P` in the bounding sphere of the ray from the local point `V` in the direction of the reflected vector `R`. Create a new vector `R'` from the center of the cubemap `C`, where the cubemap was generated, to the intersection point `P`. Use this vector to fetch the texture from the cubemap.

This approach produces good results in the surfaces of objects with a near spherical shape but reflections in plane reflective surfaces are deformed. Another drawback of this method is that the algorithm to calculate the intersection point with the bounding sphere solves a second degree equation and this is relatively complex.

In 2010, a developer proposed a better solution in a forum at *http://www.gamedev.net*. This approach replaces the previous bounding sphere by a box and solves the deformations and complexity problems of the previous method. For more information see: *http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/?&p=4637262*.

The following figure shows a local correction using a bounding box:

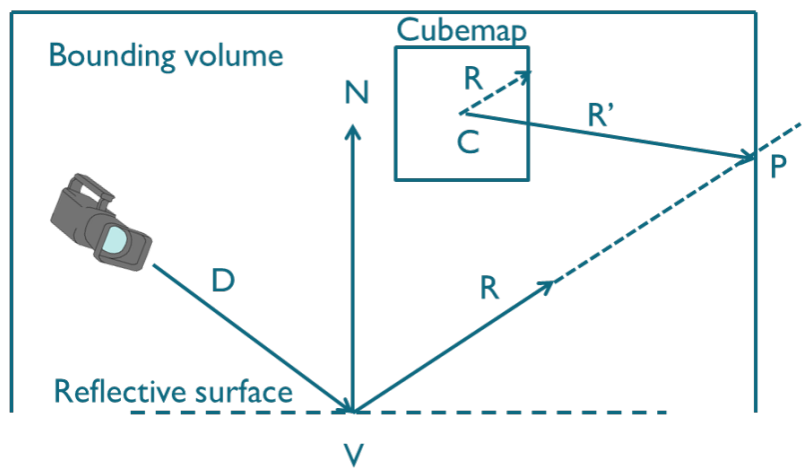

Figure 3-13 Local correction using a bounding box

A more recent work in 2012 by Sebastien Lagarde uses this new approach to simulate more complex ambient specular lighting using several cubemaps and uses an algorithm to evaluate the contribution of each cubemap and efficiently blend on the GPU. See *http://seblagarde.wordpress.com*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-2 Differences between infinite and local cubemaps

Infinite cubemaps	Local cubemaps
• Mainly used outdoors to represent the lighting from a distant environment. • Cubemap position is not relevant.	• Represents the lighting from a finite local environment. • Cubemap position is relevant. • The lighting from these cubemaps is right only at the location where the cubemap was created. • Local correction must be applied to adapt the intrinsic infinite nature of cubemaps to local environment.

The following figure shows the scene with correct reflections generated with local cubemaps.

Figure 3-14 Correct reflections

3.4.3 Implementing reflections in Unreal Engine

Reflections are an important element in how we visualize the real world and developers therefore use them extensively in their games.

This section describes how to render reflections based on local cubemaps and the advantages of this technique for mobile games where resources must be carefully balanced at runtime.

Generating a cubemap

This describes how to generate a cubemap in Unreal Engine.

Use these steps to implement reflections, beginning with generating the cubemap that you use in your reflective material.

Procedure

1. Look for a **Scene Capture Cube** in the **Modes** panel and place it into the scene. This actor is the origin of your cubemap.
   Example: The following image shows a scene capture cube:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

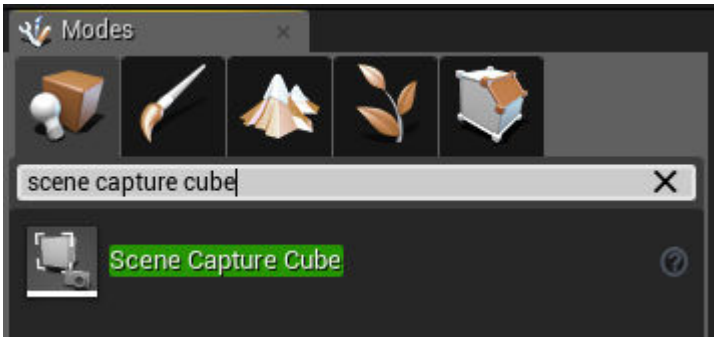

Figure 3-15 Scene capture cube

2. Drag the actor to the required point of your scene, where it appears on-screen as a camera. Make sure you do not rotate it, this ensures the cubemap is generated with the default orientation and sampling it is easier.
   Example: The following image shows dragging the scene capture cube into your scene:

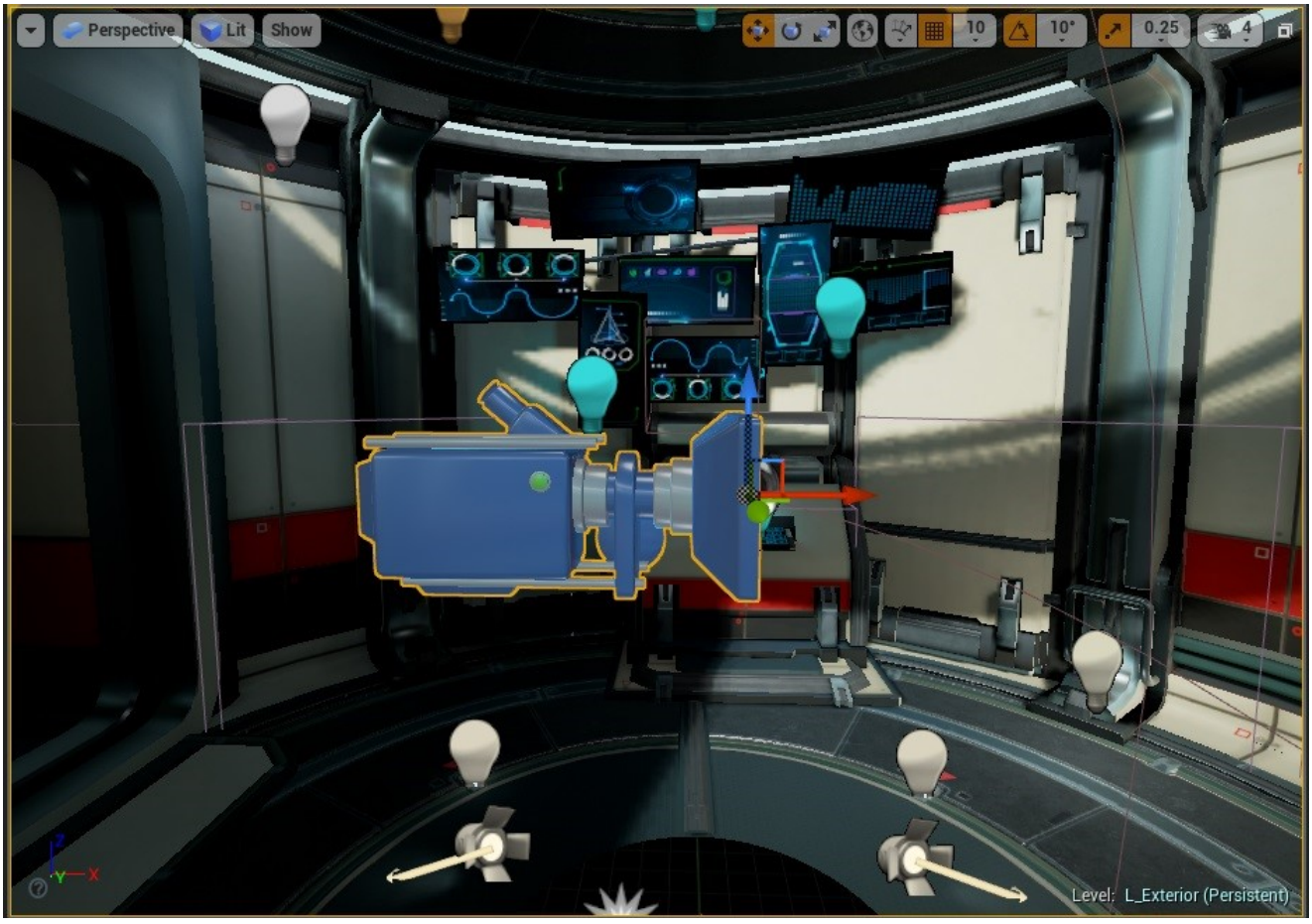

Figure 3-16 Drag the scene capture cube into your scene

3. Connect the **Scene Capture Cube** to a **Cube Render Target**. To do this, look for the **Texture Target** parameter in the **Details** panel of the **Scene Capture Cube**, click it and then select **Cube Render Target** under Create New Asset.
   Example: The following image shows connecting the scene capture cube to a cube render target:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

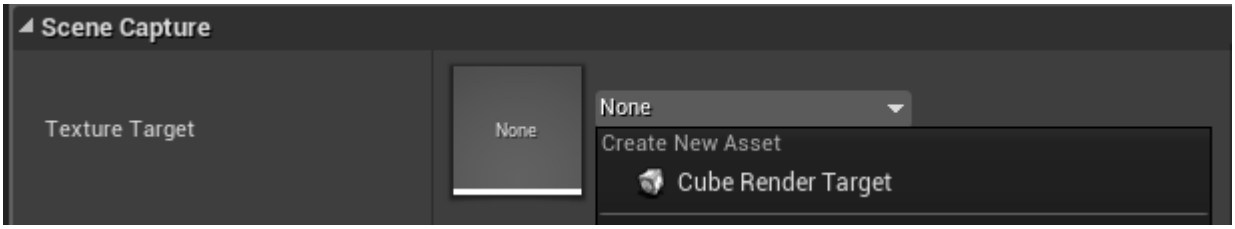

Figure 3-17 Connect to cube render target

4. When completed, the scene capture happens immediately. If that is not the case, try playing the game in the editor. Double-click on the **Cube Render Target** you created and look at the following parameters:
   - **Size X** corresponds to the cubemap resolution. The default 256 is too low in this example, so 1024 is a suitable compromise between quality and performances. In general, the resolution of the cubemap must be proportional to the size of the reflective surface, to ensure high-quality reflections.
   - **HDR** enables the storage of HDR values. Untick this, because Mobile HDR can be expensive and it is not available in VR.

   Example: The following image shows texture render target cube values:

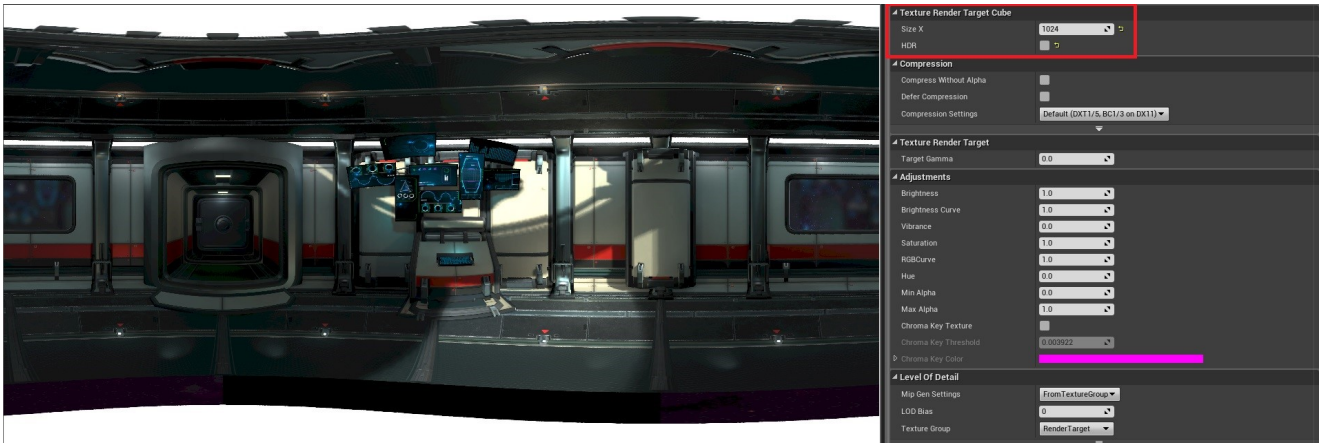

Figure 3-18 Texture render target cube values

5. There is an actor that renders a cubemap each frame. You require a static cubemap. Right-click on the **Cube Render Target** in the **Content Browser** and select **Create Static Texture**. You require the world space location of the **Scene Capture Cube** in your material. Get this from the **Transform** section in the **Details** panel for the actor. Click **Location** and select **World** from the drop-down menu, so that the location vector is in world space. Apart from these coordinates, you do not need the **Scene Capture Cube** and the **Cube Render Target** anymore.
   Example: The following image shows creating a static texture:

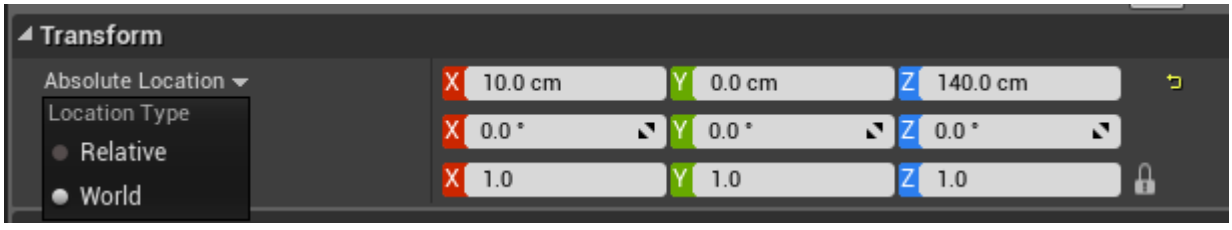

Figure 3-19 Creating a static Texture

Setting up the material function

You must set up the material function.

Use the material functions in Unreal Engine to wrap your code in a reusable block, and create a material function called `Local Correction`.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

You require these input nodes: `BBoxOrigin`, `BBoxMax`, and `BBoxMin`. These are all of type `Vector3`, and they represent the position of the origin, and the extremes of the bounding box.

The extremes are in the form (minX, minY, minZ) and (maxX, maxY, maxZ). You also require the **Absolute World Position** and the **Reflection Vector**, that are available as nodes in the **Material Editor**.

The first part of the algorithm is the ray-box intersection. Considering a ray in its parametric form, that is, you are looking for the value of the `t` parameter corresponding to the intersection point.

The following image shows how to implement ray-box intersection in Unreal Engine:

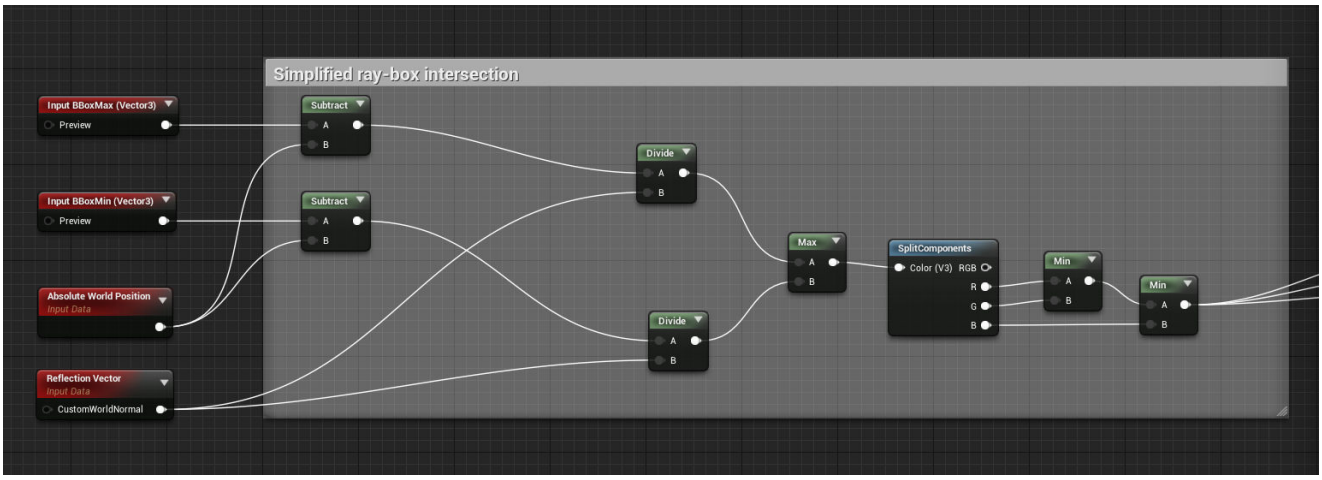

Figure 3-20 Simplified ray box intersection

You now have the value of `t` you were looking for, so you can get the intersection point with the bounding box by implementing `P = P0+Rt`. Get the locally corrected vector by subtracting the intersection point from the origin of the cubemap. You use this vector in your material to sample the cubemap.

The following image shows the implementation for the second part of the algorithm:

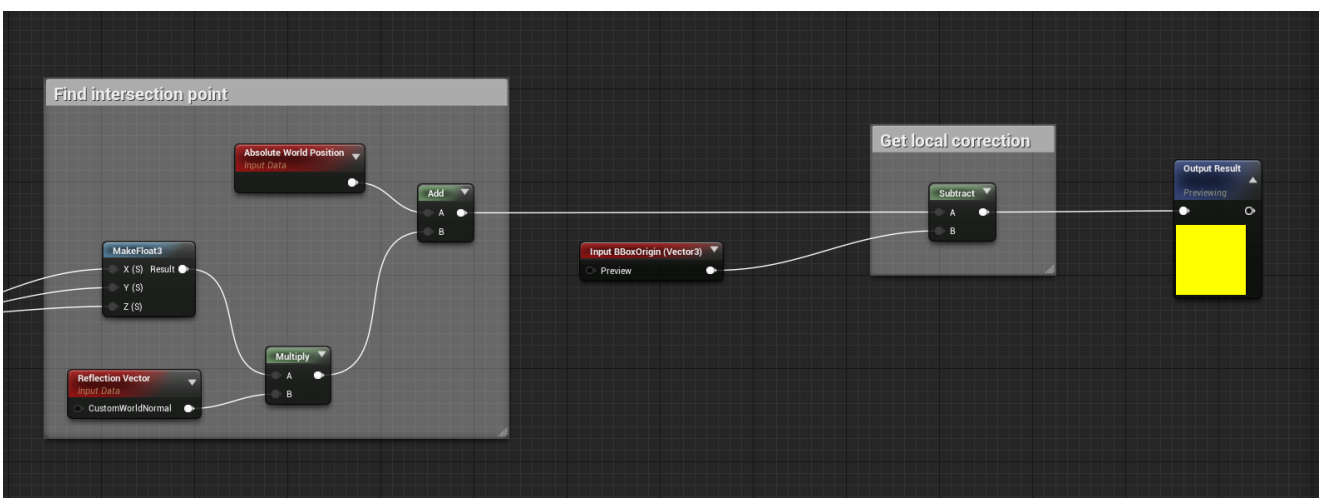

Figure 3-21 Find intersection point

Click the background of your **Material Editor** to bring up the **Material Function** properties. Tick the **Expose to Library** checkbox, so that you can easily add the function to any material as a standard **Material Editor** node.

Setting up the material

You must now set up a material that uses the material function.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

You typically have a pre-existing material that you want to add reflections to. You can add this reflection by attaching the sampled cubemap color to the **Emissive Color** node.

Place the **Local Correction** node that you created in your material editor, and provide its inputs. The coordinates include the origin and the extremes of the bounding box, in world space coordinates.

The output of the local correction node is attached to a **Texture Sample Parameter Cube** node. Set its texture input to the cubemap you generated previously.

The following image shows a cubemap blueprint:

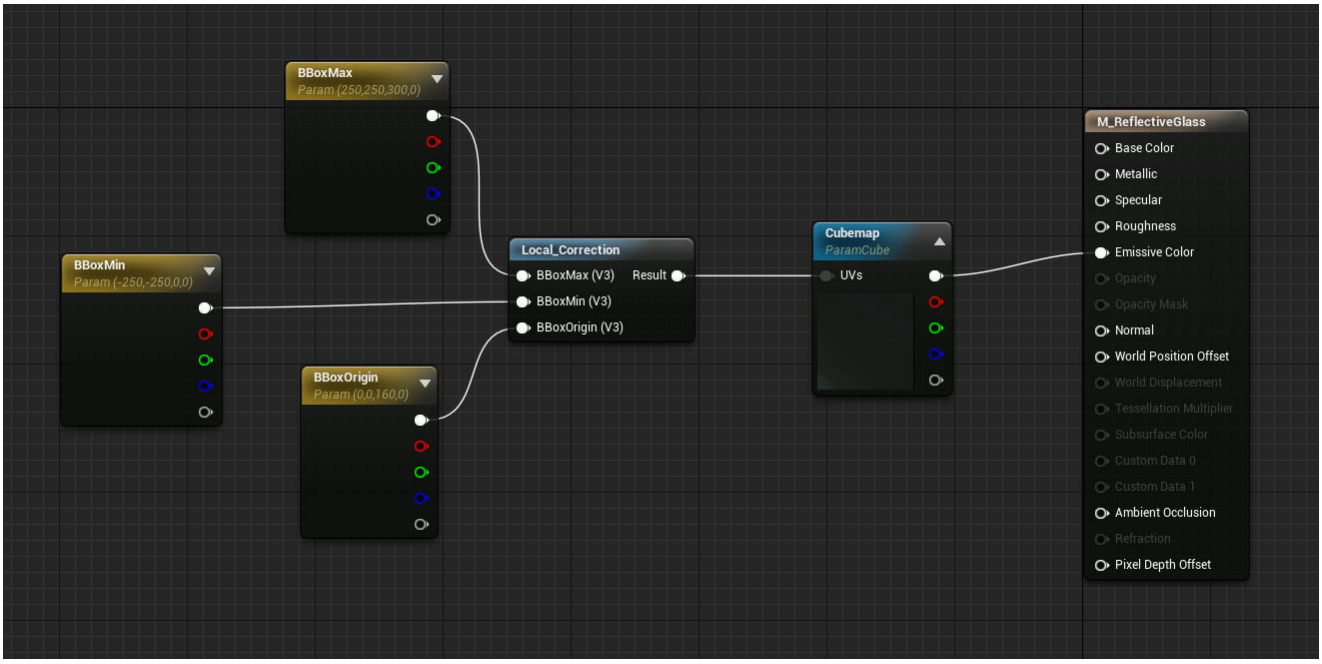

Figure 3-22 Cubemap blueprint

When that is done, your reflections are in place. You can then do any sort of processing with them, depending on your specific requirements. A very basic control you can implement is a Linear Interpolation or Lerp node, interpolating between (0, 0, 0) and the sampled texture color. This enables you to tone down the reflection intensity, based on the Alpha input of the Lerp.

Further improvements

The opportunity for offline processing of the cubemap is important in mobile development, where post processing is not typically an option because it is too demanding.

You can achieve the required effects by exporting the cubemap, editing it in an external program, and reimporting it into Unreal Engine. As an example, blurring the cubemap is a useful tool for creating softer reflections.

When implementing localized reflections, there can be a serious issue with flickering. Reflections might appear unstable, as if a different point in the cubemap was sampled at each frame.

There are several approaches that can help to reduce this issue:

- Change the mip gen settings to have more blurred reflections.
- Experimenting with the filtering options in the texture editor.
- Simplifying the normal map of the object. Do this if the issue is more prominent near the edges.
- Using full precision computations can fix this problem but it increases the execution time of the shaders but you might want to accept this tradeoff between quality and performance. To do this select **Use Full Precision** under the **Mobile** section in the **Material Editor**.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If you want to reuse the same reflective material with several different cubemaps, you can create material instances based on that material. This is more efficient because they can share most of the compiled code.

Material instances can change the parameters of the original material. While you have already specified the cubemap as a parameter, you must also provide the inputs to the `Local Correction` function as parameters.

Conclusions

Reflections using static local cubemaps are a powerful tool to give a better sense of realism to your game, at an affordable cost in terms of performance.

Their main advantage is that they remove runtime workload by moving it to an offline rendering process.

Using a manual approach is more complex than using the native reflection environment feature, but it also gives you more control over the generation and processing of the cubemap.

On certain platforms, the native feature is only supported at a low resolution. In this case, using a manual implementation is the only way to achieve high-quality static reflections.

A limitation of precomputed reflections is that they are able to reproduce correct reflections only in a static environment, because moving objects require their reflections to be computed at runtime.

However, considering the fluidity requirements for a good mobile virtual reality experience and the performance budget you are working with, high-quality static reflections are the right choice to convey that sense of immersion that mobile gaming is all about.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 4
Virtual reality optimization techniques

This chapter describes optimization techniques for *Virtual Reality* (VR) applications.

It contains the following sections:

- *4.1 Virtual reality development best practices* on page 4-52.
- *4.2 Monoscopic far field rendering for mobile VR* on page 4-56.
- *4.3 Understanding Multiview* on page 4-58.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1 Virtual reality development best practices

To generate an immersive mobile VR experience, spend as much time optimizing your game as possible.

When developing content for VR projects, you must ensure that the player has the best VR experience possible. This section describes the steps you can take.

This section contains the following subsections:

- *4.1.1 Do not use mobile high dynamic range* on page 4-52.
- *4.1.2 Post processing selection* on page 4-52.
- *4.1.3 Stereo layers* on page 4-52.
- *4.1.4 Materials* on page 4-53.
- *4.1.5 Material shading models* on page 4-54.
- *4.1.6 Material blend modes* on page 4-54.
- *4.1.7 Mobile specific material settings* on page 4-54.
- *4.1.8 Customized UVs* on page 4-55.
- *4.1.9 Mesh considerations* on page 4-55.

4.1.1 Do not use mobile high dynamic range

Mobile *High Dynamic Range* (HDR) does not work correctly with VR in Unreal Engine 4.15.

Mobile VR enables you to use advanced features, but do not enable HDR. Mobile HDR does not work correctly with a VR Headset. Enabling this feature causes your project to render incorrectly when placed into the VR Headset.

4.1.2 Post processing selection

Post processing encompasses any function that improves or alters the quality of an image after it has been added, but before it is displayed on-screen.

There are many post processing features that do not perform well when they are used with VR. This is because of the very high rendering requirements of mobile VR headsets.

You can use the following post processing operation without problems:

Exposure Bias increases or decreases the scenes brightness.

4.1.3 Stereo layers

Stereo layers is a technique that enables you to render one or two textures at a higher quality level than other layers.

These textures are not being rendered in-engine, the compositor renders them. This provides two main advantages:

1. The layer is raytraced in the compositor through the lenses, so only one sampling pass of the texture is required. One-pass sampling reduces the amount of blurring that occurs.
2. If you look at the render targets, they are lower than what you would need for 1:1-pixel density – in comparison to the screen on the mobile device. When calculated in the compositor, the layer runs one time per-pixel on the device, resulting in higher quality.

Unreal Engine supports the following types of layers:

Quads
: Used to render objects in quads.

Cylinders
: A wrap-around layer that is suited to user interfaces.

Cubemaps
: Allows for higher-quality detail on far-away items.

The following image shows Stereo layer settings in Unreal Engine.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 4-1 Stereo layer settings in Unreal Engine

4.1.4 Materials

When you are making materials for mobile VR, there are a few things you must consider, to ensure that your material works with the mobile VR Headset.

Creating and interacting with materials for mobile VR projects is similar to creating and interacting with materials for other projects. You use the same **Material Editor** to create materials and you can use many of the same material nodes.

The following table shows material parameters.

Table 4-1 Materials table

Material Input Name	Description
Base Color	Base Color defines the overall color of the Material.
Roughness	Roughness defines how rough or smooth the Material is.
Metallic	Metallic defines how metal-like the Material is.
Specular	Specular defines the intensity of the reflections.
Normal	Normal helps to add detail that would otherwise be too costly to model in.
Emissive	Emissive helps to make parts of Materials glow like they are emitting light.
Opacity	Opacity helps to define how transparent something is.
Opacity Mask	Opacity Mask helps to define if a certain pixel is transparent or not.

Note

Translucent and masked materials can be expensive to render, especially on hardware like smartphones. Whenever possible, use opaque materials because they are cheaper to render.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1.5 Material shading models

Certain restrictions apply if you are using material shader models in mobile VR.

The following material shading models are available for use in mobile VR projects.

Table 4-2 Material shader models available for use

Shading model name	Description
Default Lit	This shading model uses both direct and indirect lighting, and specularity for reflections. This is the default shading model.
Unlit	The unlit shading model only outputs emissive for color, making it useful for special effects such as fire or illuminating objects.

4.1.6 Material blend modes

Material blend modes describe how the output of the current material blends over the background.

The following material blend modes are available for mobile VR content:

Table 4-3 Blend modes for mobile VR

Blend Mode	Description
Opaque	For a solid object opaque is the cheapest material, but it has no support for translucency. That means that zero creates overdraw, and rendering one pixel of the material always takes the same time depending on the complexity of the node graph.
Masked	Use Masked if you require some pixels that are not drawn on your material. The material only creates overdraw for the area that is masked out. But the material is 100% opaque or 100% see through. The bigger the area that is masked out, the more expensive overdraw there is.
Translucent	Translucent is the most expensive blend mode. It can look like a masked and opaque material, but translucent has an extra effect on performance regardless of it being transparent or not.
Additive	This is a cheapest alternative to Translucent because it does not sort the pixels. This adds its pixels to the scene so it can only brighten. Black pixels in the material have no effect on the scene.
Modulated	Is the second cheapest alternative to Translucent. This multiplies its pixels with the scene so it can only darken the scene.

Note

If you use the Modulate blend mode, disable **Separate Translucency** to ensure the Material compile correctly.

4.1.7 Mobile specific material settings

Mobile specific material settings help reduce the cost of rendering in Mobile VR.

There are settings that help you reduce material rendering costs on mobile devices. These are located on the **Main Material Node**, under the **Details** panel, in a section called **Mobile**.

The following settings are available:

Fully Rough
When enabled, saves numerous shader ALU instructions and one sampler. This effectively disables specular while maintaining overall reflective energy.

Use Lightmap Directionality
When this is disabled, the lighting from light maps is flat, but it is also cheaper.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1.8 Customized UVs

Customized UVs enable you to perform some shader operations in vertex shaders instead of pixel shaders, reducing dependent texture fetches.

Customized UVs provide a way to increase performance when you are doing simple things like UV manipulation. If you are developing content for mobile VR, try to do all UV manipulation with customized UVs.

Note

Customized UVs are currently only implemented for certain component types such as Static Meshes, Skeletal Meshes, BSP, Landscape, and Mesh Particles.

4.1.9 Mesh considerations

Considerations when adding meshes to your mobile VR game.

When creating meshes to use with your mobile VR Project, the following information must be kept in mind to ensure that what you are creating works with the mobile VR:

- Use as few material IDs on 3D meshes as possible, to save on draw calls.
- Ensure you use static mesh LODs.
- Try to keep the polygon count of your 3D meshes as low as possible.
- Ensure that all static models have a second UV set, so they can use light mapping.
- Use imposter sprites to replace far away static meshes. These are much cheaper to render.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.2 Monoscopic far field rendering for mobile VR

Monoscopic far field rendering provides a performance boost by only rendering distant objects once. This technique is available in Unreal Engine 4.15 and higher.

Stereo divergence decreases as objects are rendered further from the camera. At a certain distance, the stereo rendering of distant objects is indistinguishable from a regular monoscopic rendering.

Monoscopic far field rendering takes advantage of the divergence by dividing the scene into two partitions with a clipping plane: near field and far field. Everything on the near field side of the clipping plane is rendered in stereo, everything on the far field side of the plane is only rendered once and then composited into the near field result.

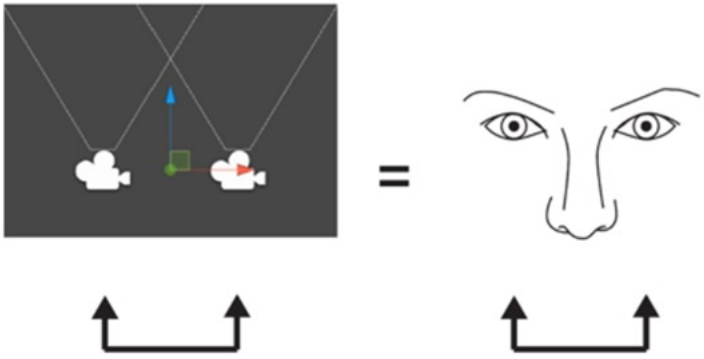

Figure 4-2 Monoscopic rendering illustration

The position difference between your eyes becomes less significant as the distance from the eyes grows. After 50ft, humans cannot perceive depth as well.

Two stereo cameras have a 30ft far plane so they do not render anything that is further than 30ft away. A third camera, known as the mono camera, is positioned as if it was in between your eyes. The mono camera renders pixels that appear more than 30ft away. This results in a strict ordering of pixels when the data from all three cameras are calculated.

In the following diagram, the green object is rendered twice, by both stereo cameras, but only the mono camera renders the yellow object.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

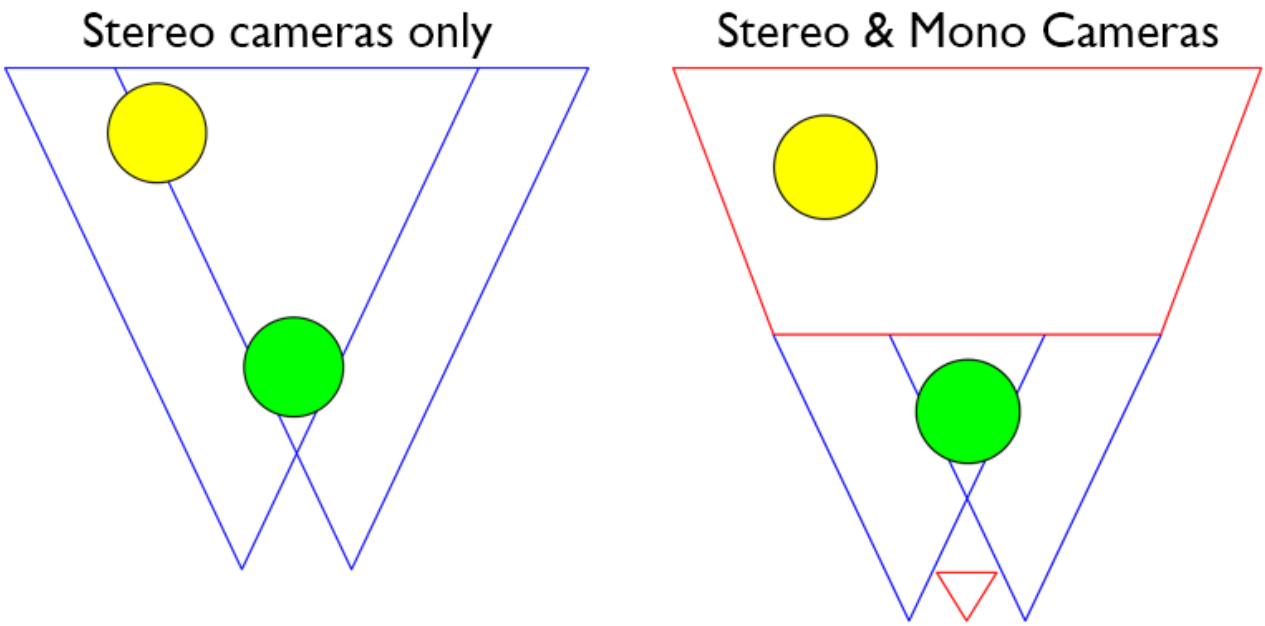

Figure 4-3 Monoscopic cameras

The following image shows results from the near and far field views, and the combined result.

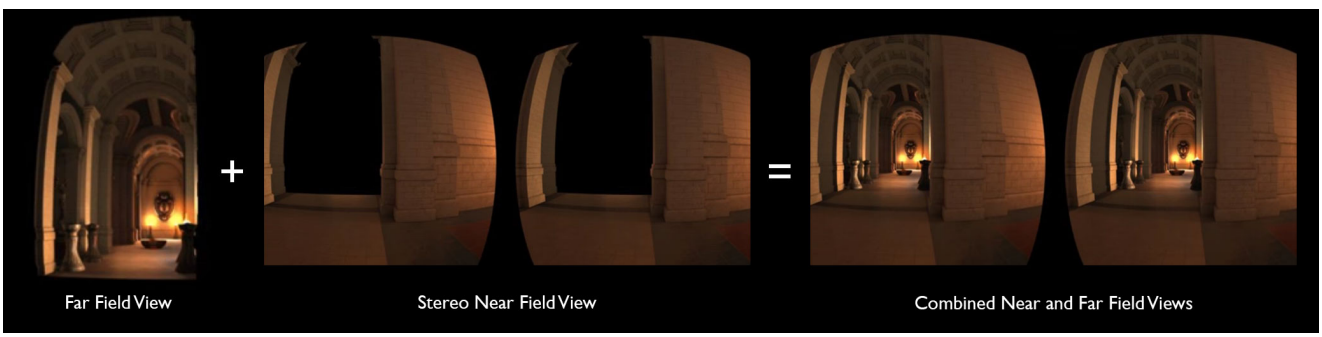

Figure 4-4 Monoscopic in-game view

To enable monoscopic far field rendering, select the **UE4** checkbox under **Project Settings** then **Rendering** and finally, **VR**.

Note

Unreal Engine does not support both mobile multiview and monoscopic far field simultaneously. You must also disable mobile HDR.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3 Understanding Multiview

Multiview rendering enables draw calls to render to several layers of an array texture simultaneously. The vertex shader knows the layer it is writing to, so that the rendering results can be different for each layer.

Virtual reality applications must render all their scenes twice from different view angles to create the illusion of depth. Creating two angles by rendering everything twice, with different view and perspective matrices is not optimal, because it requires setting up a mostly identical draw call multiple times.

The `GL_OVR_multiview` extension addresses this issue by enabling one draw call to render to multiple texture layers of an array texture, removing the overhead of setting up multiple draw calls. The vertex and fragment shaders are then invoked a single time for each texture layer in the attached array texture, and have access to the variable `gl_ViewID_OVR` that is used to select view-dependent input for each layer.

Using this mechanism, an array of view and projection matrices can be used instead of a single matrix. The shader can choose the right matrix for each layer using `gl_ViewID_OVR`, enabling one draw call to render from multiple eye positions with little overhead.

You can also potentially render to multiple layers with only one draw using layered geometry shaders, however this presents a much larger overhead compared to the multiview extension. This is because geometry shaders are very demanding on performance and multiview is a fixed function solution that enables many internal optimizations.

This section contains the following subsections:

- *4.3.1 Stereoscopic rendering* on page 4-58.
- *4.3.2 Benefits of using multiview* on page 4-59.
- *4.3.3 Relative processor time* on page 4-59.
- *4.3.4 Relative GPU time* on page 4-60.
- *4.3.5 Multiview Summary* on page 4-61.

4.3.1 Stereoscopic rendering

Stereoscopic rendering tricks your brain into generating a 3D object by supplying it with two points of view, emulating the way the eyes work.

Two cameras are generated with a slight padding, one on the left, the other on the right. If they share the projection matrix, their view matrices are not the same. This way, you have two different viewpoints on the same scene.

The following diagram shows a stereo camera.

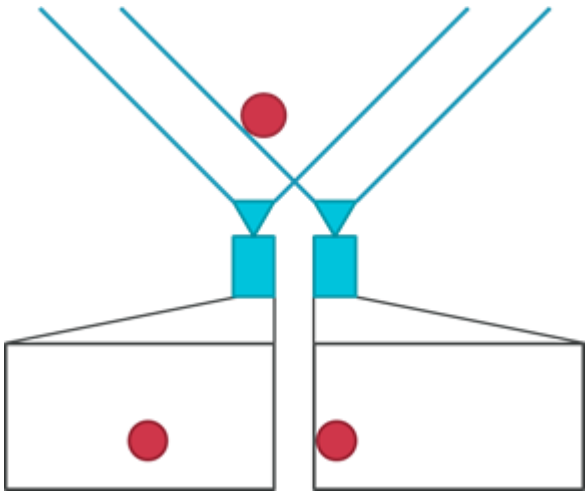

Figure 4-5 Stereo camera

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following is a high-level description of a pipeline for rendering stereo images:

1. Compute and upload left *Model View Projection* (MVP) matrix.
2. Upload Geometry.
3. Emit the left eye draw call.
4. Compute and upload right MVP matrix.
5. Upload Geometry.
6. Emit the right eye draw call.
7. Combine the left and right images onto the backbuffer.

A pattern is visible because you are emitting two draw calls, and sending the same geometries twice. Vertex Buffer Objects can mitigate sending geometry twice, but doubling the draw calls is still a problem because it increases overhead. Multiview enables you to render the same scene with multiple points of view with one draw call.

4.3.2 Benefits of using multiview

A visual example of how a single draw call using multiview is more efficient in VR than two draw calls.

The following diagram shows how your application processor and GPU system is interacting when rendering a frame using regular stereo. First, the processor is working to get all the information ready, then the vertex jobs are executed, and finally the fragment jobs. On this timeline, the light blue are all the jobs that are related to the left eye, the dark blue to the right eye and the orange to the composition, rendering both eyes side by side in a buffer.

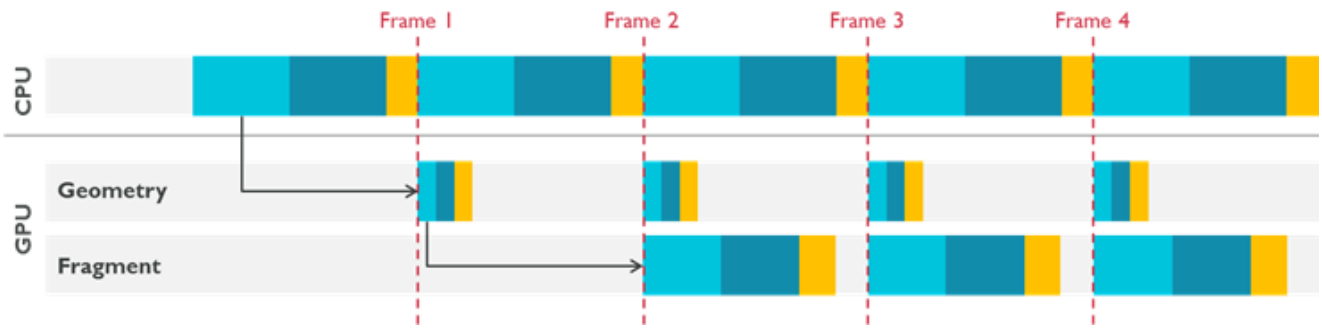

Figure 4-6 Regular stereo job scheduling timeline

In comparison, the following image is the same frame rendered using multiview. The application processor only sends one draw call so it only processed once on the processor. The vertex job is also smaller because it does not execute the non-multiview part of the shader twice. The fragment job, however, remains the same because it must evaluate each pixel of the screen one-by-one.

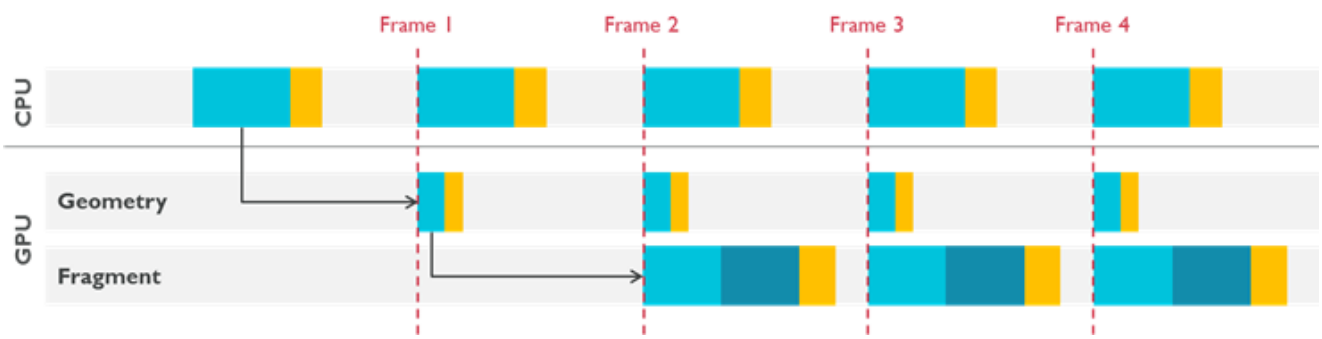

Figure 4-7 Multiview job scheduling timeline

4.3.3 Relative processor time

Multiview mainly works by reducing the number of draw calls you must issue to draw your scene.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Consider an application that is application processor bound. In the application, the number of cubes is changing over time, starting from one and going up to 1,000. Each of them are drawn using a different draw call.

The following image shows the scene with a variable number of cubes.

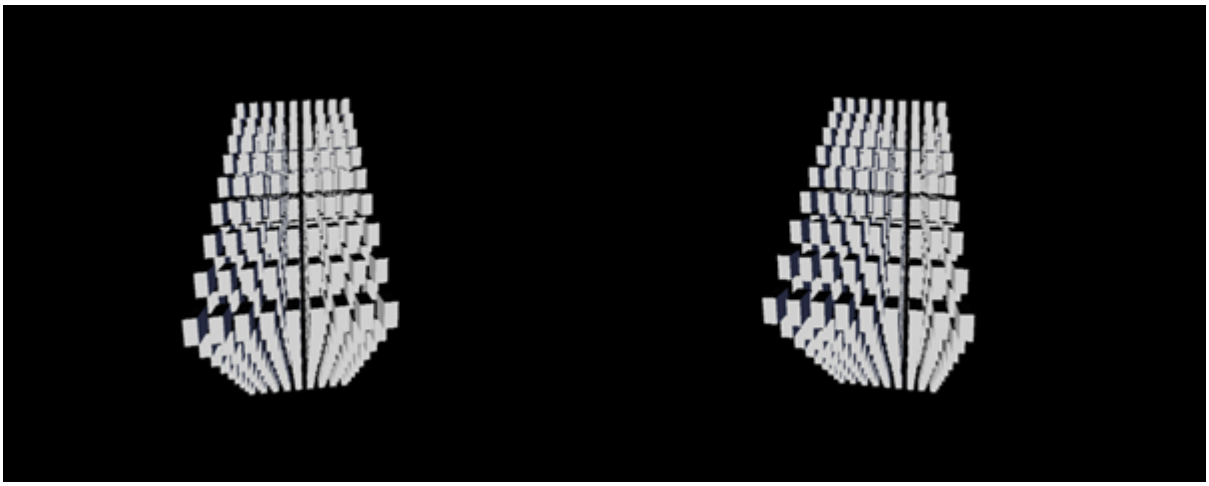

Figure 4-8 Scene used to measure performances

The more cubes are added, the longer the frame takes to render. On the following graph, smaller is better and the relative application processor time is measured between regular stereo shown in blue, and multiview shown in red.

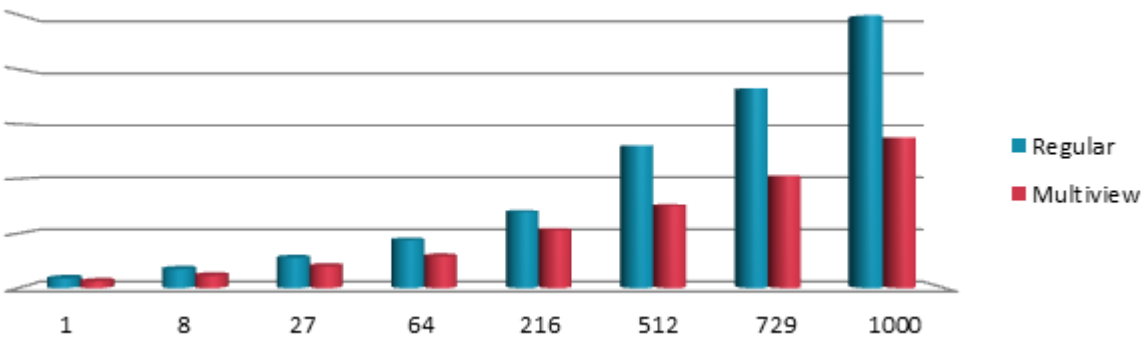

Figure 4-9 Regular versus multiview

4.3.4 Relative GPU time

GPUs run vertex and fragment jobs and can also be optimized using multiview.

On Midgard and Bifrost-based Mali GPUs, only multiview related parts in the vertex shaders are executed for each view.

The following image shows the relative application processor and GPU time, with regular stereo shown in blue and multiview shown in red.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

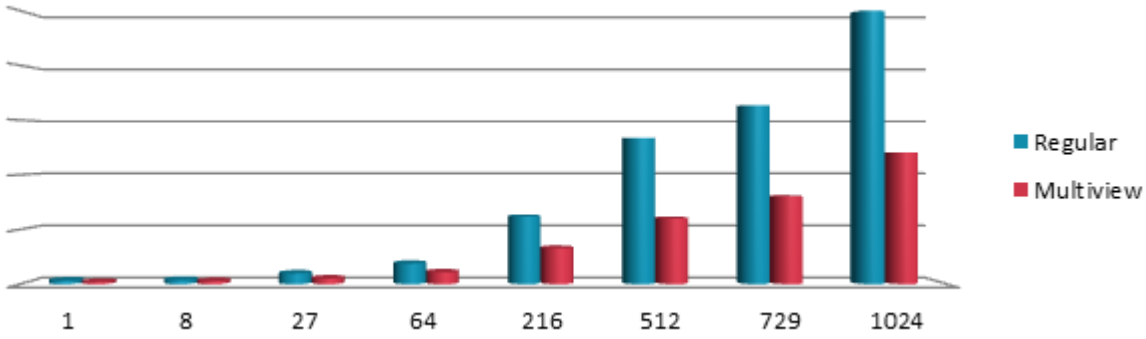

Figure 4-10 Relative GPU time

The savings are immediately visible on this chart, because the GPU is no longer computing most of the shader twice.

4.3.5 Multiview Summary

Multiview is a good extension for application processor bound applications, because you can reasonably expect between 40% and 50% improvements.

If your application is not application processor-bound, do not overlook multiview, because it can also improve vertex processing time for a limited cost.

Multiview renders to an array of textures inside a framebuffer so the result is not directly ready for the front buffer. You must first render the two views side by side. A composition step is required but the time required for this is small compared to the rendering time. You can also integrate the composition step directly into the lens deformation or timewarp process.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 5
Arm Mobile Studio

This chapter discusses Arm Mobile Studio and how the Graphics Analyzer and Streamline components can improve your development when using Unreal Engine 4.

It contains the following sections:

- *5.1 Introduction to Arm Mobile Studio* on page 5-63.
- *5.2 Introduction to Streamline* on page 5-64.
- *5.3 Introduction to Graphics Analyzer* on page 5-69.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.1 Introduction to Arm Mobile Studio

Arm Mobile Studio is a software suite that enables you to visualize performance data of your Android game or application. Identify bottlenecks with Arm Streamline, then use graphics application tracing with Graphics Analyzer to determine exactly where rendering defects occur.

Easy setup with non-root support

All of the tools that are included with Arm Mobile Studio are designed to work on a non-rooted Android device. This enables you to profile and debug your game without device modification.

Detect bottlenecks easily

Use Arm Streamlime *5.2 Introduction to Streamline* on page 5-64 to visualize performance data from the system. Easy-to-use templates help you select the most appropriate set of counters for your target device, so you get the information that you need to optimize your application and filter out the rest of the data.

Optimize for mobile faster, including for 64-bit, quickly determining whether your performance bottleneck relates to the CPU or GPU using easy-to-interpret charts.

Where performance issues are related to the CPU, you can pinpoint the biggest problem areas in your code by drilling down through threads and functions calls, right down to line-byline source code analysis.

Identify graphics issues

Use the Graphics Analyzer *5.3 Introduction to Graphics Analyzer* on page 5-69 to view all the graphics API calls in your application. Easily trace OpenGL ES and Vulkan API calls and understand frame-by-frame the effect on application performance.

The tool captures state changes throughout the application, so you can easily identify rendering defects and trace back to the triggering API function call which caused the problem to occur.

Render the scene draw call by draw call, to see exactly how a frame in the application is composed. This enables you to easily pinpoint any graphical defects in the application and to spot any inefficient or redundant draw calls.

Accelerate shader programs

Use Mali Offline Compiler to compile all shaders and kernels from OpenGL ES, Vulkan and Open CL. Mali Offline Compiler performance reports provide easy visibility of the expected performance and the likely performance bottlenecks of your shader programs on any of the available Mali GPU targets.

Get Arm Mobile Studio

Download Arm Mobile Studio and find the learning resources and the tools that are available for Unreal Engine 4 at *https://developer.arm.com/ue4*. To read more about Arm Mobile Studio visit *https://developer.arm.com/mobile-studio*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2 Introduction to Streamline

Arm Streamline helps you develop Android applications for devices that use Arm processors. To evaluate where the processors in the system are spending most of their time, capture a performance profile of your Android application running on your target device and visualize the data.

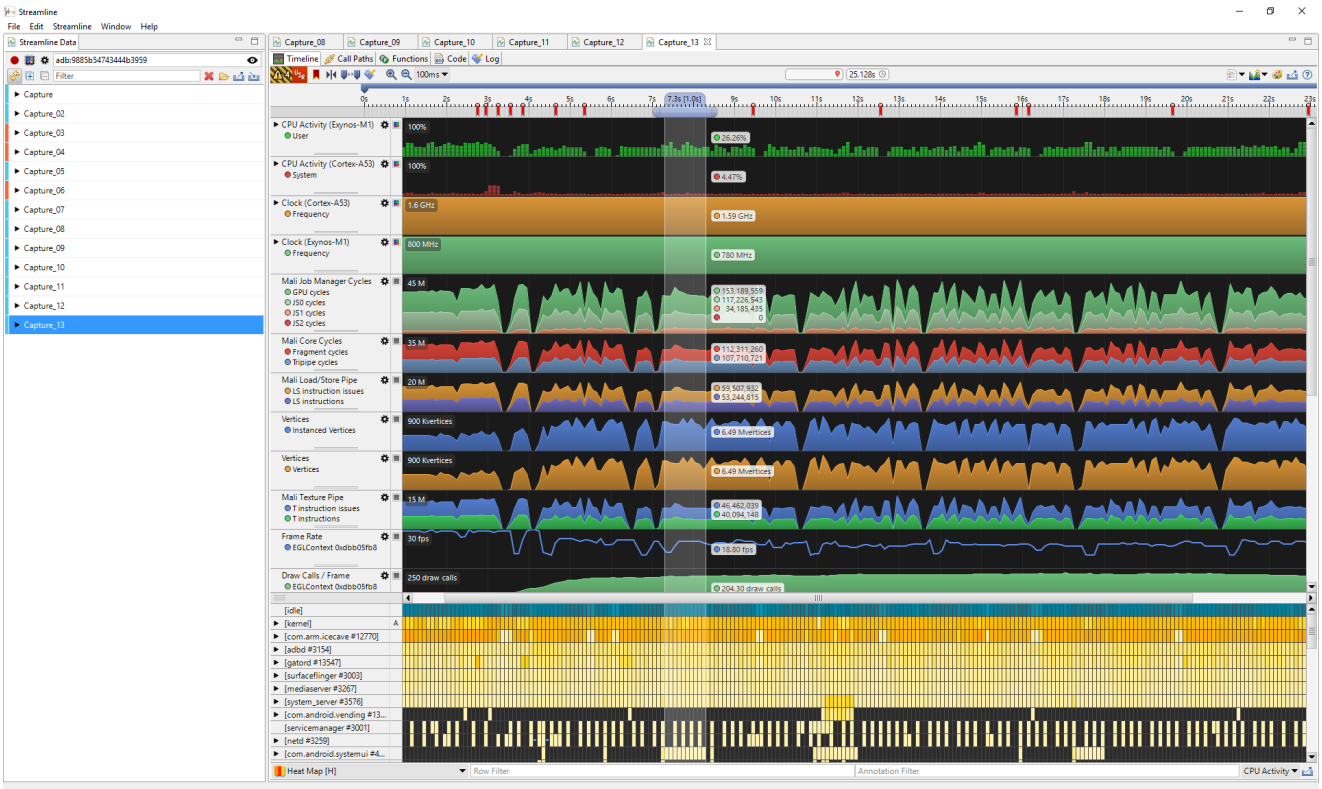

Figure 5-1 Arm Streamline

Streamline can help you to quickly determine whether your performance bottleneck relates to the CPU processing or GPU rendering by providing interactive charts and comprehensive data views.

For CPU bottlenecks, use the native profiling functionality in Arm Streamline to locate specific problem areas in your application code. You can locate problems by investigating how processes, threads, and functions behave, from high-level views, right down to line-by-line source code analysis.

The basic profile is based on a regular sampling of the *Program Counter* (PC) of the running threads, allowing identification of the hotspots in the running application. The target processors that provide the hardware performance counters can supplement the analysis. Hotspot analysis can include the knowledge of hardware events, such as cache misses and branch mispredictions.

For GPU bottlenecks, use performance data from the Mali GPU driver and GPU hardware performance counters to explore the rendering workload efficiency. Visualize the workload breakdown, pipeline loading, and execution characteristics to quickly identify where to apply rendering optimizations.

With Arm Streamline, you can:

- Identify the processor that is the major bottleneck in the performance of your application. Find hot spots in your code that must be targeted for software optimization.
- Use CPU performance counters to provide insights into L1 and L2 cache efficiency, enabling cache-aware profiling.
- Identify the cause of heavy rendering loads which cause poor GPU performance, and use GPU performance counters to identify workload inefficiencies.
- To optimize workloads to reduce device power consumption and improve energy efficiency, use performance counters from the CPU, GPU, and memory system.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Android application profiling

Arm Streamline supports data capture on a non-rooted Android device, collecting CPU performance data and Mali GPU performance data. Therefore, you can profile your game or app without device modification.

Configuring Streamline to collect the right data is easy – use our templates to select the most appropriate set of counters for your target device. Identify bottlenecks and optimize your application for mobile devices faster.

Android system profiling

In addition to the single application profiling for non-root devices, Arm Streamline also supports system-wide Android profiling when running on rooted devices. System profiling enables you to simultaneously monitor all applications and services running on your device, allowing the identification of problematic processes, or scheduling behaviors.

The following image shows Streamline Performance Analyzer displaying the call stack:

[Process]/{Thread}/Code	Total	Self	% Self	Process	% Process	Stack	Location
[idle]	94.10%						-
[gatord #2949]	4.99%						-
[annotate #2948]	0.85%						-
{annotate #2948}	0.51%			98	59.39%		-
main	0.28%	0	0.00%	55	33.33%	128	annotate.c:195
render	0.28%	51	30.91%	54	32.73%	208	annotate.c:122
gator_annotate_str	0.01%	0	0.00%	2	1.21%	320	streamline_annotate.c:755
gator_buf_write_time	< 0.01%	1	0.61%	1	0.61%	384	streamline_annotate.c:356
gator_get_thread	< 0.01%	1	0.61%	1	0.61%	400	streamline_annotate.c:636
gator_annotate_color	< 0.01%	0	0.00%	1	0.61%	336	streamline_annotate.c:776
gator_buf_write_time	< 0.01%	0	0.00%	1	0.61%	400	streamline_annotate.c:356
gator_buf_write_long	< 0.01%	1	0.61%	1	0.61%	544	streamline_annotate.c:331
gator_annotate_color	< 0.01%	1	0.61%	1	0.61%	256	streamline_annotate.c:776
<unknown code in kernel>	0.18%	35	21.21%	35	21.21%	0	<anonymous>
<unknown code in libc-2.17.so>	0.03%	5	3.03%	5	3.03%	0	<anonymous>
<unknown code in ld-2.17.so>	0.01%	2	1.21%	2	1.21%	0	<anonymous>
<unknown code in libpthread-2.17.so>	< 0.01%	1	0.61%	1	0.61%	0	<anonymous>
{gator-annotate #2955}	0.35%			67	40.61%		-
<unknown code in kernel>	0.28%	55	33.33%	55	33.33%	0	<anonymous>
<unknown code in libpthread-2.17.so>	0.04%	8	4.85%	8	4.85%	0	<anonymous>
main	0.01%	0	0.00%	2	1.21%	128	annotate.c:195
gator_func	0.01%	2	1.21%	2	1.21%	272	streamline_annotate.c:471
__udivsi3	< 0.01%	1	0.61%	1	0.61%	0	annotate
<unknown code in libc-2.17.so>	< 0.01%	1	0.61%	1	0.61%	0	<anonymous>
[gdbserver #2945]	0.03%						-
{gdbserver #2945}	0.03%			5	100.00%		-
<unknown code in kernel>	0.03%	5	100.00%	5	100.00%	0	<anonymous>
[kernel]	0.02%						-
[dhclient #2545]	< 0.01%						-
{dhclient #2545}	< 0.01%			1	100.00%		-
<unknown code in dhclient>	< 0.01%	1	100.00%	1	100.00%	0	<anonymous>
[sshd #2831]	< 0.01%						-
{sshd #2831}	< 0.01%			1	100.00%		-
<unknown code in libc-2.17.so>	< 0.01%	1	100.00%	1	100.00%	0	<anonymous>
[accounts-daemon #2238]	0.00%						-
{accounts-daemon #2238}	0.00%			0			-
[gatord #1337]	0.00%						-
{gatord-main #1337}	0.00%			0			-

Figure 5-2 Streamline Performance Analyzer call stack

Get started with Arm Streamline

Arm Streamline is available as part of the *Arm Mobile Studio* software suite.

- *Download Arm Mobile Studio*
- *Get started with Arm Streamline*
- *Streamline documentation*
- *Learning resources*

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This section contains the following subsections:

- *5.2.1 GPU counters in Arm Streamline* on page 5-67.
- *5.2.2 Using Arm Streamline to analyze graphs* on page 5-68.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2.1 GPU counters in Arm Streamline

Arm Streamline captures data from the hardware counters in the application processor and the Mali GPU. It displays these results as graphs.

To select an appropriate counter template for your target GPU, choose **Add counters from a template**. Counter templates are pre-defined sets of counters that have been chosen to enable you to perform an initial performance review of both CPU and GPU behavior.

The following figure shows the **Counter Configuration** window where you select counters to record and display in Streamline Performance Analyzer:

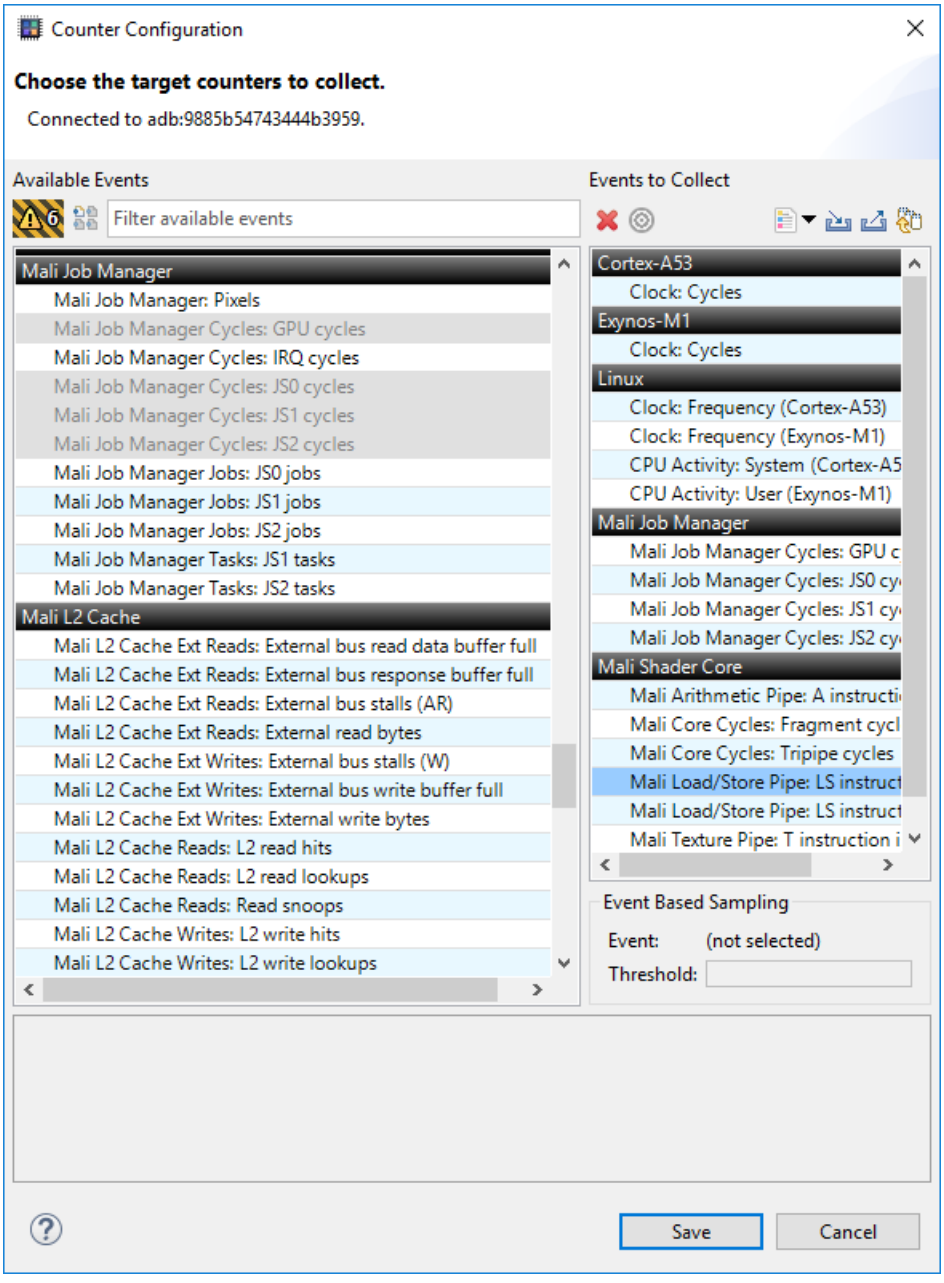

Figure 5-3 Arm Streamline counters

You can also add virtual counters that show relationships between values. For example:

- Cache hit to miss ratios.
- Triggers that generate a value that is based on a defined criteria.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2.2 Using Arm Streamline to analyze graphs

In Arm Streamline, you can view multiple graphs. The graphs include information on the activity of all the processors that are used by the application, a stack trace displayed as a call graph, and native application profiling.

You can make the following observations from graphs in Arm Streamline:

- Short tasks generate small spikes of activity.
- Intensive tasks generate high processor utilization for extended periods.
- What processor is active at what time.

Note

Arm Streamline does not directly measure frame rate. You can measure it directly in your application or you can measure it indirectly by:

- Looking for API calls such as `EGLswapbuffers`.
- Looking for repeating patterns in the processor graphs.

Procedure for analyzing graphs

1. Look at the following graphs

- `GPU Vertex activity`.
- `GPU Fragment activity`.
- <*Application processor*> `Instruction: Executed`.

2. Analyze the graphs

- Look for the processor with the highest and longest graph. This processor is used the most intensively.
- If it is difficult to find a single processor that is taking too much time, the problem might be bandwidth overuse or graphics pipeline stalls.
- When you have identified the most intensively used processor, take more measurements to isolate the problem.
- If all graphs are high, your application is making good use of the Mali GPU.

Note

For the most accurate measurements, Zoom in to `EGLSwapBuffers()` calls and use the calipers to isolate a frame.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.3 Introduction to Graphics Analyzer

The Graphics Analyzer is an API trace and debug tool for OpenGL ES, Vulkan, and OpenCL. It helps you to identify possible issues with your applications.

Graphics Analyzer provides you with a set of tools that enable you to monitor and analyze the operation of your application.

It can display different kinds of information and statistics that tell you about the data accesses and processing in your application. You can use this information to help you debug applications or to identify performance bottlenecks.

The following image shows Graphics Analyzer:

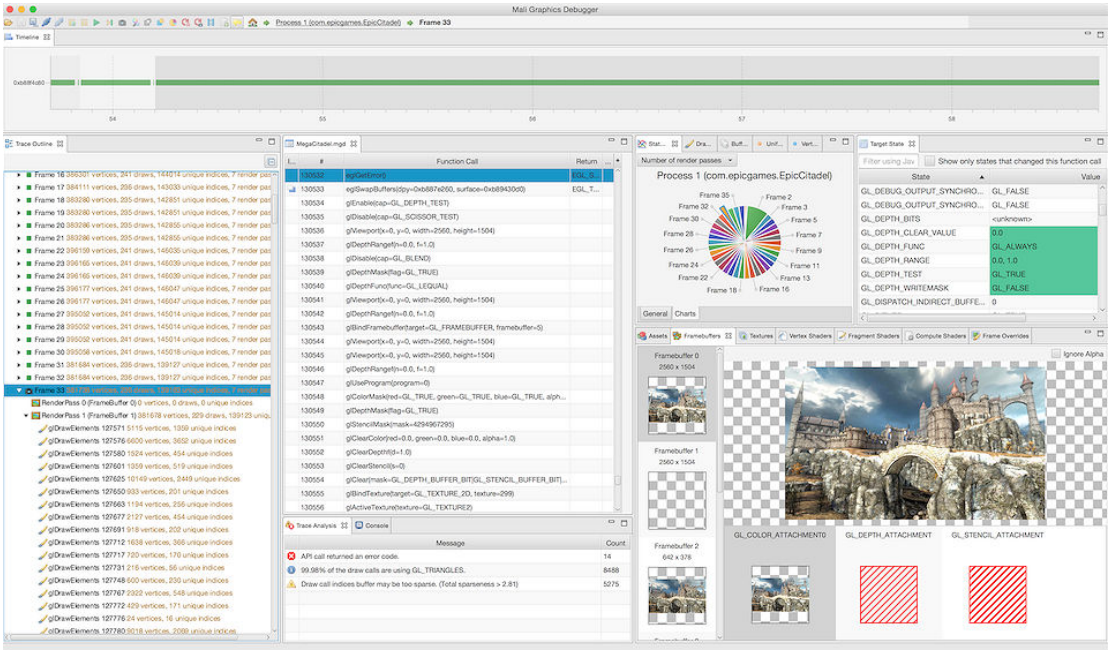

Figure 5-4 Graphics Analyzer

For more information, see *https://developer.arm.com/tools-and-software/embedded/arm-development-studio/components/graphics-analyzer*.

This section contains the following subsections:

- *5.3.1 Graphics Analyzer features* on page 5-69.
- *5.3.2 Enabling Graphics Analyzer support in Unreal Engine applications* on page 5-74.

5.3.1 Graphics Analyzer features

The Graphics Analyzer has a number of features that help you identify possible issues with your applications.

Graphics Analyzer enables you to trace API calls and understand their frame-by-frame effects in your application. It can also highlight common API problems and provide performance improvement advice.

Graphics Analyzer has the following features and benefits:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Trace API calls

Draw-call by draw-call stepping helps you to identify draw-call related issues, redundant draw calls, and other opportunities to optimize.

You can trace OpenGL ES, Vulkan, EGL, and OpenCL API calls with arguments, time, process ID, and thread ID.

There is full trace from the start and it automatically checks for errors. There are search capabilities and it includes multi-process support.

An automated trace view enables you to run a range of standard Graphics Analyzer commands automatically when a specific frame is encountered.

Outline view

The outline view enables you to investigate at the frame level to find what draw calls have a high geometry impact. It provides you with quick access to important graphics events such as frames, render targets, and draw calls.

The following image shows the outline view:

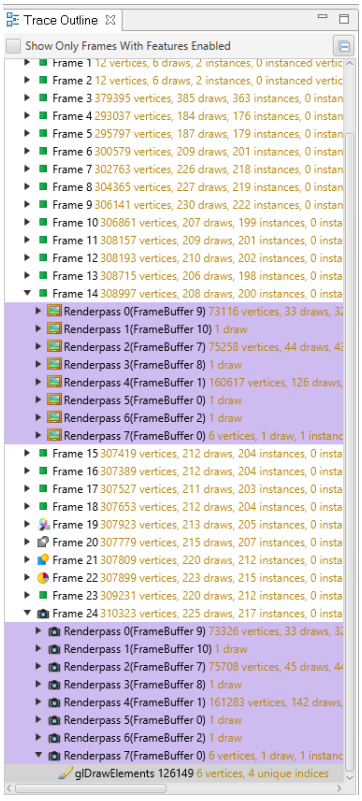

Figure 5-5 Graphics Analyzer outline view

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Target state debug

Target state debug provides a complete view of the graphics state and state changes, at any point in the trace. You can also use it to discover when and how a state is changed and you can jump between the outline view and the trace view seamlessly.

Every time a new API call is selected in the trace, the state is updated. This is useful for debugging problems and understanding the causes of performance issues.

The following image shows target state view:

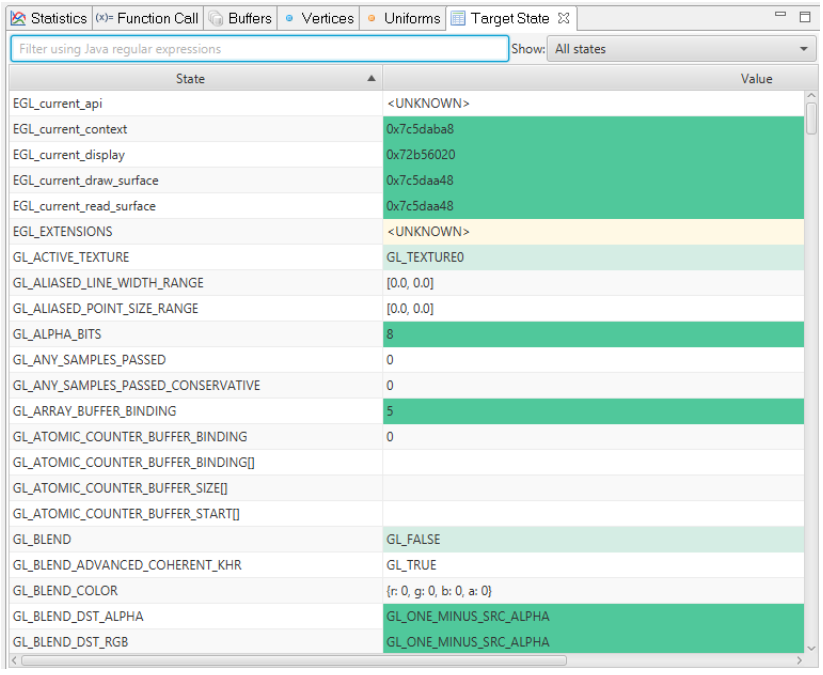

Figure 5-6 Graphics Analyzer target state view

Trace statistics and analysis

You can analyze the trace to find common issues such as performance warnings.

Graphics Analyzer provides per-frame and per draw-call, high-level statistics about the trace.

Graphics Analyzer also shows API misusage by highlighting problematic API calls in the trace view.

Uniform and vertex attribute data views

When you select a draw call, Graphics Analyzer shows all the associated uniform and vertex attribute data. The vertex attribute and uniform views enable you to see vertex and index buffer data. You can analyze these to see their usage in your application.

The uniform view shows uniform values such as samplers, matrices, and arrays.

The vertex attributes view shows vertex attributes such as vertex names and vertex positions.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Buffer and textures access view

Texture view enables you to visualize the texture usage in your applications, helping you to identify opportunities to compress textures or change formats.

Client and server-side buffers are captured every time they change. This enables you to see how each API call affects the buffers.

All textures uploaded are captured at native resolution, enabling you to check the size, format, and type of the textures. You can also access all the large assets including data buffers and textures, for debugging.

The following image shows the texture view:

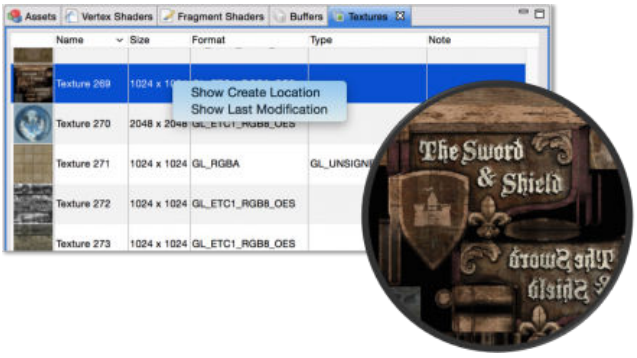

Figure 5-7 Graphics Analyzer texture view

Shaders reports and statistics

Shader statistics and cycle counts help you understand which vertex and fragment shaders are the most computationally expensive.

All the shaders used by the application are measured. For each draw call, Graphics Analyzer counts the number of times the shader has been executed, and calculates overall statistics.

Each shader is also compiled with the Mali Offline Compiler. This statically analyzes the shader to display the number of instructions for each GPU pipeline, the number of work registers, and uniform registers.

Frame capture and analysis

Graphics Analyzer can capture frames so you can analyze the effect of each draw call.

A native resolution snapshot of each framebuffer is captured after every draw call. The snapshot capture happens on the target device, so you can investigate target-dependent bugs or precision issues.

You can also export the captured images for separate analysis.

Alternative drawing modes

Graphics Analyzer includes specialist drawing modes that help you analyze your frames. You can use these for both live rendering and frame captures:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Native mode

In Native mode, frames are rendered with the original shaders.

The following image shows native mode:

Figure 5-8 Native mode

Overdraw mode

The overdraw mode highlights where overdraw is present. You can analyze the impact of overdrawn pixels on each framebuffer by looking at the overdraw captures and histograms.

The following image shows overdraw mode:

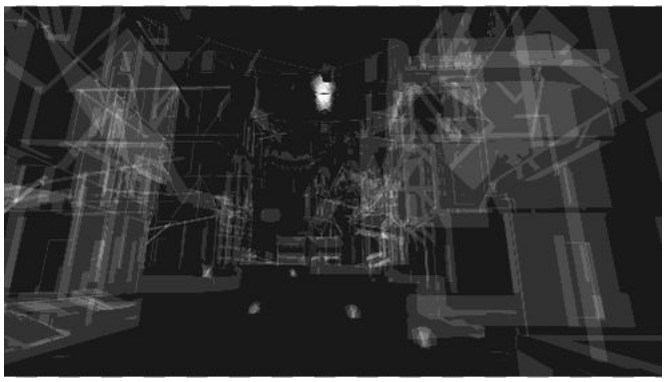

Figure 5-9 Overdraw mode

Shader map mode

In shader map mode, native shaders are replaced with different solid colors. This shows you where the different shaders are used in a frame.

The following image shows shader map mode:

Figure 5-10 Shader map mode

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Fragment count mode

In fragment count mode, all the fragments that are processed for each frame are counted. This shows you how many fragments are used in different parts of a frame.

5.3.2 Enabling Graphics Analyzer support in Unreal Engine applications

How to add support for Graphics Analyzer on a device to your Unreal Engine application.

Enabling Graphics Analyzer support in versions of Unreal Engine 4.15 and higher

This describes how to enable Graphics Analyzer support on Unreal Engine 4.15 and higher engine versions.

In Unreal Engine 4.15, Epic introduced the ability to directly enable profiling on mobile devices from the editor.

For Unreal Engine version 4.15 and higher, you can activate Graphics Analyzer by selecting the option in the project settings.

The first step is to open your project.

Open the **project settings** window. From there, in the left menu, find the **Android** option, then **Platforms** on the right, **Graphic Debugger** and finally, select **Mali Graphics Debugger** in the list.

The following image shows the project settings:

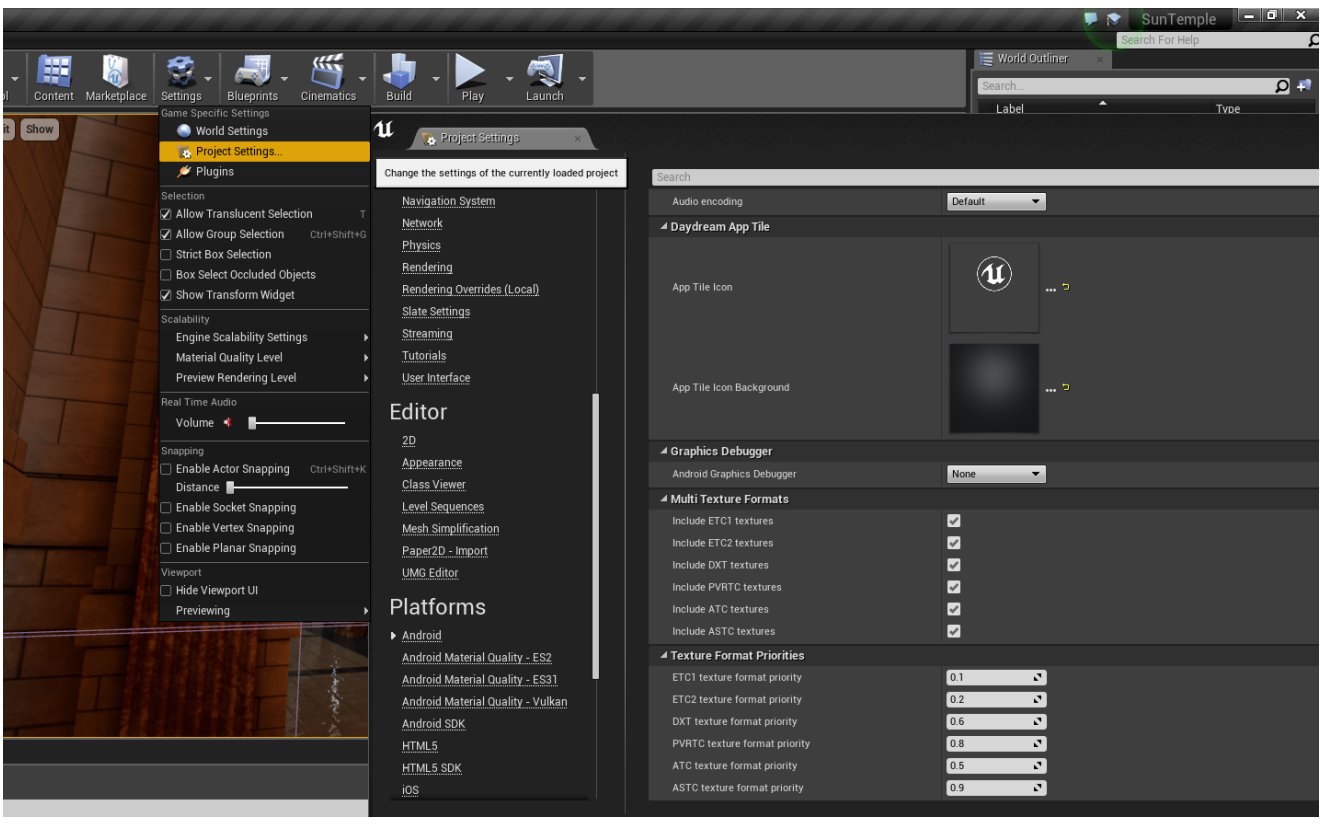

Figure 5-11 Unreal Engine project settings

When you have selected **Mali Graphics Debugger** in the list, you must enter the path to the Graphics Analyzer installation on your host computer. You must have a working version of Graphics Analyzer installed on your system.

For information on how to install Graphics Analyzer see the documentation at *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/learn/graphics-analyzer-documentation*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following image shows the project settings with the Graphics Analyzer path:

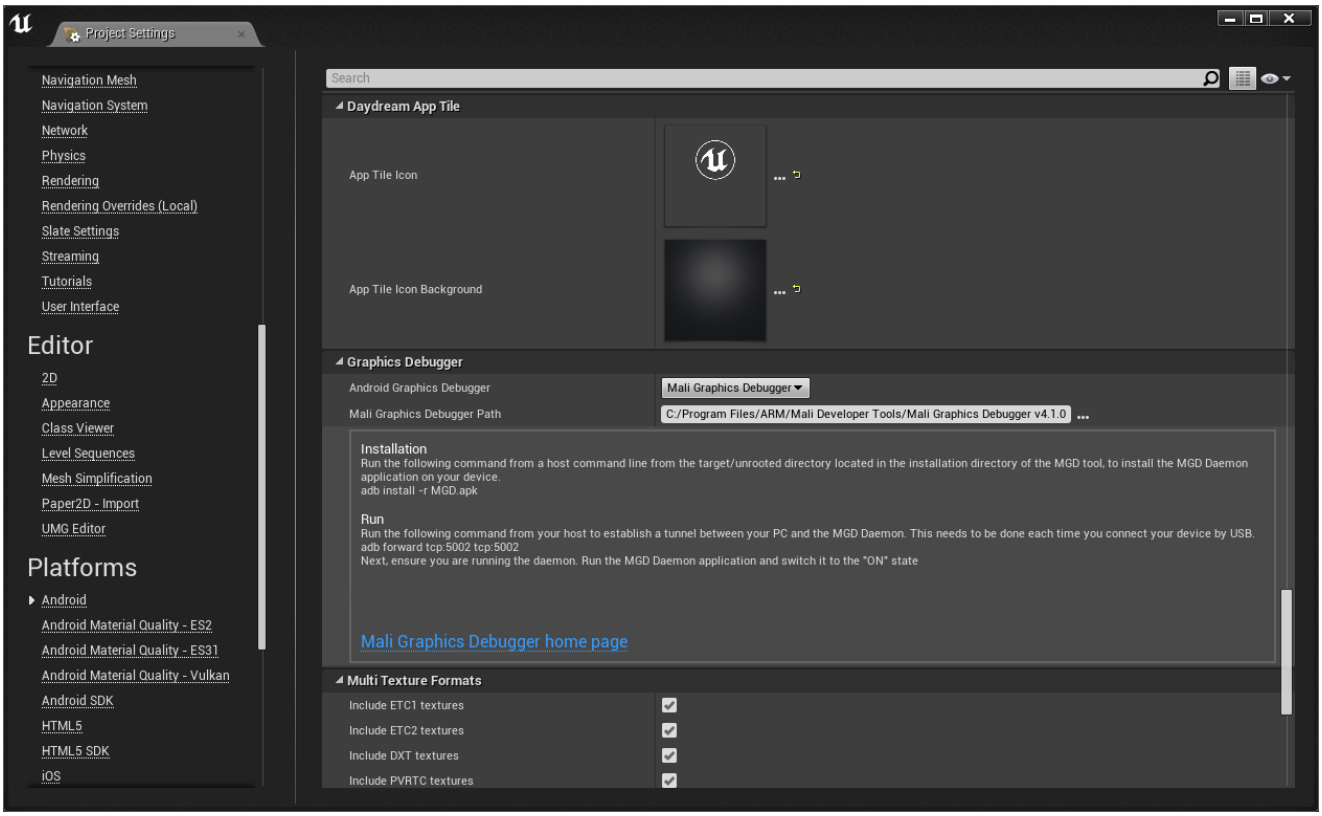

Figure 5-12 Project settings with the Graphics Analyzer path

Install the Graphics Analyzer Daemon onto your target device. To do this, locate Graphics Analyzer installation directory `target\android\arm`.

Install the Graphics Analyzer Daemon with:

```
adb install -r AGA.apk
```

Ensure that the device is visible by running:

```
adb devices
```

Before running your app, you must run this command from the host PC:

```
adb forward tcp:5002 tcp:5002
```

Run the Graphics Analyzer daemon application on the target phone and activate the daemon.

The following image shows the Graphics Analyzer daemon on a device:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

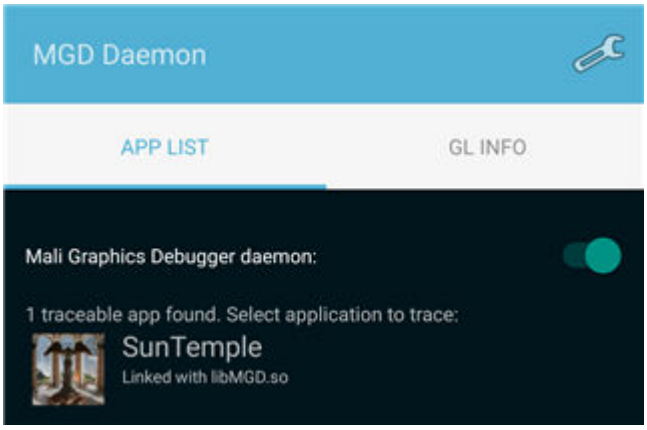

Figure 5-13 Graphics Analyzer daemon on a device

You can now connect the daemon to Graphics Analyzer on the host PC, start your application, and begin debugging it.

For information on how to use Graphics Analyzer see the documentation at *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/learn/graphics-analyzer-documentation*.

The following image shows an example application on a device:

Figure 5-14 Example application on a device

The following image shows Graphics Analyzer connected to a device:

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

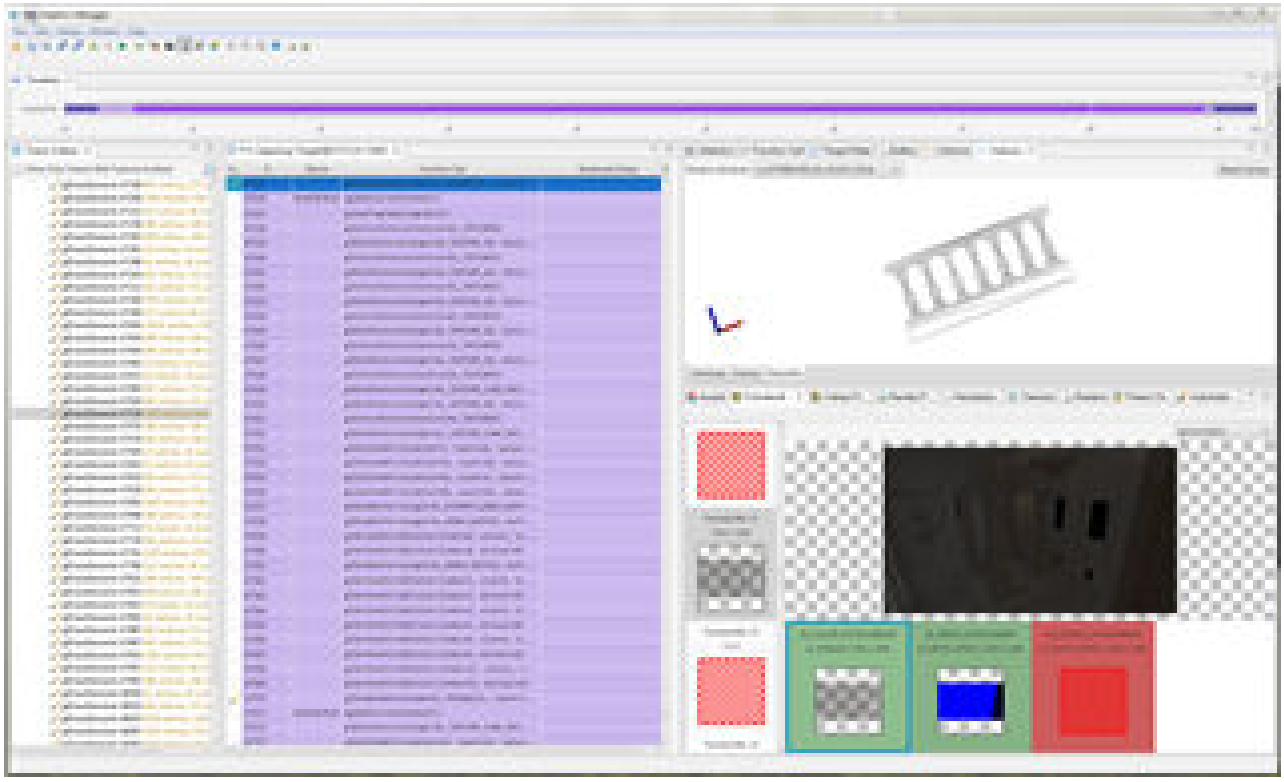

Figure 5-15 Graphics Analyzer connected to a device

Sometimes, if you have complex applications, the game thread might time out waiting for the render thread. This is recorded in the log.

To see the log type:

```
adb logcat -s "UE4"
```

GameThread timed out waiting for RenderThread after 30.01 seconds.

In that case, open the Unreal console by tapping four fingers on the device screen.

To disable timeout, type the following into the Unreal console:

```
g.TimeoutForBlockOnRenderFence 9999999
```

Enabling Graphics Analyzer support in versions of Unreal Engine lower than 4.15

This describes how to enable Graphics Analyzer support in versions of Unreal Engine lower than 4.15.

The procedure is:

1. Add the interceptor library to the build system.

Download a version of Unreal Engine from the sources available on Github. For more information on this step, see the Unreal Engine documentation at *https://docs.unrealengine.com/latest/INT/*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. Edit the activity to load the interceptor library.

When you have a working copy of Unreal Engine, you must get Graphics Analyzer working.

Locate the `android-non-root` directory in your Graphics Analyzer installation directory, and your Unreal Engine installation directory. This is where you cloned the repository.

Copy the `android-non-root` directory to `Engine\Build\Android\Java\`.

You must change the Android makefile to ensure that the interceptor is properly packaged inside the engine build. To do this, edit the Android.mk file under `Engine/Build/Android/Java/jni/`.

Add the following line at the end:

```
include $(LOCAL_PATH)/../android-non-root/MGD.mk.
```

The result should look like this:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := UE4
LOCAL_SRC_FILES := $(TARGET_ARCH_ABI)/libUE4.so
include $(PREBUILT_SHARED_LIBRARY)

include $(LOCAL_PATH)/../android-non-root/MGD.mk
```

Instruct the main game activity to load the Graphics Analyzer library.

Locate GameActivity.java inside `Engine\Build\Android\Java\src\com\epicgames\ue4\` and edit the `onCreate` function to look like this:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
     super.onCreate(savedInstanceState);
     try {
          System.loadLibrary("MGD");
     }
     catch( UnsatisfiedLinkError e ){
          Log.debug( "libMGD not loaded" );
     }

     // create splashscreen dialog (if launched by SplashActivity)
     Bundle intentBundle = getIntent().getExtras();

     // Unreal Engine code continues there
```


Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3. Install the Graphics Analyzer Daemon application on the target device.

Prepare your device by installing the Graphics Analyzer daemon on the target device using the following command in the android-non-root directory:

```
adb install -r AGA.apk
```

Before running your app, ensure that the device is visible by running:

```
adb devices
```

Run the following command from the host workstation:

```
adb forward tcp:5002 tcp:5002
```

Run the Graphics Analyzer daemon application on the target device and activate the daemon.

The following image shows the Graphics Analyzer daemon on a device:

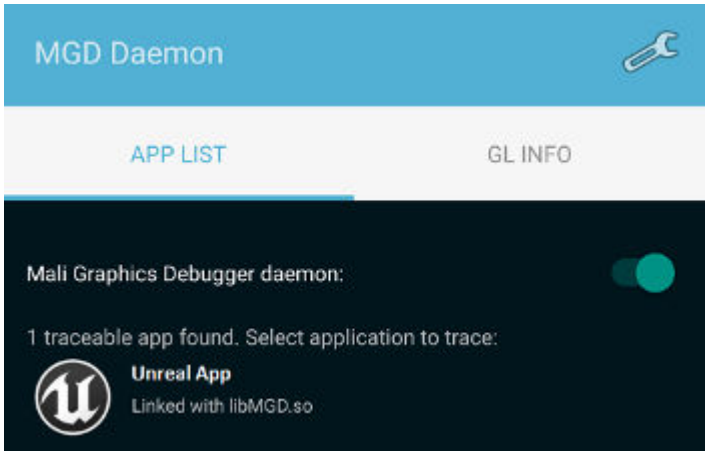

Figure 5-16 Graphics Analyzer daemon

At this point you can connect the daemon to Graphics Analyzer on the host workstation, start your application, and begin debugging it.

For information about using Graphics Analyzer, see the documentation at

https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/learn/graphics-analyzer-documentation*https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/learn/graphics-analyzer-documentation*.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix A
Revisions

This appendix describes the changes between released issues of this book.

It contains the following section:

- *A.1 Revisions* on page Appx-A-81.

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.1 Revisions

This appendix describes the technical changes between released issues of this book.

Table A-1 Issue 1.0

Change	Location	Affects
First release	-	-

Table A-2 Issue 1.1

Change	Location	Affects
Added the Arm Mobile Studio chapter.	*Chapter 5 Arm Mobile Studio* on page 5-62	Version 1.1 first release
Renamed all instances of Mali Graphics Debugger to Graphics Analyzer.	Throughout the document	Version 1.1 first release
Moved Graphics Analyzer content to within the Arm Mobile Studio chapter.	*5.3 Introduction to Graphics Analyzer* on page 5-69	Version 1.1 first release
Deleted all references to Enlighten within the main text.	Throughout the document	Version 1.1 first release

Copyright © 2017, 2019 Arm Limited or its affiliates. All rights reserved.
Non-Confidential