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
Contents
Graphics Analyzer User Guide

Preface
About this book ...................................................... ...................................................... 8

Chapter 1  Introduction
1.1  Installation package ................................................ ................................................ 1-11
1.2  Performance ................................................................. ................................................ 1-12

Chapter 2  Minimum requirements
2.1  Host system requirements ........................................... ........................................... 2-14
2.2  Target system requirements ................................................ ........................................... 2-15

Chapter 3  Target installation
3.1  Unrooted Android ................................................................. ................................................ 3-18
3.2  System-wide Android ................................................................. ................................................ 3-26
3.3  Connect to your Android device from the command line ................................................ ................................................ 3-33
3.4  Linux ................................................................. ................................................ 3-34
3.5  Chrome OS ................................................................. ................................................ 3-37
3.6  webOS ................................................................. ................................................ 3-40
3.7  Troubleshooting ................................................................. ................................................ 3-43

Chapter 4  Device Manager
4.1  Android devices ................................................................. ................................................ 4-46
4.2  Linux devices ................................................................. ................................................ 4-49
4.3  Connect to an IP address ................................................................. ................................................ 4-50
Chapter 5  Getting started

5.1  Run the host GUI ................................................................. 5-52
5.2  Trace an application ............................................................ 5-53
5.3  Process Configuration .......................................................... 5-54
5.4  Dealing with multiple processes ........................................ 5-56
5.5  Pause, step frames, and resume ........................................ 5-57
5.6  Capturing frame buffer content ........................................... 5-58
5.7  Capturing all frame buffer attachments ............................. 5-59
5.8  Limitations on capturing all frame buffer attachments ......... 5-60
5.9  Analyzing overdraw ........................................................... 5-61
5.10 Analyzing the shader map .................................................. 5-62
5.11 Overdraw and shader map limitations ............................... 5-64
5.12 Analyzing the fragment count .......................................... 5-65
5.13 Replaying a frame ............................................................. 5-66
5.14 Frame replay limitations ................................................... 5-67
5.15 Frame overrides ............................................................... 5-68
5.16 Debugging an OpenCL application .................................... 5-72
5.17 Using GPUVerify to validate OpenCL kernels .................. 5-73
5.18 Comparing state between function calls ......................... 5-75
5.19 Bookmarks ........................................................................ 5-76
5.20 Dealing with VR applications ........................................... 5-77
5.21 Tracing an application that is already running .................. 5-78

Chapter 6  The Graphics Analyzer interface

6.1  Perspectives ......................................................................... 6-81
6.2  Trace view ........................................................................... 6-82
6.3  Trace Outline view ............................................................. 6-85
6.4  Timeline view ...................................................................... 6-86
6.5  Statistics view ...................................................................... 6-87
6.6  Function Call view ............................................................. 6-88
6.7  Trace Analysis view ............................................................ 6-89
6.8  Target State view ............................................................... 6-90
6.9  Buffers view ........................................................................ 6-91
6.10 OpenGL ES Framebuffers view .......................................... 6-92
6.11 Vulkan Frame Capture view .............................................. 6-94
6.12 Assets view ....................................................................... 6-96
6.13 Shaders view ...................................................................... 6-98
6.14 Textures view ...................................................................... 6-99
6.15 Images view ........................................................................ 6-100
6.16 Vertices view ...................................................................... 6-101
6.17 Uniforms view ..................................................................... 6-103
6.18 Automated Trace view ....................................................... 6-104
6.19 Render Pass Dependencies view ....................................... 6-107
6.20 Bookmarks view ............................................................... 6-109
6.21 Console view ....................................................................... 6-110
6.22 Scripting view ..................................................................... 6-111
6.23 Filtering and searching in Graphics Analyzer .................. 6-113
6.24 Host-side headless mode .................................................... 6-114
6.25 Target-side headless mode ................................................ 6-116
Chapter 7 Integration with Arm Streamline
7.1 Installation ...................................................... ......................................................  7-123
7.2 Using Arm Streamline annotations ................................................................. 7-124

Chapter 8 Known issues
8.1 OpenGL ES extensions ..........................................................................................  8-127
8.2 Shading language version ................................................................................. 8-128
8.3 Shader compiler .................................................................................................. 8-129
8.4 API asset coverage .............................................................................................. 8-130
8.5 Memory .............................................................................................................. 8-131
8.6 Partial support for earlier trace versions .............................................................. 8-132
8.7 Graphics Analyzer becomes unresponsive when closed while dynamic help is open ................................................................................................................. 8-133
8.8 Issues viewing Khronos reference pages ..............................................................  8-134
8.9 Intercepting without using LD_PRELOAD ............................................................ 8-135
8.10 Multiple drivers installed on the system ............................................................  8-136
8.11 Daydream VR compositor skips right eye execution on capturing a frame ....... 8-137

Appendix A Analytics
A.1 Analytics information .................................................................................... Appx-A-139
A.2 Disable analytics data collection ........................................................................ Appx-A-140
Preface

This preface introduces the Graphics Analyzer User Guide.

It contains the following:
• About this book on page 8.
About this book

This book describes how to install and use Graphics Analyzer on a Windows, Linux, or macOS host to examine an application running on an Android or Linux target.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Graphics Analyzer is a tool to help OpenGL ES and Vulkan developers get the best out of their applications through analysis at the API level.

Chapter 2 Minimum requirements
This chapter describes the prerequisites for the host and target systems to run Graphics Analyzer.

Chapter 3 Target installation
Graphics Analyzer has two target components that interact to collect and transmit trace information from your application to the GUI on the host. They are the interceptor library and the daemon application.

Chapter 4 Device Manager
The Device Manager is used to detect and to connect to devices that are running the Graphics Analyzer daemon.

Chapter 5 Getting started
This chapter describes how to use the host GUI to configure and perform a trace and to capture frame buffer content while capturing a trace. It also describes how to use the capture modes in Graphics Analyzer to capture additional content.

Chapter 6 The Graphics Analyzer interface
This chapter describes the Graphics Analyzer host GUI which provides different views over the captured application trace. The GUI also provides access to headless mode, which enables automated data capture on the target.

Chapter 7 Integration with Arm Streamline
The Graphics Analyzer interceptor library generates Arm Streamline annotations and chart information.

Chapter 8 Known issues
This chapter describes some known issues in this release of Graphics Analyzer.

Appendix A Analytics

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.

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\textsuperscript{®} Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

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 Graphics Analyzer User Guide.
• The number 101545_0505_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.

Other information

• Arm\textsuperscript{®} Developer.
• Arm\textsuperscript{®} Information Center.
• Arm\textsuperscript{®} Technical Support Knowledge Articles.
• Technical Support.
• Arm\textsuperscript{®} Glossary.
Chapter 1
Introduction

Graphics Analyzer is a tool to help OpenGL ES and Vulkan developers get the best out of their applications through analysis at the API level.

The tool allows you to observe API call arguments and return values, and interact with a running target application to investigate the effect of individual API calls. It highlights attempted misuse of the API, and gives recommendations for improvements.

For help and support from Arm and fellow developers, visit the Arm Mali™ Graphics Community.

It contains the following sections:
- 1.1 Installation package on page 1-11.
- 1.2 Performance on page 1-12.
1.1 Installation package

The installation package for Graphics Analyzer contains everything that you need to start investigating GPU applications on a desktop computer.

Download the installation package appropriate to your platform. Graphics Analyzer is available for Windows, Linux, and macOS.

The installation package contains three main components:
• The GUI application
• The target interceptor components
• Sample traces.

After you have installed Graphics Analyzer, the extracted directory hierarchy on the host contains a target directory with the following subdirectories:

**linux/soft_float/**
Contains daemon and Graphics Analyzer interceptor for Linux-based Armv7 or Armv8 (32-bit) target devices.

**linux/hard_float/**
Contains daemon and Graphics Analyzer interceptor for Linux-based Armv7 or Armv8 (32-bit) hard-float target devices.

**linux/arm64/**
Contains daemon and Graphics Analyzer interceptor for Linux-based Armv8 (64-bit) target devices.

**android/arm/**
Contains daemon and Graphics Analyzer interceptor for Android-based Armv8 (64-bit) and Armv7 (32-bit) target devices.

**android/intel/**
Contains daemon and Graphics Analyzer interceptor for Android-based x86-64 (64-bit) and x86 (32-bit) target devices.

--- Note ---
You must manually add the libraries to your execution environment, see Chapter 3 Target installation on page 3-16 for details.
1.2 Performance

The Graphics Analyzer aims to be as unobtrusive as possible. However, there is a performance impact imposed by the gathering and sending of trace data to the host.

This impact is particularly pronounced when you request a snapshot of a frame as this causes pixel read-back calls to be inserted by the interceptor after each draw call.

The interceptor library has negligible impact on application performance when not tracing.
Chapter 2
Minimum requirements

This chapter describes the prerequisites for the host and target systems to run Graphics Analyzer.

It contains the following sections:

• 2.1 Host system requirements on page 2-14.
• 2.2 Target system requirements on page 2-15.
2.1 Host system requirements

Ensure that your host meets these requirements for running Graphics Analyzer.

- It can connect to the target using TCP/IP, for online analysis, and has port 5002 open.
- The installed application claims up to 48GB of RAM as it runs.

This section contains the following subsections:
- 2.1.1 Increase the available memory on page 2-14.
- 2.1.2 Temporary storage on page 2-14.

2.1.1 Increase the available memory

To increase the maximum trace size that Graphics Analyzer can accommodate, increase the amount of memory that is available to the application.

Procedure
1. Open <install_directory>/gui/aga.ini with a text editor.
2. Find the Java Virtual Machine (JVM) argument starting with -Xmx.
   The number that follows the argument defines the maximum amount of memory that the application claims when running, with a trailing m for megabytes.
3. Increase this number with a multiple of four that matches the capabilities of your system.
   Ensure that your modifications follow the same format for the argument, that is no spaces and a trailing lowercase m.

2.1.2 Temporary storage

Depending on the complexity of the application being traced, Graphics Analyzer can require a large amount of temporary disk storage.

By default, the system temporary storage directory is used. If there is not enough free space available in this directory, Graphics Analyzer displays a warning and stops caching data to the directory, increasing memory usage.

Change the temporary storage directory by clicking on Edit > Preferences and selecting an existing directory to be used in the Custom temporary storage directory field.
2.2 Target system requirements

Graphics Analyzer supports tracing the following Khronos Group APIs:

- Vulkan 1.0
- OpenGL ES 2.0, 3.0, 3.1, or 3.2
- OpenCL 1.0, 1.1, and 1.2

Some Graphics Analyzer features require a valid license, for instance to trace a Linux target system that is not Android-based. Refer to the relevant Studio documentation for information about license setup.
Chapter 3
Target installation

Graphics Analyzer has two target components that interact to collect and transmit trace information from your application to the GUI on the host. They are the interceptor library and the daemon application.

Interceptor library
Intercepts calls made by your application to one of the supported APIs and collects information about each call. The interceptor library must be loaded before your application starts. This library also doubles as a Vulkan layer, allowing Vulkan applications to be traced using the Vulkan API layers system.

Daemon application
Collects the trace data from all running applications being intercepted and sends this data to the host. All data is transferred using TCP/IP on port 5002.

The way that you use these components varies by target platform.

For Vulkan applications, the installation instructions in this chapter describe the simplest method of getting Graphics Analyzer running with your application using the layers system. However, the Vulkan layers system is flexible and there might be other valid methods for installing the Graphics Analyzer layer on your target platform that better suit your needs. For more details, see the Vulkan Loader Specification and Architecture Overview.

Arm does not recommend that the validation layers are used at the same time as the Graphics Analyzer layer. The validation layers are designed to validate a Vulkan application, that is to detect invalid use of the Vulkan API. Graphics Analyzer is designed to ensure that a Vulkan application that has already been validated does what it is expected to do.

Due to these different goals, using the Graphics Analyzer layer at the same time as the validation layers might result in unexpected behavior.
It contains the following sections:

- 3.1 Unrooted Android on page 3-18.
- 3.2 System-wide Android on page 3-26.
- 3.3 Connect to your Android device from the command line on page 3-33.
- 3.4 Linux on page 3-34.
- 3.5 Chrome OS on page 3-37.
- 3.6 webOS on page 3-40.
- 3.7 Troubleshooting on page 3-43.
3.1 Unrooted Android

Follow these instructions for installing Graphics Analyzer on an unrooted Android target.

This section contains the following subsections:

- 3.1.1 Prerequisites on page 3-18.
- 3.1.2 Limitations on page 3-18.
- 3.1.3 Installation on page 3-18.
- 3.1.4 Use the OpenGL ES, EGL, and OpenCL interceptor in a target application on page 3-19.
- 3.1.5 Use the OpenGL ES, EGL, and OpenCL interceptor in an Android 10 target application on page 3-20.
- 3.1.6 Use the Vulkan layer in a target application on page 3-20.
- 3.1.7 Build an OpenGL ES Unity application with Graphics Analyzer support on page 3-21.
- 3.1.8 Build an OpenGL ES Unreal Engine application with Graphics Analyzer support on page 3-25.
- 3.1.9 Uninstall Graphics Analyzer on page 3-25.

3.1.1 Prerequisites

To run Graphics Analyzer on an unrooted Android target, ensure that the host and target meet the following requirements:

- The Android Software Development Kit (SDK) must be installed on the host machine.
- The PATH environment variable on the host machine must include the path to the adb binary.
- You have a valid adb connection to the target device. In other words, adb devices returns the ID of your device with no permission errors and you can run adb shell without issues. See Android Studio User Guide for more information.
- The target device must permit TCP/IP communication on port 5002.
- The source code for the application to be traced must be accessible.
- The Android device must be running Android 7.0 or above.
- The host machine must have Python 3.5 or later installed. You can download it from python.org, or if you are using Linux, from your package repositories.

Note

The aga_me.py command-line script, which is a method for connecting Graphics Analyzer to your device, requires Python 3.5. See 3.3 Connect to your Android device from the command line on page 3-33 for more information.

3.1.2 Limitations

Installing Graphics Analyzer inside the target application rather than in the /system or /vendor directories on the device has some drawbacks:

- Source code access is required
- Tracing extensions might work but is unsupported

3.1.3 Installation

Installation on an unrooted device involves several steps.

Before launching the application to be traced, you must:

- For Android 9 and below, modify the source code and build script of the application, as described in the following sections.
- For Android 10, make the application debuggable.

To configure your target device and run your application, see 3.3 Connect to your Android device from the command line on page 3-33.
3.1.4 Use the OpenGL ES, EGL, and OpenCL interceptor in a target application

To use the interceptor in an Android application, you must package the interceptor into your application, alter the build scripts, then enable the interceptor.

--- Warning ---

Do not use this procedure on Android 10. For Android 10, use the aga_me.py script to install the interceptor as an OpenGL ES layer driver, which is loaded into your application at runtime. Follow the steps in 3.1.5 Use the OpenGL ES, EGL, and OpenCL interceptor in an Android 10 target application on page 3-20.

Prerequisites

Ensure that your build is debuggable in its Android manifest.

Procedure

1. Add the following code to the `build.gradle` file for your module:

   ```groovy
   android {
       sourceSets {
           main {
               jniLibs.srcDirs += ['<install_directory>/target/android/arm/unrooted/']
           }
       }
   }
   ```

   This code adds the Graphics Analyzer unrooted library folder to the list of locations that the Android build system searches for native libraries. Adding this folder ensures that the Graphics Analyzer interceptor is included in the APK for your application.

2. In the `AndroidManifest.xml` for your application, add the following attribute to the `<activity>` element, if it is not already present:

   ```xml
   android:debuggable="true"
   ```

   Because the interceptor for unrooted devices is only available for the `armeabi-v7a` and `arm64-v8a` ABIs, you must also specify an appropriate value for `abiFilters` in the `build.gradle` for your module. For example, to target both Armv7 and Armv8, use the following code:

   ```groovy
   android {
       defaultConfig {
           ndk {
               abiFilters 'armeabi-v7a', 'arm64-v8a'
           }
       }
   }
   ```

   For more information about Android ABIs, see https://developer.android.com/ndk/guides/abis.html.

3. If your application has Java components, to ensure that it loads Graphics Analyzer, add the following code to the beginning of the main Activity class in your project:

   ```java
   static {
     try {
       System.loadLibrary("AGA");
     } catch (UnsatisfiedLinkError e) {
       // Feel free to remove this log message.
       Log.e("[ GA ]", "GA not loaded: " + e.getMessage());
       Log.d("[ GA ]", Log.getStackTraceString(e));
     }
   }
   ```

Next Steps

Recompile the application and install it on your Android device.
Using the interceptor in a C or C++-only application

If your application is C or C++-only, you must add a small Java component to it.

Native applications on Android cannot load a library that depends on non-system libraries. Therefore, linking your native application against the Graphics Analyzer interceptor causes the application to fail at runtime. It fails because the application cannot load the interceptor, even if you have included it in your APK. The solution is to add a small Java component to your application.

Procedure

1. Create an Activity class that extends android.app.NativeActivity. Include the code that is given in 3.1.4 Use the OpenGL ES, EGL, and OpenCL interceptor in a target application on page 3-19.
2. Open your AndroidManifest.xml.
3. Locate the <activity> element and reference the new Activity as the android:name attribute.
4. Locate the <application> element and check that the android:hasCode attribute is true. If this attribute is false, the new Java files are excluded from the APK.

3.1.5 Use the OpenGL ES, EGL, and OpenCL interceptor in an Android 10 target application

Install the layer driver, configure the command-line daemon, and connect to your target device.

--- Note ---

- Arm recommends that you do not use the validation layers at the same time as the Graphics Analyzer layer. For more information, see Chapter 3 Target installation on page 3-16.

---

Prerequisites

Ensure that your build is debuggable in its Android manifest.

Procedure

1. Open a shell terminal and run the following from <install_directory>	arget\android\arm:

   ./aga_me.py com.<application.name>

   The script configures the target device and installs the layer. See https://developer.android.com/ndk/guides/graphics/validation-layer for more details.
2. Open Graphics Analyzer and click Open the Device Manager.
3. Under Connect to an IP, enter localhost as the IP address and port 5002.
4. Click Connect.

--- Note ---

Do not use the direct connection methods or the layer install methods in the table at the top of the Device Manager dialog box.

---

5. Press Enter on the command line.

   The aga_me.py script launches the application to trace.

Next Steps

When tracing is complete, press Enter on the command line. The aga_me.py script cleans up any files on the target device.

3.1.6 Use the Vulkan layer in a target application

Install the layer driver, configure the command-line daemon, and connect to your target device.
Note

- This procedure is not applicable to Android 8.
- Arm recommends that you do not use the validation layers at the same time as the Graphics Analyzer layer. For more information, see Chapter 3 Target installation on page 3-16.

Prerequisites
Ensure that your build is debuggable in its Android manifest.

Procedure
1. Open a shell terminal and run the following from <install_directory>/target/android/arm:

   ```
   ./aga_me.py -v com.<application.name>
   ```

   The script configures the target device and installs the layer. The flag -v instructs the script to install the Vulkan layer. See https://developer.android.com/ndk/guides/graphics/validation-layer for more details.

2. Open Graphics Analyzer and click Open the Device Manager.

3. Under Connect to an IP, enter localhost as the IP address and port 5002.

4. Click Connect.

   Note

   Do not use the direct connection methods or the layer install methods in the table at the top of the Device Manager dialog box.

5. Press Enter on the command line.

   The aga_me.py script launches the application to trace. Alternatively, run your application manually.

Next Steps
When tracing is complete, press Enter on the command line. The aga_me.py script cleans up any files on the target device.

3.1.7 Build an OpenGL ES Unity application with Graphics Analyzer support

To enable Graphics Analyzer to trace your application, add the Arm interceptor library to your Unity project. The interceptor library intercepts and collects information about the OpenGL ES and EGL function calls made by the application. Unity detects the presence of the interceptor library and automatically loads it when you build the application.

The library libMGD.so is provided in your Arm Mobile Studio package:

```
<install_directory>/graphics_analyzer/target/android/arm/unrooted/
```

Two versions of the library are provided:

- For 64-bit applications, use the library file that is located in the arm64-v8a directory.
- For 32-bit applications, use the library file that is located in the armeabi-v7a directory.

Note
You can package one or both interceptor libraries depending on the requirements of your application.

To add the library to Unity and set the player and build settings to use it, follow these steps. For more detailed instructions, see Arm guide for Unity developers.
Prerequisites

Check that you are using the Unity recommended Android SDK and Android NDK versions in the Unity editor.

- On Windows and Linux, select Edit > Preferences > External Tools.
- On macOS, select Unity > Preferences > External Tools.

If the checkboxes are selected, they are installed. Otherwise, you can add them as modules. See https://docs.unity3d.com/Manual/android-sdksetup.html for more information.

Procedure

1. Copy the required interceptor library file libMGD.so into your Unity project, in the Assets/Plugins/Android/ directory. Create this directory if it does not exist.

If you are packaging both interceptor libraries:

1. Create two directories in the Assets/Plugins/ directory. For example, armv7 and armv8.
2. Create the Android directory within each of these directories.
3. Copy each libMGD.so file into the appropriate Android directory.

2. In Unity, select the library in the Project window, and set the following attributes in the Inspector:

1. Under Select platforms for plugin, select Android.
2. Under Platform settings, set the CPU architecture to ARM64 for 64-bit applications, or ARMv7 for 32-bit applications.
3. Select **File > Build Settings** and select **Player Settings**.

4. Under **Identification**, set **Target API Level** to the required Android version.

    ________ Note _________

    By default, **Target API Level** is set to the latest version of the Android SDK tools that you have installed. If you change to a lower API level, ensure that you have the SDK tools for that version installed, and be aware that if you want to build for a higher API version later, you will need to change this setting accordingly.

    ________

5. Under **Configuration**, set the following options.

    - To build a 64-bit application:
      1. Set the scripting backend in Unity to work with 64-bit targets. Set **Scripting Backend** to **IL2CPP**. For more information about IL2CPP, refer to the Unity documentation.
      2. Under **Target Architectures**, select **ARM64**.
To build a 32-bit application:
1. Leave the **Scripting Backend** at its default setting, **Mono**.
2. Under **Target Architectures**, select **ARM7**.

6. Close the **Player Settings**. In the **Build Settings**, select the **Development Build** checkbox. This option ensures that your application is marked as debuggable in the Android application manifest.
7. Select **Build and Run** to build your APK and install it on your device in one step. Alternatively, select **Build** to build the APK and then install it on your device using Android Debug Bridge:

```
adb install -r YourApplication.apk
```

**Next Steps**

To trace the Unity application, follow the instructions in [3.3 Connect to your Android device from the command line](#) on page 3-33.

### 3.1.8 Build an OpenGL ES Unreal Engine application with Graphics Analyzer support

You can install and use the Graphics Analyzer with help from Unreal Engine on an unrooted Android device.

In version 4.15 and later, Unreal Engine supports packaging the Graphics Analyzer interceptor into your application from the package settings. You can read a detailed set of instructions with screenshots on the [Arm Community website](#).

**Note**
The instructions refer to Mali Graphics Debugger but they apply to Graphics Analyzer.

The following list is a summary of the instructions:

**Procedure**

1. Open your Unreal Engine project.
2. Open the **Project Settings** window.
3. From the left menu, select the **Android** option, then **Platforms**, then select Graphics Analyzer in the **Graphic Debugger** list.
4. Enter the path to your Graphics Analyzer installation.

**Next Steps**

After the Unreal Engine application has installed, follow the instructions in [3.3 Connect to your Android device from the command line](#) on page 3-33 to connect to the device and trace the application.

### 3.1.9 Uninstall Graphics Analyzer

Uninstall the Graphics Analyzer daemon application in the same way as any other Android application.

**Procedure**

1. Open the menu
2. Press and hold on the application icon
3. Drag it to the uninstall (trash bin) icon on-screen
4. To exclude `libAGA.so` from the target application build, comment out or delete the code that you added in the installation instructions, see [3.1.3 Installation](#) on page 3-18.
3.2 System-wide Android

Intercept any application on the target device, including applications that are not debuggable.

System-wide Android support requires a valid license. Refer to the relevant Studio documentation for information about license setup.

This section contains the following subsections:

- 3.2.1 Prerequisites on page 3-26.
- 3.2.2 Installation on page 3-26.
- 3.2.3 Install the daemon on page 3-27.
- 3.2.4 Install the OpenGL ES, EGL, and OpenCL interceptor on a 32-bit device on page 3-27.
- 3.2.5 Install the OpenGL ES, EGL, and OpenCL interceptor on a 64-bit device on page 3-28.
- 3.2.6 Install the Vulkan layer on a 32-bit device on page 3-29.
- 3.2.7 Install the Vulkan layer on a 64-bit device on page 3-30.
- 3.2.8 Choose which applications or processes to trace on page 3-31.
- 3.2.9 Uninstall Graphics Analyzer on page 3-32.

3.2.1 Prerequisites

To gather system-wide information on an Android target, ensure that the host and target meet the following requirements:

- The Android Software Development Kit (SDK) must be installed on the host machine.
- The \texttt{PATH} environment variable on the host machine must include the path to the Android Debug Bridge (adb) binary.

\textbf{Note}

\texttt{adb} is available free of charge as part of the Android SDK. For more information, see the \textit{Android Debug Bridge User Guide}.

- You have a valid \texttt{adb} connection to the target device. In other words, \texttt{adb devices} returns the ID of your device with no permission errors and you can run \texttt{adb shell} without issues. See \textit{Android Studio User Guide} for more information.
- The target device must allow modification of \texttt{/system} and, on many devices, \texttt{/vendor}. Generally, the way to achieve this is to root the device. The way to do this varies from manufacturer to manufacturer and is beyond the scope of this document. If rooting is not possible, Graphics Analyzer provides an alternative, but it has limitations, and impacts application startup performance. To install Graphics Analyzer on an unrooted device, see \textit{3.1 Unrooted Android on page 3-18}.
- Graphics Analyzer allows you to connect to devices using TCP/IP. If you intend to use this function, the target device should allow the use of port 5002.
- For Android targets you need at least 11MB free in \texttt{/vendor}, and an additional 14MB if you are also installing the 64-bit interceptor. On older Android devices, the interceptor is installed into \texttt{/system}.

To check free space in the filesystems mounted at \texttt{/vendor} and \texttt{/system}, run the following commands from a host command line:

\begin{verbatim}
adb shell df /vendor
adb shell df /system
\end{verbatim}

3.2.2 Installation

Graphics Analyzer does not require you to make changes to traced applications on rooted Android devices.
Arm recommends that you use the graphical Device Manager to install the relevant Graphics Analyzer components on your rooted device. For details, see 4.1 Android devices on page 4-46. However, this section provides instructions for installing these components manually.

--- Note ---

If you are tracing an Android application targeted at SDK level 26 or later, it might be necessary to enter one of the following special commands at a root shell prompt each time you reset your Android device:

- `setenforce 0`
- `supolicy --live 'permissive untrusted_app'`

See 3.7.3 Socket permission errors on page 3-43 for more details.

### 3.2.3 Install the daemon

Graphics Analyzer uses a daemon application to collate information from different traced applications. This topic describes the process for system-level installation of the command-line version of this daemon, called `aga-daemon`.

--- Note ---

Arm recommends that you use a command-line script called `aga_me.py`. For details, see 3.3 Connect to your Android device from the command line on page 3-33. This script sets up the daemon on a device whenever it is required, bypassing the need for system installation.

--- Procedure ---

1. Open a shell on your host machine.
2. Navigate to the Graphics Analyzer installation directory.
3. Copy the daemon to your target device.
   - If your device is Arm-based, use this command:
     `adb push target/android/arm/aga-daemon /sdcard`
   - If you are using an x86-based device, use this command:
     `adb push target/android/intel/aga-daemon /sdcard`
4. Open a shell on your target device as the super user by running:
   ```bash
   adb shell
   su
   ```
5. On some devices, `/system` is a mount point for a filesystem. To make sure that this filesystem is writable, run:
   ```bash
   mount -o rw,remount /system
   ```
   On other devices, `/system` is not a filesystem and this command fails with an error which you can safely ignore.
6. Install the daemon on the device, and grant appropriate permissions:
   ```bash
   cp /sdcard/aga-daemon /system/bin/aga-daemon
   chmod 777 /system/bin/aga-daemon
   ```

### 3.2.4 Install the OpenGL ES, EGL, and OpenCL interceptor on a 32-bit device

Install the interceptor that collects information about OpenGL ES, EGL, and OpenCL function calls from your application on a 32-bit Android device.

--- Note ---

In 2017, Google introduced a new driver interface as part of an initiative called Project Treble. If you try to install the interceptor into the `/system` directory on a device with Treble enabled, your device might
become unbootable. The device drivers have been moved to the /vendor directory, but otherwise the structure is the same.

Arm recommends that you use the Device Manager to automate the installation, if possible. For more information, see 4.1 Android devices on page 4-46. Otherwise, follow these instructions, but you must install the interceptor into the /vendor directory instead of the /system directory.

You can check whether your Android device is Treble-enabled using this command:

```
adb shell getprop ro.treble.enabled
```

If it returns `true`, your device is Treble-enabled.

---

**Prerequisites**

Your Android device must have at least 11MB of free space in /system.

**Procedure**

1. Open a terminal on your host machine, and navigate to `<install_directory>/target/android/`.
2. Copy the 32-bit interceptor onto your target device.
   • If you are using an Arm-based device, run the following command in your terminal:
     ```
     adb push arm/rooted/armmeabi-v7a/libGLES_aga.so /sdcard/
     ```
   • If you are using an x86-based device, run the following command in your terminal:
     ```
     adb push intel/rooted/x86/libGLES_aga.so /sdcard/
     ```
3. Run the following commands:

   ```
   adb shell
   su
   mount -o rw,remount /system
   cp /sdcard/libGLES_aga.so /system/lib/egl/libGLES_aga.so
   chmod 777 /system/lib/egl/libGLES_aga.so
   ln /system/lib/egl/libGLES_aga.so /system/lib/egl/libGLES.so
   ln /system/lib/egl/libGLES_aga.so /system/lib/egl/libGLES
   ```

   These commands do the following:
   • Open a shell on your device as the super user.
   • Ensure that you can write to the filesystem that is mounted at /system.
   • Install the interceptor.
   • Grant appropriate permissions to the interceptor.
   • Name the interceptor library /system/lib/egl/libGLES.so so that Android loads the Graphics Analyzer interceptor library before the system library.

   **Note**
   A bug in the Android 8.0 driver loader meant that the interceptor library must be named /system/lib/egl/libGLES, with no file extension, on Android 8.0 devices. Usually, no library exists with either of these names in the system, so you can run the command with and without the file extension to ensure that the interceptor library has the correct name for the Android version on the target.

4. Reboot the target device.

   `libGLES_aga.so` intercepts all OpenGL ES, EGL, and OpenCL function calls in any open applications.

---

**3.2.5 Install the OpenGL ES, EGL, and OpenCL interceptor on a 64-bit device**

To install the interceptor on a 64-bit Android device, follow these instructions.

On 64-bit Android devices, you must install two copies of the interceptor library:
• One for tracing 32-bit applications
• One for tracing 64-bit applications

To install the 32-bit interceptor, follow the instructions in 3.2.4 Install the OpenGL ES, EGL, and OpenCL interceptor on a 32-bit device on page 3-27.

Note
The same warning about Project Treble devices applies to the 64-bit interceptor. If you try to install the interceptor into the /system directory on a device with Treble enabled, your device might become unbootable. The device drivers have been moved to the /vendor directory, but otherwise the structure is the same.

We recommend that you use the Android Device Manager to automate the installation, if possible. For more information, see 4.1 Android devices on page 4-46. Otherwise, follow these instructions, but you must install the interceptor into the /vendor directory instead of the /system directory.

Next, install the 64-bit interceptor in the same way as the 32-bit interceptor, except copy it into /system/lib64/egl/ instead of /system/lib/egl/. Use the following steps:

Procedure
1. Open a terminal on your host device, and navigate to <install_directory>/target/android.
2. Copy the 64-bit interceptor onto your target device.
   • If you are using an Arm-based Mali Android device, run:
     ```bash
     adb push arm/rooted/arm64-v8a/libGLES_aga.so /sdcard/
     ```
   • Otherwise, if you are using an x86-based Mali Android device, run:
     ```bash
     adb push intel/rooted/x86-64/libGLES_aga.so /sdcard/
     ```
3. Run the following commands to install the 64-bit interceptor:
   ```bash
   adb shell
   su
   mount -o rw,remount /system
   cp /sdcard/libGLES_aga.so /system/lib64/egl/libGLES_aga.so
   chmod 777 /system/lib64/egl/libGLES_aga.so
   ln -s /system/lib64/egl/libGLES_aga.so /system/lib64/egl/libGLES
   ln -s /system/lib64/egl/libGLES_aga.so /system/lib64/egl/libGLES
   ```

There should be two copies of the interceptor:
• A 32-bit version in /system/lib/egl
• A 64-bit version in /system/lib64/egl

After you have rebooted the device, libGLES_aga.so intercepts all OpenGL ES, EGL, and OpenCL function calls in any opened applications.

3.2.6 Install the Vulkan layer on a 32-bit device

Follow these instructions to install the Vulkan layer onto your device, which enables you to trace all Vulkan applications.

Note
• Alternatively, you can package the Vulkan layers into just your Android application. For details, see 3.1.6 Use the Vulkan layer in a target application on page 3-20.
• We do not recommend that the Vulkan validation layers are used at the same time as the Graphics Analyzer Vulkan layer. For more information, see Chapter 3 Target installation on page 3-16.

Procedure
1. Open a terminal on your host device, and navigate to <install_directory>/target/android/.
2. Copy the 32-bit interceptor onto your target device, and rename it to `libVkLayerAGA.so`. If you are using an Arm-based Mali Android device, run:

   `adb push arm/rooted/armeabi-v7a/libGLES_aga.so /sdcard/libVkLayerAGA.so`

   Otherwise, if you are using an x86-based Mali Android device, run:

   `adb push intel/rooted/x86/libGLES_aga.so /sdcard/libVkLayerAGA.so`

3. Open a shell as a super user on your target device by running:

   `adb shell su`

4. On some Android devices, `/system` is a mount point for a filesystem. To make sure that this filesystem is writable, run:

   `mount -o rw,remount /system`

   On other devices, `/system` is not a filesystem and this command fails with an error, which you can safely ignore.

5. In Android 7 and later, Vulkan tries to load layers from the `/system/fake-libs/` folder. Install the layer on the device by running:

   `cp /sdcard/libVkLayerAGA.so /system/fake-libs/`

   On older versions of Android, install the layer by running:

   `mkdir -p /data/local/debug/vulkan/
cp /sdcard/libVkLayerAGA.so /data/local/debug/vulkan/`

6. You must grant appropriate permissions to the layer and all of its parent folders. On Android 7, run:

   `chmod 777 /system/fake-libs/libVkLayerAGA.so`

   On older versions of Android, run:

   `chmod 777 /data/local/debug
   chmod 777 /data/local/debug/vulkan
   chmod 777 /data/local/debug/vulkan/libVkLayerAGA.so`

### 3.2.7 Install the Vulkan layer on a 64-bit device

To install the Vulkan layer onto your 64-bit Android device, follow these instructions.

On 64-bit Android devices, you must install two copies of the Vulkan layer:

- One for tracing 32-bit applications
- One for tracing 64-bit applications

To install the 32-bit Vulkan layer, follow the instructions in 3.2.6 Install the Vulkan layer on a 32-bit device on page 3-29.

--- Note ---

For Android, the names of the Vulkan layers do not matter. The Vulkan loader enumerates all libraries that it finds in `/system/fake-libs/`, `/system/fake-libs64/`, or `/data/local/debug/vulkan/`, regardless of the file name. It ignores any libraries that it cannot load, such as those libraries with a different bitness to the Vulkan application.

---

For Android 7 and later:

1. Open a terminal on your host device, and navigate to `<install_directory>/target/android/`.
2. Copy the 64-bit interceptor onto your target device, and rename it to `libVkLayerAGA.so`. If you are using an Arm-based device, run:

   `adb push arm/rooted/arm64-v8a/libGLES_aga.so /sdcard/libVkLayerAGA.so`
Otherwise, if you are using an x86-based device, run:

```
adb push intel/rooted/x86-64/libGLES_aga.so /sdcard/libVkLayerAGA.so
```

3. To install the layer, execute these commands:

```
adb shell
su
mount -o remount /system
cp /sdcard/libVkLayerAGA.so /system/fake-libs64/
chmod 777 /system/fake-libs64/libVkLayerAGA.so
```

For older versions of Android:

1. Open a terminal on your host device, and navigate to `<install_directory>/target/android/`.
2. Copy the 64-bit interceptor onto your target device, and rename it to `libVkLayerAGA64.so`. If you are using an Arm-based device, run:

```
adb push arm/rooted/arm64-v8a/libGLES_aga.so /sdcard/libVkLayerAGA64.so
```

Otherwise, if you are using an x86-based device, run:

```
adb push intel/rooted/x86-64/libGLES_aga.so /sdcard/libVkLayerAGA64.so
```

3. To install the layer, execute these commands:

```
adb shell
su
mount -o remount /system
mkdir -p /data/local/debug/vulkan/
cp /sdcard/libVkLayerAGA64.so /data/local/debug/vulkan/
chmod 777 /data/local/debug/vulkan
chmod 777 /data/local/debug/vulkan/libVkLayerAGA64.so
```

As a result, you have two copies of the Vulkan layer:

- The 32-bit `libVkLayerAGA.so`
- The 64-bit `libVkLayerAGA64.so`

### 3.2.8 Choose which applications or processes to trace

By default Graphics Analyzer traces all running OpenGL ES, EGL, OpenCL, and Vulkan applications. To limit Graphics Analyzer to only trace specific applications, follow these instructions.

Include the names of the applications to trace in `/system/lib/egl/processlist.cfg`. This file can contain more than one process name, separated by newline characters.

_______ Note _______

You can set the appropriate permissions for `processlist.cfg` by running this command on the target device:

```
chmod 666 /system/lib/egl/processlist.cfg
```

For convenience, the interceptor prints the names of any processes that link against OpenGL ES, EGL, OpenCL, or Vulkan to `logcat`, which can help you to find the process names. For example, `logcat` might contain this line:

```
```

In this case, your `processlist.cfg` file should contain the single line:

```
com.arm.mali.Timbuktu
```

_______ Note _______

If `/system/lib/egl/processlist.cfg` exists but is empty, then no applications are traced.
3.2.9 Uninstall Graphics Analyzer

To uninstall Graphics Analyzer, follow these instructions.

**Procedure**

1. Open a terminal on your host device, and open a shell as a super user on your target device by running:
   ```
   adb shell
   su
   ```

2. Graphics Analyzer might have been installed into /vendor and /system. To ensure that filesystems mounted at these locations are writable, run:
   ```
   mount -o rw,remount /vendor
   mount -o rw,remount /system
   ```
   These commands fail if your device does not have filesystems mounted at these locations.

3. Remove the daemon by running:
   ```
   rm /system/bin/aga-daemon
   ```

4. If you installed the 32-bit OpenGL ES, EGL, and OpenCL interceptor:
   a. Remove the interceptor by running:
      ```
      rm /system/lib/egl/libGLES_aga.so
      ```
   b. If you created the /system/lib/egl/libGLES.so symlink, remove it by running:
      ```
      rm /system/lib/egl/libGLES.so
      ```
   c. If you created symlinks for a non-monolithic system, remove them by running:
      ```
      rm /system/lib/egl/libEGL_aga.so
      rm /system/lib/egl/libGLESv1_CM_aga.so
      rm /system/lib/egl/libGLESv2_aga.so
      ```

5. If you installed the 64-bit OpenGL ES, EGL, and OpenCL interceptor:
   a. Remove the interceptor by running:
      ```
      rm /system/lib64/egl/libGLES_aga.so
      ```
   b. If you created the /system/lib64/egl/libGLES.so symlink, remove it by running:
      ```
      rm /system/lib64/egl/libGLES.so
      ```

6. If you installed the 32-bit Vulkan layer, remove it by running:
   ```
   rm /data/local/debug/vulkan/libVkLayerAGA.so
   ```

7. If you installed the 64-bit Vulkan layer, remove it by running:
   ```
   rm /data/local/debug/vulkan/libVkLayerAGA64.so
   ```

8. Reboot the target device to allow the original libraries to be reloaded.
3.3 Connect to your Android device from the command line

The `aga_me.py` command-line tool configures an Android device so that you can trace specific applications.

**Prerequisites**

- Link the interceptor into the application using the instructions in 3.1.4 Use the OpenGL ES, EGL, and OpenCL interceptor in a target application on page 3-19 and make your application debuggable.
- Download Python 3.5 or later from python.org, or if you are using Linux, from your package repositories.

  The tool works best when Python is in your shell `PATH`. Linux and macOS systems are likely to be already configured in this way. On Windows, the Python installer has options to change your shell `PATH` and to associate `.py` files with Python. Select both options or set them after Python has been installed.
- Ensure that a working `adb` is available in your shell.

**Procedure**

1. Open a command prompt.
2. Navigate to your Graphics Analyzer installation directory.
3. If your device is Arm-based, pass the application name to the tool.
   
   For example:
   ```bash
   android/arm/aga_me.py Timbuktu
   ```
   Otherwise, use this command:
   ```bash
   android/intel/aga_me.py Timbuktu
   ```

   ________ Note _________
   
   It is not necessary to supply the full application name, only text that uniquely identifies the application. In this example, we assume that there is only one application name containing the word `Timbuktu`. If that was not the case, the script would have stopped after listing all matching applications.

   ________

4. You are prompted to use Device Manager to connect the host GUI application to IP address 127.0.0.1 and port 5002. See 4.3 Connect to an IP address on page 4-50 for more details.
5. If you want the tool to start the application:
   a. Press Enter.
   b. Observe tracing information in the Graphics Analyzer host GUI application.
   c. When you have finished or the traced application has terminated, press Enter again to terminate both `aga_me.py` and the traced application, if it is still running.

      Alternatively, start the traced application yourself, perhaps from the menu system on your device, then:
      a. On application launch, observe tracing information in the Graphics Analyzer host GUI application.
      b. When you have finished or the traced application has terminated, press Ctrl+C to end `aga_me.py`.
      c. If necessary, terminate the traced application yourself.
6. For a list of traceable applications on the device, run:
   ```bash
   agra_me.py --list_packages
   ```

   Not all applications are traceable. In particular, applications that are not debuggable are also not traceable.
3.4 Linux

Follow these instructions for installing and using Graphics Analyzer on a Linux target.

Note

Linux support requires a valid license.

This section contains the following subsections:

- 3.4.1 Prerequisites on page 3-34.
- 3.4.2 Install Graphics Analyzer on a Linux target on page 3-34.
- 3.4.3 Connect the host and the target on page 3-35.
- 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application on page 3-35.
- 3.4.5 Trace a Vulkan application on page 3-35.
- 3.4.6 Uninstall Graphics Analyzer on page 3-36.

3.4.1 Prerequisites

To run Graphics Analyzer on a Linux target, ensure that the target has the following:

- A running OpenGL ES, EGL, OpenCL, or Vulkan application
- A network connection to a host running the Graphics Analyzer GUI
- The target must permit TCP/IP communication on port 5002

3.4.2 Install Graphics Analyzer on a Linux target

Take the following steps to install Graphics Analyzer on a Linux target.

1. Navigate to `<install_directory>/target/linux` and then to the `soft_float`, `hard_float`, or `arm64` directory, according to the configuration of your system.

   Inside each of these directories, there are the following files:
   - `libinterceptor.so`
   - `aga-daemon`

   Note

   - The Linux interceptor only supports Armv7 and Armv8 target architectures.
   - Make sure that you use the correct libraries for your target architecture. If you are running on Armv7, you must use either the soft float libraries or the hard float libraries, depending on the requirements of your system.
   - If you are running on Armv8 (64-bit) and intend to trace a 64-bit application, you must use the Armv8 libraries.
   - If you are running on Armv8 but intend to trace a 32-bit application, you must use the appropriate Armv7 libraries.
   - You can use the 64-bit build of the daemon when tracing both 64-bit and 32-bit applications.

2. Install the daemon by copying `aga-daemon` to anywhere on your target device, and setting the execute permission bit on the file. You can do this by running the following command from inside the directory into which `aga-daemon` was copied:

   ```
   chmod +x aga-daemon
   ```

3. Install the OpenGL ES, EGL, and OpenCL interceptor by copying `libinterceptor.so` to anywhere on your target device.

4. Install the Vulkan layer:
a. The Graphics Analyzer interceptor doubles as a Vulkan layer. To use it in this way, copy `libinterceptor.so` to anywhere on your target device, and rename it to `libVkLayerAGA.so`.

--- Note ---

Graphics Analyzer supports tracing Vulkan applications on all Linux targets except for Arm soft-float.

--- Note ---

b. `<install_directory>/target/linux` contains the `VK_LAYER_ARM_AGA.json` manifest file. Copy this file into the same directory as `libVkLayerAGA.so`.

--- Note ---

Manifests allow the Vulkan loader to identify the names and attributes of layers and extensions without needing to load them, which might be expensive and unnecessary if the application does not query or request them.

### 3.4.3 Connect the host and the target

To connect Graphics Analyzer running on your host device to the target device you want to trace on, the daemon application must be running on the target device.

1. To start the daemon, open a terminal, navigate to the directory you copied `aga-daemon` into on your target, and run:

   ```
   ./aga-daemon
   ```

   The daemon can handle multiple applications starting and stopping, and you should only close it when you have finished tracing all the applications.

2. Connect to `aga-daemon` running on the device using the Device Manager, see 4.2 Linux devices on page 4-49. If the Device Manager detects a running instance of `aga-daemon` on the local network, it gives you the option to connect to it. Otherwise, you can use the Device Manager to directly connect to the IP address of the device, see 4.3 Connect to an IP address on page 4-50.

3. Start the application that you want to trace by following the instructions in 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application on page 3-35, or in 3.4.5 Trace a Vulkan application on page 3-35.

### 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application

To trace an OpenGL ES, EGL, or OpenCL application, the system must preload the `libinterceptor.so` library that you copied onto your target.

To do this, define the `LD_PRELOAD` environment variable to point at this library. For example:

```
LD_PRELOAD=/path/to/intercept/libinterceptor.so ./your_app
```

If you are unable to use `LD_PRELOAD` on your system, see 8.9 Intercepting without using `LD_PRELOAD` on page 8-135 for an alternative method.

If you have more than one version of your graphics driver on your system and are having issues, see 8.10 Multiple drivers installed on the system on page 8-136 for more information.

### 3.4.5 Trace a Vulkan application

To trace your Vulkan application, you must tell the Vulkan loader the location of the Graphics Analyzer layer and manifest that you copied onto your target, and the name of the Graphics Analyzer layer to load as both an instance and a device layer.

For example:

```
VK_LAYER_PATH=/path/to/aga/layer/ VK_INSTANCE_LAYERS=VK_LAYER_ARM_AGA
VK_DEVICE_LAYERS=VK_LAYER_ARM_AGA ./your_vulkan_app
```
Note

The concept of *device layers* has been deprecated. However, some older drivers might still require the `VKDEVICE_LAYERS` environment variable to be set to allow Graphics Analyzer to trace all Vulkan function calls in the application.

3.4.6 Uninstall Graphics Analyzer

To uninstall Graphics Analyzer, remove the files that were copied onto the target platform.
3.5 Chrome OS

Follow these instructions for using Graphics Analyzer to trace an application running on a Chrome OS device.

This section contains the following subsections:

- 3.5.1 Prerequisites on page 3-37.
- 3.5.2 Trace an Android application on Chrome OS on page 3-37.
- 3.5.3 Trace a Linux application on Chrome OS on page 3-37.
- 3.5.4 Trace the Chrome application on Chrome OS on page 3-38.

3.5.1 Prerequisites

Graphics Analyzer supports tracing the following types of application running on Chrome OS devices:

- Android applications on Chrome OS devices that support the App Runtime for Chrome (ARC)
- Linux applications
- The Chrome application itself, and any Chrome apps that are running within Chrome OS

To debug your Chrome OS device, first enter Developer Mode. The next steps depend on the type of application.

Note

To debug Chrome or Linux applications, enable debugging features when you boot into Developer Mode.

3.5.2 Trace an Android application on Chrome OS

The App Runtime for Chrome (ARC) allows you to run Android applications on your Chrome OS device. If your device supports the ARC, then you can trace your application in Graphics Analyzer.

Note

There is no equivalent to the system-wide Android installation on Chrome OS. See 3.1 Unrooted Android on page 3-18 for further installation information.

To trace an Android application on Chrome OS:

1. Connect to your device over a network using adb. You can find the IP address of the target device in the Chrome OS WiFi menu, and the default port to use is 22. For example:

   ```shell
   adb connect (IP address):22
   ```

   For more information about connecting to a Chrome OS device over adb, see The development environment.

2. To connect to the device and trace the application, follow the instructions in 3.3 Connect to your Android device from the command line on page 3-33.

3.5.3 Trace a Linux application on Chrome OS

Use the Linux interceptor and Graphics Analyzer daemon to trace a Linux application running on a Chrome OS device.

Note

You must set up SSH access to your Chrome OS device before you can trace native Linux applications with Graphics Analyzer. For details, see Setting up SSH Access to your test device.
1. SSH into the Chrome OS device as root using the command:

   ```
   ssh root@(IP address)
   ```

2. Create a directory to store the Graphics Analyzer daemon and interceptor library, for example:

   ```
   mkdir /usr/bin/aga
   ```

3. To allow connections on port 5002, run the following command:

   ```
   sudo iptables -A INPUT -p tcp -m tcp --dport 5002 -j ACCEPT
   ```

4. Copy the Graphics Analyzer daemon and interceptor library onto your device. Depending on your device, use a version of the Linux Graphics Analyzer components appropriate to your architecture, either hard float, soft float, or 64-bit. You might need root access to copy files onto the Chrome OS file system. For example:

   ```
   scp libinterceptor.so root@(IP address):/usr/bin/aga/
   ```

5. From your root user SSH session, launch `aga-daemon`.

6. Using the Device Manager, connect to the running daemon by following the instructions in 4.2 Linux devices on page 4-49 or 4.3 Connect to an IP address on page 4-50.

7. Launch a new SSH session.

8. Run your Linux application while preloading the interceptor library. For instructions, see 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application on page 3-35.

As a result, trace data starts appearing in the desktop Graphics Analyzer client.

### 3.5.4 Trace the Chrome application on Chrome OS

Graphics Analyzer supports tracing the Chrome application in Chrome OS, which is useful for debugging websites, web apps, and Chrome applications.

The method is similar to tracing a Linux application on Chrome OS, with a few differences.

**Note**

- You must set up SSH access to your Chrome OS device before you can trace Chrome with Graphics Analyzer. For details, see Setting up SSH Access to your test device.

- Chrome OS tries to reboot when you attempt to stop the UI. To prevent this, you must modify the file `/usr/share/cros/init/ui-post-stop` by commenting out the following lines:

  ```
  while ! sudo -u chronos kill -9 -- -1 ; do
  sleep .1
  done
  # Check for still-living chronos processes and log their status.
  ps -u chronos --no-headers -o pid,stat,args |
  logger -i -t "${JOB}-unkillable" -p crit
  ```

After you have SSH access and the ability to stop the UI, install Graphics Analyzer on your device:

1. Follow the Graphics Analyzer daemon and interceptor installation instructions in 3.5.3 Trace a Linux application on Chrome OS on page 3-37.

2. Set up a password for the chronos user by starting an SSH session as root and using:

   ```
   passwd chronos
   ```

3. Start a new SSH session, this time using the chronos user:

   ```
   ssh chronos@(IP address)
   ```

4. From your root user SSH session, launch `aga-daemon`. You might need to restart the root session after starting an SSH session as chronos.

5. Connect to your Chrome OS device from Graphics Analyzer.
6. From your chronos SSH session, suspend the Chrome OS UI using the command:

```bash
sudo stop ui
```

7. You must preload the interceptor library and launch Chrome from the chronos user SSH session. For example:

```bash
LD_PRELOAD=/usr/bin/aga/libinterceptor.so /opt/google/chrome/chrome \
--ozone-platform=gbm --ozone-use-surfaceless \ 
--user-data-dir=/home/chronos/ --bwsi \ 
--login-user='$guest' --login-profile=user
```

For more information about preloading the interceptor library, see 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application on page 3-35.

As a result, trace data starts appearing in the desktop Graphics Analyzer client.

**Note**

On some devices, Chrome might not launch correctly while the desktop Graphics Analyzer client is connected, and might launch numerous subprocesses. If this happens, disconnect the Graphics Analyzer client and Chrome should launch correctly. You should be able to connect Graphics Analyzer and trace Chrome after it has launched.
3.6 webOS

Follow these instructions for installing and using Graphics Analyzer to trace different types of webOS applications.

This section contains the following subsections:

- 3.6.1 Application support on page 3-40.
- 3.6.2 Install Graphics Analyzer on webOS on page 3-40.
- 3.6.3 Trace a web-based application on page 3-41.
- 3.6.4 Trace a QML application on page 3-41.
- 3.6.5 Trace a native application on page 3-42.

3.6.1 Application support

Graphics Analyzer can trace all types of webOS applications, but different approaches are required for each.

The application types are:

- Web-based applications
- QML-based applications
- Native applications

It is possible to trace one application type without tracing the others.

--- Note ---

For each installed application, you can find the application type, internal name, installation location, and main executable by examining the output from this command:

```
luna-send -n 1 -f luna://com.webos.service.applicationManager/listApps "{}"
```

3.6.2 Install Graphics Analyzer on webOS

webOS devices are based on Linux and can use Graphics Analyzer executables that are intended for Linux.

Ensure that you use binaries that are compiled for the architecture of the target device, see 1.1 Installation package on page 1-11.

The installation steps are as follows:

1. Make a directory on the device named `/opt/graphics_analyzer/`.
2. Copy the following files into this new directory:
   - The Graphics Analyzer interceptor, `libinterceptor.so`
   - The Graphics Analyzer daemon process, `aga-daemon`
3. Create a script named `/opt/graphics_analyzer/aga-wrapper`. This script applies the Graphics Analyzer interceptor to arbitrary applications. Populate the script as follows:

   ```
   #!/bin/sh
   MGD_LIBRARY_PATH=/usr/lib
   LD_PRELOAD=/usr/lib/libcbe.so:
   `dirname $0`/ga/libinterceptor.so:
   $LD_PRELOAD
   export MGD_LIBRARY_PATH LD_PRELOAD
   exec $0.bin "$@"
   ```
4. Create a script named `/etc/init/graphics_analyzer.conf`. This script ensures that the Graphics Analyzer daemon launches at device boot. Populate the script as follows:

   ```
   description "Launch the Graphics Analyzer daemon from Arm Ltd."
   start on started sam
   respawn
   script
   ```
exec /opt/graphics_analyzer/aga-daemon > /var/log/aga-daemon.log 2>&1
end script

--- Note ---
This script directs messages from the Graphics Analyzer daemon to /var/log/aga-daemon.log.

5. Edit the file `/etc/luna-service2/ls-hubd.conf` as follows:
   - Locate the `[Security]` section.
   - Change the `Enabled` key from `true` to `false` and save it.

   This step allows you to change executables on the device without a security fault being issued.

### 3.6.3 Trace a web-based application

This method involves loading the Graphics Analyzer interceptor into the Web Application Manager (WAM), which is the process responsible for displaying a web-based application.

The steps are as follows:
1. Open `/etc/init/WebAppMgr.conf`
2. Near the end of this file is a line beginning `exec $WEBOS_NICE $WAM_EXE_PATH ...` that loads WAM. Immediately before this line, insert the following line to include the Graphics Analyzer interceptor into the environment:

   ```
   export LD_PRELOAD=/usr/lib/libcbe.so:/opt/graphics_analyzer/libinterceptor.so:$LD_PRELOAD
   ```

3. Reboot or turn on the device
4. Open the Graphics Analyzer host GUI application on your workstation
5. Connect to the device using the Graphics Analyzer Device Manager. The device appears under Linux Devices and can be chosen with a single click.
6. Start the web-based application
7. Observe function calls being traced in the Graphics Analyzer host GUI. You might see other applications being traced at the same time, because many applications in webOS are web-based.

--- Caution ---
Here and elsewhere, `libcbe.so` (Google Chrome) must be placed in `LD_PRELOAD` before `libinterceptor.so`. If you do not do this, web-based applications hang.

For an example web-based application, see the app store, `/mnt/otncabi/usr/palm/applications/com.webos.app.discovery`

### 3.6.4 Trace a QML application

QML applications are interpreted by `/usr/bin/qml-runner`.

These applications can be traced in the following way:
1. Navigate to `/usr/bin/`
2. Create the subdirectory `/usr/bin/ga/`, if it does not already exist
3. Hard link `/opt/graphics_analyzer/libinterceptor.so` into subdirectory `/usr/bin/ga/`
4. Rename the executable `/usr/bin/qml-runner` to `/usr/bin/qml-runner.bin`
5. Hard link `/opt/graphics_analyzer/aga-wrapper` to `/usr/bin/qml-runner`
6. Reboot or turn on the device
7. Open the Graphics Analyzer GUI application on your workstation. Connect to the device using the Graphics Analyzer Device Manager.
8. Start the QML application
9. Observe function calls being traced in the Graphics Analyzer GUI application
To temporarily disable the tracing of QML applications, rename `/usr/bin/ga/libinterceptor.so`, for example, to `libinterceptor.so.removed`.

For an example QML-based application, see the screensaver, `com.webos.app.screensaver`, under `/usr/palm/applications/com.webos.app.screensaver`.

### 3.6.5 Trace a native application

The following steps show the general method of tracing a native application:

1. Run the following command:

   ```
   luna-send -n 1 -f luna://com.webos.service.applicationManager/listApps "{}"
   ```

   Note the following information relating to the traced application:

   - **folderPath**: The home directory of the application
   - **main**: The main executable of the application. We will change this so that it loads the Graphics Analyzer interceptor.

2. Move into the home directory of the application.

3. Make a subdirectory named `ga`. Hard link `/opt/graphics_analyzer/libinterceptor.so` into this subdirectory. Do not use soft links because some native apps execute in a chroot jail and cannot see `/opt/graphics_analyzer` while running. As an alternative, copy `libinterceptor.so` into this subdirectory.

4. Add an extension `.bin` to the main executable of the application.

5. Hard link `/opt/graphics_analyzer/aga-wrapper` into the home directory of the application, giving `aga-wrapper` the same name that the main executable originally had. Alternatively, use a copy rather than a hard link.

6. Open the Graphics Analyzer host GUI application on your workstation

7. Connect to the device using the Graphics Analyzer Device Manager

8. Start the application being traced in the usual way

9. Observe function calls being traced in the Graphics Analyzer Host GUI application

To temporarily disable tracing, rename the hard link to `libinterceptor.so`, for example to `libinterceptor.so.removed`. Note that this will affect other native applications in the same directory.

---

### Example 3-1 webOS main menu (com.webos.app.home)

This is the application that draws the main system menu bar. The main executable of this application is `/usr/bin/com.webos.app.home`. The commands to trace this application are:

```bash
mkdir -p /usr/bin/ga
ln /opt/graphics_analyzer/libinterceptor.so /usr/bin/ga
mv /usr/bin/com.webos.app.home /usr/bin/com.webos.app.home.bin
ln /opt/graphics_analyzer/aga-wrapper /usr/bin/com.webos.app.home
```

After the host GUI has started and is connected to the webOS device, launch the main menu and observe function calls being traced. To temporarily disable tracing, use this command:

```bash
mv /usr/bin/ga/libinterceptor.so /usr/bin/ga/libinterceptor.so.removed
```
3.7 Troubleshooting

This section describes how to avoid some issues that might prevent Graphics Analyzer working correctly with your target.

This section contains the following subsections:

- 3.7.1 Target device without cp support on page 3-43.
- 3.7.2 No trace is visible on page 3-43.
- 3.7.3 Socket permission errors on page 3-43.

3.7.1 Target device without cp support

On systems that do not have cp, you can use cat on the target instead.

For example:

```
cat /sdcard/libGLES_aga.so > /system/lib/egl/libGLES_aga.so
```

3.7.2 No trace is visible

The interceptor component on the target reports through logcat on Android. If no trace is found then it is recommended to review the logcat trace.

In general, you should ensure the following:

- On Linux, ensure that the interceptor library is in your PRELOAD path
- On Android, ensure that your processlist.cfg is set correctly
- On Android, the system must be fully restarted to load the interceptor library
- Ensure you force close and reopen your application after installing the interceptor, to ensure the interceptor is loaded
- Ensure the daemon is started before the application
- Ensure your application is making OpenGL ES, EGL, OpenCL, or Vulkan calls

3.7.3 Socket permission errors

The Graphics Analyzer interceptor uses a Unix domain socket, AF_UNIX, to connect to the Graphics Analyzer daemon. This socket does not need any special permissions for your application.

However, if your application is compiled with targetSdkVersion >= 26, Android limits access to AF_UNIX sockets using security-enhanced Linux (SELinux) rules. For details about targetSdkVersion, see Android developer guides.

If you have followed the instructions in the installation guide but there is no trace output, see the log output on your device, for example by using adb logcat on Android. The following messages indicate that there is a problem which prevents the interceptor talking to the daemon:

```
06-26 20:57:18.430: I/aga_interceptor(9622): Trying to connect to the daemon...
06-26 20:57:18.430: E/aga_interceptor(9622): error socket connect: Permission denied
02-21 00:34:06.833 11030 11047 E aga_interceptor: (11047): SocketMessagePort: Unable to connect socket: Unable to connect socket: errno = 13
```

On unrooted Android devices, there are two workarounds for this issue:

- Build your application with a targetSdkVersion of 25 or lower. This workaround prevents the additional SELinux security policies from being applied.
- Rather than using the Graphics Analyzer daemon application, use the aga_me.py script as described in 3.3 Connect to your Android device from the command line on page 3-33. This workaround is compatible with targetSdkVersion settings of 26 and above.
If your Android device is rooted, you could try disabling the SELinux rules that cause this issue. You can do this by running one of the following commands:

• `setenforce 0`
• `supolicy --live 'permissive untrusted_app'`

Either command might produce an error. Use the one that works. You must re-issue this command every time the device reboots.
Chapter 4
Device Manager

The Device Manager is used to detect and to connect to devices that are running the Graphics Analyzer daemon.

The Device Manager can detect Android devices that are connected to the host, then install Graphics Analyzer components automatically. These components include:
- The system-level Vulkan layer
- The system-level interceptor for OpenGL ES, EGL, and OpenCL.

The Device Manager can automatically detect Linux devices that have the daemon running and are on the same local network and the same subnet as the host.

If a device has not been automatically detected, or it is outside the local network, the Device Manager can be used to directly connect to the device using its IP.

Launch the Device Manager by clicking the button.

It contains the following sections:
- 4.1 Android devices on page 4-46.
- 4.2 Linux devices on page 4-49.
- 4.3 Connect to an IP address on page 4-50.
4.1 Android devices

Device Manager can install Graphics Analyzer components to Android devices and connect Graphics Analyzer to your host machine with a single click. Device Manager displays Android devices that are connected using adb.

Prerequisites

To install Graphics Analyzer components to your Android device using Device Manager, target your application at Android SDK level 25 or below. This avoids security measures that were introduced in SDK 26 which block mechanisms used by components installed by the tool.

Note

If you need to lower the SDK level of your application, this is purely for the purposes of tracing or debugging. It is not a restriction over what you can release.

Device Manager is the easiest way to install Graphics Analyzer components onto your Android device. However, if it is not possible to target SDK 25, you have these options:

- Use the command line aga_me.py tool. For information, see 3.3 Connect to your Android device from the command line on page 3-33. This tool has no restrictions relating to Android SDK releases, but is limited to debuggable applications. In practice, this restricts it to applications for which you have the source code.
- Install Graphics Analyzer onto your device manually. For more information, see Chapter 3 Target installation on page 3-16.
- If you are using a rooted device, you can use Device Manager without restriction, but you must enter a special command at a root shell prompt at any point before you trace your application. For details, see 3.7.3 Socket permission errors on page 3-43. This command changes device settings to avoid the restrictions that were introduced in Android SDK 26. The changed settings remain until the device is restarted.

Setting up Device Manager

The first time that you launch Device Manager, you are prompted to provide the location of the adb binary. Device Manager can auto-detect your adb binary if it is in a standard location such as the system PATH. Alternatively, provide the location manually.

You can change the adb path at any time using Edit > Preferences > Graphics Analyzer.

Graphics Analyzer system-level components

The system-level components of Graphics Analyzer allow you to trace any application on your device, even if you do not have access to application source code. Device Manager can install these components, which normally requires a rooted device.

Note

Android devices are normally sold unrooted, that is, without system-level access. The way to root a device depends on the manufacturer and is beyond the scope of this document.

The Device Manager is conservative about writing to privileged areas of your device such as /vendor and /system. It runs checks to make sure that common failure conditions, like insufficient storage, do not exist. It tries to clean up if it fails unexpectedly. It warns you before it attempts anything that might be dangerous.

Although we recommend using the Device Manager over manual installation, Arm cannot guarantee that your device will not be corrupted. If you have the source code for your application and do not want to install system-level components, use the aga_me.py script to connect to your device. For details, see 3.3 Connect to your Android device from the command line on page 3-33.
### 4.1.1 Installing Graphics Analyzer components

After the Device Manager has found the adb binary, it searches for devices that are visible to adb, including devices that you have connected to over a network using `adb connect` and devices plugged into your host machine.

![Device Manager showing connected devices](image)

The Device Manager presents a list of all connected devices and attempts to detect each of the Graphics Analyzer target-side components:

**Interceptor**

Responsible for tracing your application. The Device Manager installs it into your `/system` or `/vendor` directory, as appropriate. This component allows you to trace any application that uses the OpenGL ES, EGL, or OpenCL APIs.

**Vulkan Layer**

Responsible for tracing Vulkan applications. The Device Manager installs it into your `/system` directory for Android 7, or your `/data` directory for earlier versions of Android. It allows you to trace any application that uses the Vulkan API.

---

**Note**

- Arm does not recommend that the Vulkan validation layers are used at the same time as the Graphics Analyzer Vulkan layer. For more information, see Chapter 3 Target installation on page 3-16.
- Since Device Manager installs the interceptor and Vulkan layer at system level, its installation process usually requires your phone to be rooted. For unrooted devices, an alternative approach is described in 3.1 Unrooted Android on page 3-18.

To install each component, click the install button. If the Device Manager detects that a component is out of date, it presents you with an update button instead. In both cases, it installs the most recent component and cleans up any leftover components.

### 4.1.2 Uninstalling Graphics Analyzer components

To uninstall a component, right-click on an Android device in the device table, and select the option to uninstall the component.

### 4.1.3 Connecting to your device

After you have installed the Graphics Analyzer daemon application, you can connect to your device from the Device Manager. The Device Manager handles setting up your device for tracing, including port forwarding, starting the daemon on the device, and enabling the Graphics Analyzer Vulkan layer.
--- Note ---

To trace your application, you must have installed either the interceptor or the Vulkan layer. Alternatively, you can package them into your application as described in 3.1 Unrooted Android on page 3-18.

---

Click the connect button, or double-click on a disconnected device to launch the daemon on your device and connect it to the host client. When a connection has been established, the Device Manager automatically closes and the new live trace is shown.

Click the button, or double-click on a connected device to disconnect from the device.

**4.1.4 Using the adb tasks log**

The adb tasks table contains a list of tasks that the Device Manager has attempted. You can see their status and the description of the task that was issued. The status can be success, failure, running, or pending.

Each adb task roughly corresponds to the instructions from Chapter 3 Target installation on page 3-16. The Device Manager maintains a record of every adb command that it has issued. Double-clicking on the task shows the command log for a specific task.

You can also use the Copy Command Logs button to copy one or more selected command logs to the clipboard, which can then be pasted into a text editor.

The Device Manager tries to provide a useful error message when an adb task fails. You can use the command logs to diagnose more complex installation issues.
4.2 Linux devices

The Device Manager can automatically detect a Linux device that is on the same local network and subnet as the host, and has the aga-daemon application running on it.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Protocol</th>
<th>GL Vendor</th>
<th>GL Renderer</th>
<th>GL Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2.200.200</td>
<td>D015</td>
<td>ARM</td>
<td>OpenGL ES 3.2</td>
<td>Mali-T760</td>
</tr>
</tbody>
</table>

**Figure 4-2 Linux device detected by Device Manager**

--- Note ---

The automatic detection is done using UDP broadcasts that are sent from the target device on port 5003. If the device is not automatically detected, it is likely that there are firewall rules blocking these UDP broadcasts, either on the host device, the target device, or somewhere else along the network. In this case, you can directly connect to the target device using its IP address instead.

Click the ![connect button](connect.png), or double-click on a disconnected device to connect to the device. When a connection has been established, the Device Manager automatically closes and the new live trace is shown.

Click the ![disconnect button](disconnect.png), or double-click on a connected device to disconnect from the device.

--- Note ---

The GL Vendor, GL Renderer, and GL Version strings of the device are obtained by dynamically loading the libGLESv2 and libEGL libraries, and might appear as Unknown if the libraries could not be found or loaded.
4.3 Connect to an IP address

If you know the IP address of a target device with the Graphics Analyzer daemon running on it, you can directly connect to it.

![Figure 4-3  Connecting to an IP address in Device Manager](image)

Enter the IP address of the target device and the port that the daemon is running on. The default port number is 5002. Click on the **connect** button to connect to the device. When a connection has been established, the Device Manager automatically closes and the new live trace is shown.
Chapter 5
Getting started

This chapter describes how to use the host GUI to configure and perform a trace and to capture frame buffer content while capturing a trace. It also describes how to use the capture modes in Graphics Analyzer to capture additional content.

It contains the following sections:

- 5.1 Run the host GUI on page 5-52.
- 5.2 Trace an application on page 5-53.
- 5.3 Process Configuration on page 5-54.
- 5.4 Dealing with multiple processes on page 5-56.
- 5.5 Pause, step frames, and resume on page 5-57.
- 5.6 Capturing frame buffer content on page 5-58.
- 5.7 Capturing all frame buffer attachments on page 5-59.
- 5.8 Limitations on capturing all frame buffer attachments on page 5-60.
- 5.9 Analyzing overdraw on page 5-61.
- 5.10 Analyzing the shader map on page 5-62.
- 5.11 Overdraw and shader map limitations on page 5-64.
- 5.12 Analyzing the fragment count on page 5-65.
- 5.13 Replaying a frame on page 5-66.
- 5.14 Frame replay limitations on page 5-67.
- 5.15 Frame overrides on page 5-68.
- 5.16 Debugging an OpenCL application on page 5-72.
- 5.17 Using GPUVerify to validate OpenCL kernels on page 5-73.
- 5.18 Comparing state between function calls on page 5-75.
- 5.19 Bookmarks on page 5-76.
- 5.20 Dealing with VR applications on page 5-77.
- 5.21 Tracing an application that is already running on page 5-78.
5.1 Run the host GUI

Follow these instructions to run the host GUI.

- On Windows, click **Start**, expand the studio folder, then select Graphics Analyzer.
- On Linux, run the command:
  ```bash
  <install_directory>/gui/aga &
  ```
- On macOS, press CMD+Space, type **Graphics Analyzer**, then press Enter.

The main GUI window opens.

To load one of the supplied sample traces from `<install_directory>/samples/traces/`, select **File > Open**, or click ![Open](image). The various application windows fill with information from the loaded trace which you can examine.
5.2 Trace an application

After you have successfully established a connection to a target device, a new live trace is shown in the GUI. At this point, you can use the Process Configuration to control which assets are sent from the target to the host during the trace.

After you have set the configuration, you can then start the application to be traced. If Graphics Analyzer has been correctly installed on the target, the live trace begins to receive trace data.

See Chapter 3 Target installation on page 3-16 for instructions on how to install Graphics Analyzer on a target, connect to the target, and to start an application on the target to be traced.

The connect button attempts to connect to the last connected device, or open the Device Manager if there was no previous connection.

To disconnect from the target, use the disconnect button.
5.3 Process Configuration

The Process Configuration controls which types of API assets are sent from the target device to the host when API function calls are traced.

For example, if you do not care about the contents of the textures and the buffers being used in your OpenGL ES application, you can use the Process Configuration to disable these asset types to speed up tracing of your application. In this case, Graphics Analyzer still tracks the existence and lifecycle of these assets, but their contents cannot be examined.

The more types of assets that are sent, the more information is visible in Graphics Analyzer. However, the application being traced runs more slowly, the trace is larger, and Graphics Analyzer uses more memory.

A distinction is made in the Process Configuration between explicit and implicit memory:

- Explicit memory is memory that is uploaded explicitly by a function call. For example, in a call to `glBufferData`, or any other function call that explicitly specifies a host pointer and a length.
- Implicit memory is memory uploaded by modifying a buffer that has been mapped into the host memory space, for example in a call to `glMapBuffer`, `vkMapMemory`, or any other function call that returns a pointer to memory that can then be modified by the application.

When a connection to a target device has been established, the Process Configuration pane can be opened, and the desired configuration can be set. The configuration that is set immediately after the connection has been established is the configuration that each traced application process is initialized with. The configuration can be changed at any time, even after the process has started, and affects any subsequently traced function calls.

![Process Configuration pane](image-url)
The configuration can be changed at any time, even after an application has been started on the target device. Any function calls traced in the application process after the new configuration has been set will use the newly set configuration.

If multiple processes are being traced at the same time in the same trace, each process has its own process configuration. Changing the process configuration of one traced process does not affect another.

A number of configuration presets are available, or a custom configuration can be manually set. Any configuration can be saved as the default, and is then used as the starting configuration for any future connections to target devices.

------ Note ------

By default, the **Legacy** configuration is used. The types of assets sent in this configuration are equivalent to the types sent by default in versions of Graphics Analyzer before the introduction of the Process Configuration.
5.4 Dealing with multiple processes

It is possible to trace multiple processes concurrently with Graphics Analyzer. Any intercepted application launched after the host has connected to the daemon is received into the trace file on the host. Individual processes are displayed in the 6.3 Trace Outline view on page 6-85.

Note

Most commands that are available for interacting with a live target only affect the currently selected process. This includes all capture commands, the resume and step commands, all replay commands including the frame overrides, and all automated trace commands, except the command to disconnect.
5.5 Pause, step frames, and resume

You can pause, step, or resume processes using different buttons in the host GUI.

The currently selected process can be paused by pressing the ✯ button. The process is halted at the next eglSwapBuffers() call to allow you to examine the result. Pressing this button again before the process has paused forces the application to pause on the next function call, regardless of whether it is eglSwapBuffers() or not.

All connected processes can be paused by pressing the ✯ button. The processes are halted at the next eglSwapBuffers() call to allow you to examine the result.

When paused, individual frames for the selected process can be rendered on target by stepping with the ▶ button.

The selected process can be resumed by pressing the ▶ button.

Note

Only the threads that are calling the graphics API functions are paused.
5.6 Capturing frame buffer content

While it is executing, it is possible to perform an in-depth capture of the output from the graphics system on a draw-by-draw basis.

Press the button to take a snapshot of the frame buffer content following each draw call in the next full frame of the application.

--- Note ---
This involves considerable work in composing and transferring data, which slows the application down.

---

To capture subsequent frames, you must pause the application using the button. After you have paused it, proceed by pressing the button for each frame that needs capturing.

In the Trace Outline view, any regular frame now has a icon to show that it is a captured frame.

There are some extra limitations on frame capture for Vulkan applications. For more information, see Vulkan Frame Capture View on page 6-94.

In addition to capturing the frame buffer content provided by the application, Graphics Analyzer has some special capture modes. These capture modes either capture extra content, such as frame buffer attachments, or modify the frame buffer content in some way. These capture modes are outlined later in this chapter.

To enable a capture mode, toggle the capture mode icon in the Graphics Analyzer toolbar, then press the button to trigger the frame capture.

--- Note ---
Only one capture mode can be enabled at any time.
5.7 Capturing all frame buffer attachments

It is possible to capture most frame buffer attachments including all color attachments, and the depth and stencil attachments.

While capturing a live trace, press the 🎨 button to toggle the capture mode. When a frame is captured with this mode enabled, all the available attachments are captured for each draw call. This information is visible for each frame buffer in the Framebuffers view.

In the following example, three squares are drawn on screen with varying depths moving from -1.0 towards 0.0, with a colored cube rendered behind them. All four draw calls, that is the three squares and one cube, have different values set for the stencil buffer write mask, with the stencil pass operation set to GL_REPLACE.

![Figure 5-2 Color attachment](image1)

![Figure 5-3 Depth attachment](image2)

![Figure 5-4 Stencil attachment](image3)

Depth attachment values range from -1.0 to 1.0, where -1.0 is full blue, 0.0 is black, and 1.0 is full red. The output is enhanced on the host to increase contrast.

Stencil attachment values are from 0-255, where 0 is black, and 255 is red.
5.8 Limitations on capturing all frame buffer attachments

There are some limitations on which attachments can be captured by this mode.

- This mode is disabled in the interceptor for Mali-400 series devices.
- It is not possible to capture the depth or stencil attachments for FBO 0 on an OpenGL ES 2.0-only configuration.
- For any configuration where depth texture sampling is not supported, or where the device only has OpenGL ES 2.0 available and the depth attachment is a renderbuffer, only a low-resolution capture of the depth buffer is possible.

Note

Capturing all attachments increases the per-draw-call capture time and the amount of data that is transmitted by the device to the host.
5.9 Analyzing overdraw

Overdraw mode highlights overdraw in a given scene.

While capturing a live trace, click 🔄 to toggle overdraw mode. When overdraw mode is enabled, whenever Graphics Analyzer captures a frame, it replaces the fragment shader in the target application with an almost transparent white fragment shader. Each time a pixel is rendered to the frame buffer, the alpha value is increased using an additive blend. Therefore, as more overdraw happens in an area, the whiter the final image appears. An application with low levels of overdraw is a uniform dull gray.

![Original image](image1.png)

**Figure 5-5 Original image**

![Image with overdraw mode turned on](image2.png)

**Figure 5-6 Image with overdraw mode turned on**

To see the level of overdraw in an area of the frame, move the cursor over the area in the Framebuffer view. The overlay displays the level of overdraw.

Any frame with overdraw mode turned on has the 🔄 icon in the Trace Outline view.
5.10 Analyzing the shader map

Graphics Analyzer can give each shader program that a scene uses a different solid color.

While capturing a live trace, click 🔄 to toggle the capture mode. While this capture mode is enabled, when Graphics Analyzer captures a frame it tracks which shader each object in the scene uses. It then maps each shader to a solid color. This mapping allows detection of any bugs that incorrect shader assignment might cause. An example of this feature is displayed here:

![Figure 5-7 Original image](image1)

![Figure 5-8 Image with shader map feature turned on](image2)

There are 100 unique colors that Graphics Analyzer can assign to shader programs, after which programs have duplicate colors. You can identify which program corresponds to each color by putting the cursor on a frame buffer image that was captured in shader map mode. The active shader is identified above the image.
Any frame with shader map mode turned on has the icon in the Trace Outline view.
5.11 Overdraw and shader map limitations

Applying any full-screen post-processing effects, for example rendering to a texture, prevents the overdraw map or shader map from displaying correctly on the device. However, the information can be seen by switching to the correct frame buffer in the UI after capturing a frame while either feature is active.
5.12 Analyzing the fragment count

Graphics Analyzer can count the number of fragments that are processed by a shader per draw call. If depth testing is enabled and a fragment would be excluded as a result, then that fragment is not included in the count.

To toggle this feature, click the button. When a frame is captured while this capture mode is enabled, each draw call increments the Fragments field of the fragment shader used to draw it. The fragment count represents the number of fragments that have been rendered with the selected shader in the current frame, up to and including the currently selected draw call. For example:

![Figure 5-9 Fragment count analysis]

The Total cycles field is calculated using the average number of cycles for a given shader, multiplied by the number of fragments processed.

The Fragments and Total cycles columns are only available for those frames where the fragment count analysis has been requested. These columns indicate N/A (not available) for other frames.

Any frame with fragment count mode turned on has this icon in the Trace Outline view.

--- Note ---

- You cannot capture frame buffer content while also collecting fragment shader statistics.
- To maintain compatibility with older OpenGL ES 2.0 hardware and software, the fragment count feature uses a software method to count the number of fragments. As a result, a single draw call can take several seconds to complete. In addition, the target device screen only shows the final draw call in a frame, and the frame capture feature does not show any usable information.
5.13 Replaying a frame

Graphics Analyzer can replay certain frames on the target device, depending on what calls were in that frame.

To see if a frame can be replayed, you must pause your application in Graphics Analyzer. When paused, if a frame can be replayed, the frame replay button \( \text{\textcircled{R}} \) and frame replay with capture button \( \text{\textcircled{S}} \) are enabled.

Click \( \text{\textcircled{R}} \) to cause Graphics Analyzer to reset the OpenGL ES state of your application back to how it was before the previous frame had been drawn. It then plays back all the function calls in that frame on the target device. The \( \text{\textcircled{S}} \) button operates in the same way except it also enables frame capture at the same time. See 5.6 Capturing frame buffer content on page 5-58 for details.

This feature can be combined with the 5.7 Capturing all frame buffer attachments on page 5-59, 5.9 Analyzing overdraw on page 5-61, 5.10 Analyzing the shader map on page 5-62, and 5.12 Analyzing the fragment count on page 5-65 features. You can, for example, pause your application at an interesting position, activate the Overdraw mode and then replay the frame. The previous frame is replayed exactly as before but with the overdraw mode enabled. You can repeat this process with the other modes enabled to get a complete picture for a single frame.
5.14 Frame replay limitations

Frame replay is disabled if the frame you want to replay does any of the following:

- Creates or deletes any OpenGL ES or EGL objects
- Changes state that had not previously been changed
- Is incomplete, in other words, there is no `eglSwapBuffers()` call
- Has calls from multiple threads or contexts
- Calls any unsupported function

You can find more information about why frame replay is disabled for a particular frame in the Console view.
5 Getting started
5.15 Frame overrides

Frame overrides

Graphics Analyzer can change a frame before replaying it on a target device. These modifications are made on the Frame Overrides view.

![Frame Overrides view](image)

Figure 5-10 Frame Overrides view

You can apply the following overrides:

- **5.15.1 Replace texture on page 5-68**
  
  Replace a selected texture with a 256 x 256 pixel texture of different colors in a grid-like pattern.

- **5.15.2 Force precision on page 5-69**
  
  Replace the shaders of a program with a version that forces a specific precision for all types.

- **5.15.3 Modify shaders on page 5-69**
  
  Replace both the fragment and vertex shaders of a program with custom versions.

This section contains the following subsections:

- **5.15.1 Replace texture on page 5-68.**
- **5.15.2 Force precision on page 5-69.**
- **5.15.3 Modify shaders on page 5-69.**

**5.15.1 Replace texture**

This texture can be used to ensure that you have generated the texture coordinates of your object correctly.
Only the textures that are listed on the Frame Overrides menu are replaced. These overrides are in use for every frame replay until they are removed from the Frame Overrides list.

5.15.2 Force precision

For a given program, the fragment shader and vertex shader are modified so that the precision specifiers highp, mediump, and lowp are replaced with the precision that you specify.

Also, default precision modifiers are applied to the source for the following types:

<table>
<thead>
<tr>
<th>ESSL version</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>#version 300 es</td>
<td>float, int, sampler2D, samplerCube, sampler3D, samplerCubeShadow, sampler2DShadow, sampler2DArray, sampler2DArrayShadow, isampler2D, isampler3D, isamplerCube, isampler2DArray, usampler2D, usampler3D, usamplerCube, usampler2DArray</td>
</tr>
<tr>
<td>#version 310 es</td>
<td>float, int, atomic_uint, sampler2D, samplerCube, sampler3D, sampler2DArray, samplerCubeShadow, sampler2DShadow, sampler2DArrayShadow, sampler2DMS, isampler2D, isampler3D, isamplerCube, isampler2DArray, isampler2DMS, usampler2D, usampler3D, usamplerCube, usampler2DArray, usampler2DMS, image2D, image3D, imageCube, image2DArray, iimage2D, iimage3D, iimageCube, iimage2DArray, uimage2D, uimage3D, uimageCube, uimage2DArray</td>
</tr>
<tr>
<td>Everything else</td>
<td>float, int, sampler2D, samplerCube</td>
</tr>
</tbody>
</table>

When the frame is replayed, the modified shaders are used in place of the original shaders, allowing the user to observe the effect of changing the precision mode.

5.15.3 Modify shaders

This override allows you to change both the fragment shader and vertex shader of a program.

When you select a program and select finish on the wizard, two text editors are displayed:

- One corresponds to the vertex shader of the program
- The other corresponds to the fragment shader of the program
Each editor allows you to edit the source for that particular shader. When you save your changes, these new source files are stored and used in the next replay frame of your application. This happens until the override itself is deleted.

Note

If you do not save your modifications, they are not used when replaying the frame.
There is also an option to compile the source to see if there are any errors in your modifications.

The results of the compilation can be found in the **Console** view, for example:

```
[INFO]: Processing shader...
[INFO]: Compilation successful
```

If the changes that were made to the shader are invalid, the frame override is not used and the original program is used instead. Invalid shaders are shaders that either do not compile, or the inputs and outputs do not match the original shader.
5.16 Debugging an OpenCL application

Graphics Analyzer supports tracing OpenCL applications in addition to OpenGL ES and Vulkan.

OpenCL tracing is a part of the interceptor library and does not require any special installation. Because it is not a graphics API, there are a few things to bear in mind when debugging OpenCL with Graphics Analyzer.

When you start or open an OpenCL trace, you are prompted to launch the OpenCL perspective. This perspective adjusts the visible views to only those that are supported for OpenCL. For more information, see 6.1 Perspectives on page 6-81.

The Assets view tracks and displays contexts, kernels, memory objects, and programs. Graphics Analyzer tracks the relationship between memory objects and subbuffers, which are displayed in the Assets view. Graphics Analyzer warns you about dangerous overlapping subbuffers.

Note

Graphics Analyzer tracks memory that is initialized into any memory object, using the CL_MEM_USE_HOST_PTR flag. It also tracks calls to clEnqueueCopyBuffer and clEnqueueWriteBuffer. However, Graphics Analyzer does not support sending changes to mapped OpenCL memory, or changes to memory caused by kernel invocations. This means that the memory reported in Graphics Analyzer might not accurately reflect your application.

Because there are no conceptual frames in OpenCL, the Trace Outline assigns all function calls to Frame 0 and Render Pass 0. The Trace Outline view displays the following function calls:

1. Function calls that enqueue commands
2. Function calls that block queued commands
3. clFlush() and clFinish() function calls, which issue command queues to a device

Blocking calls tell you how long they blocked for, and also the size of the wait list passed into the function call, if applicable.
5.17 Using GPUVerify to validate OpenCL kernels

GPUVerify is a tool that can be used to test whether an OpenCL kernel is free from various issues including intra-group data races, inter-group data races, and barrier divergence.

GPUVerify is a stand-alone application and is not provided by Graphics Analyzer. Download it from the Imperial College website, GPUVerify: a Verifier for GPU Kernels. Before using it with Graphics Analyzer, Arm recommends that you get it working in a stand-alone context, using its own documentation.

To use GPUVerify with Graphics Analyzer, you must first point Graphics Analyzer to the binary in your GPUVerify directory. To do this, click Edit > Preferences and fill in the Path to GPUVerify field, as shown here:

![Preferences dialog]

The Apply and OK buttons are grayed out if you enter an invalid path in this field.

When tracing an OpenCL application, Graphics Analyzer captures calls to all OpenCL 1.2 functions, including clCreateProgramWithSource, clCreateKernel, clCreateKernelsInProgram, and clEnqueueNDRangeKernel. It then uses the data from these functions to get the names of all the OpenCL kernels, the source code associated with them, and any run-time parameters that are known at this point, for example the local and global workgroup sizes.

If you want to run any of the kernels that were found in a trace through GPUVerify you must select a function call in the trace with live kernel objects, then click Debug > Launch GPUVerify. You are presented with the following dialog:
The **Kernel** drop-down list shows all the kernel objects that are live and have available source code at the currently selected function call. Source code is only available to Graphics Analyzer for kernels that have been created from a program that was created using `clCreateProgramWithSource` followed by `clBuildProgram`. If the kernel you expect to see is not in the list, ensure that you have selected a function call where that kernel object is live and that the kernel was created using the described method.

After you have selected a kernel from the drop-down list, the **Local Work Size**, **Global Work Size**, and **Kernel Build Options** fields are populated with all the information Graphics Analyzer has available. The local and global work sizes can either be picked from the drop-down lists, which have been populated with the local and global sizes from the enqueue history of the kernel, or the sizes can be manually entered with a comma-separated list of numbers. The number of dimensions of the local and global sizes must match. If invalid options are entered, an error box is displayed, giving more information. There are no restrictions to the Kernel Build Options field, though GPUVerify might output an error if the options are unsupported.

The output for GPUVerify is shown in the Graphics Analyzer **Console** view.
5.18 Comparing state between function calls

When investigating an application trace, you might want to compare the API state between two function calls to examine what has changed. You can do this using the Generate Diff command.

To compare changes in state between two functions in the trace, either:

- Select two function calls from the trace using Ctrl+click on Windows or Linux hosts, or Command +click on OS X hosts
- Select two draw calls from the Trace Outline view, then select Generate Diff from the popup menu

### Figure 5-16 Difference Report view

This view shows the difference between states for the two function calls. It is a table of the items that are different, or have changed at some point between the two functions. Values that have changed, but have since reverted to the original value are highlighted in light blue, whereas values that are different between the two functions are highlighted in red. Where state values are made up of multiple components, for example GL_VIEWPORT, the subcomponents are highlighted individually. In this case, subcomponents that have not changed are shown with gray text.

The final column in the output table is labeled Related functions. This lists the numerical id of function calls in the trace that modified the particular state item. By menu-clicking on one of the rows, you can navigate to one of the related functions using the menu that appears.

The results of the state comparison are persistent until the window closes, so it is possible to open multiple differences at a time allowing you to manually compare sets of changes.
5.19 Bookmarks

Graphics Analyzer contains a Bookmarks feature to allow you to bookmark particular function calls and optionally add notes to the bookmark.

These bookmarks can be saved and loaded with the trace. You can use this feature, for example, to make notes on a function call that looks like it might be a candidate for optimization, as a reminder.

Bookmarks can be viewed and manipulated in the 6.20 Bookmarks view on page 6-109 and the 6.2 Trace view on page 6-82.
5.20 Dealing with VR applications

Virtual Reality (VR) applications have a peculiar pattern in term of OpenGL ES calls. They usually have multiple threads or contexts to handle different steps, for example scene rendering, barrel distortion, and chromatic aberration.

Graphics Analyzer shows every context using a different color to make it clear what is part of which context. The relevant views are the 6.2 Trace view on page 6-82, the 6.3 Trace Outline view on page 6-85, and the 6.4 Timeline view on page 6-86.

When you pause a VR application or a generic multi-threaded application, pausing is delayed over the frame end until all render passes, including those from other threads, have finished. The function calls and render passes traced after the frame end are shown as part of the next frame in the 6.3 Trace Outline view on page 6-85. That frame is considered incomplete and is marked with the icon until the application is resumed and the end of the frame is reached.
5.21 Tracing an application that is already running

It is possible to attach to a process that is already running and recover almost all of the state of the OpenGL ES and EGL APIs, despite having not intercepted all of the calls from the application. In addition, it is possible to attach to and detach from the same process multiple times within a single trace. This allows you to skip tracing parts of the application that are not of interest.

Note
The Frame Replay and Automated Trace features are not available if Graphics Analyzer is attached to a running process or reattached to a detached process.

To intercept a running application, the daemon must have been started before the application was launched, but it is not necessary to connect the host to the target until the appropriate part of the application is about to be reached.

To detach from an application that is being traced, select the toggle button from the toolbar. When the application is detached from the debugger, the button is depressed. To reattach, toggle the same button. Function calls that contain information about the state of the target API following an attach are marked with the icon , as are frames in the outline view. A bookmark is also generated for the reattach function to aid navigation, see 5.19 Bookmarks on page 5-76.

Note
All core EGL and OpenGL ES assets and state items are recovered when attaching to a process as well as most assets and state items that are defined by extensions as long as those extensions become core in a later version of the API. However, some combinations of target driver and API version mean that certain data cannot be recovered.

A non-exhaustive list of things that might not be recovered are:
• OpenGL ES 1.1 contexts might not be supported correctly
• Anything that is defined within an extension that is not part of a later revision of an API
• Buffer contents on OpenGL ES 2.0-only devices that do not support the GL_EXT_map_buffer_range extension
• Buffer contents on any API version where the contents of the buffer are mapped at the point the host attaches, unless the full buffer was mapped as readable
• Texture data for most textures other than color renderable GL_TEXTURE_2D textures, and only for mipmap-level 0
• Program pipeline objects
• Programs and shaders that are created from binaries rather than from sources might not be supported correctly

Note
OpenCL and Vulkan applications are not supported.
Chapter 6
The Graphics Analyzer interface

This chapter describes the Graphics Analyzer host GUI which provides different views over the captured application trace. The GUI also provides access to headless mode, which enables automated data capture on the target.

It contains the following sections:
- 6.1 Perspectives on page 6-81.
- 6.2 Trace view on page 6-82.
- 6.3 Trace Outline view on page 6-85.
- 6.4 Timeline view on page 6-86.
- 6.5 Statistics view on page 6-87.
- 6.6 Function Call view on page 6-88.
- 6.7 Trace Analysis view on page 6-89.
- 6.8 Target State view on page 6-90.
- 6.9 Buffers view on page 6-91.
- 6.10 OpenGL ES Framebuffers view on page 6-92.
- 6.11 Vulkan Frame Capture view on page 6-94.
- 6.12 Assets view on page 6-96.
- 6.13 Shaders view on page 6-98.
- 6.14 Textures view on page 6-99.
- 6.15 Images view on page 6-100.
- 6.16 Vertices view on page 6-101.
- 6.17 Uniforms view on page 6-103.
- 6.18 Automated Trace view on page 6-104.
- 6.20 Bookmarks view on page 6-109.
- 6.21 Console view on page 6-110.
• 6.22 Scripting view on page 6-111.
• 6.23 Filtering and searching in Graphics Analyzer on page 6-113.
• 6.24 Host-side headless mode on page 6-114.
• 6.25 Target-side headless mode on page 6-116.
6.1 Perspectives

Graphics Analyzer includes a perspectives feature which allows related windows to be grouped for ease of use.

Graphics Analyzer comes with three perspectives:

- 🎨 OpenGL ES + EGL. This is the default.
- 🎨 Vulkan
- 🎨 OpenCL

These perspectives only have the views that are operational for the named API open by default.

Open new perspectives by selecting the 🎨 button. You can switch between perspectives at any time using the perspective switcher that is at the top right.

By default, Graphics Analyzer prompts you to switch perspectives when it detects that the traced process uses a different API to the currently selected perspective. This behavior can be changed in Edit > Preferences > Graphics Analyzer to either never prompt you, or to always automatically switch perspectives when a different API is detected.

Create custom perspectives by right-clicking on an existing perspective and selecting Save As …. Custom perspectives can be removed using Edit > Preferences > Perspectives.

Customize perspectives by moving, resizing, opening, and closing views. Open views using the Window > Show View menu. Customizations are saved when Graphics Analyzer is closed.
6.2 Trace view

The main window in Graphics Analyzer shows a table of function calls made by your application as it is running. Use this information to examine what your application requested from the graphics system and what the system returned.

Each call has:
- The time at which it was made
- The time at which it finished
- The duration of the call in microseconds

Important
This duration is the time that is spent in the driver for the function call and not how much time the GPU spends doing the work that is requested by the function call.

- The list of arguments sent to the call

Note
This list is truncated to save space. For a complete list of the arguments, see 6.6 Function Call view on page 6-88.

- The value, if any, returned by the underlying system when the function was called
- The error code returned by the underlying system when the function was called
- The process ID (PID) of the process the function call was made from
- The thread ID (TID) of the thread the function call was made from
- Any bookmark notes you have added to the function call. See 6.2.1 Add a bookmark to a function call on page 6-83.

Note
Some columns in this table might initially be hidden. Click the button to enable or disable columns.

Each call executed in a different EGL context is highlighted using a different color.

The Trace Outline shows a frame-oriented view of the trace. Each frame is delimited by a call to eglSwapBuffers(). Draw calls in a frame are grouped by the frame buffer that is bound at the time they were called. Selecting an item in the overview highlights the corresponding item in the main trace.

You can open documentation for an API call, if available, in a browser by menu-clicking on the call and selecting Open Documentation.

It is possible to select two function calls from the trace using Ctrl+click on Windows or Linux hosts, or Command+click on OS X hosts. Menu-clicking on one of the two selected functions displays a popup menu showing various options including the ability to generate a state difference report, see 5.18 Comparing state between function calls on page 5-75. Alternatively, select two draw calls from the Outline view and use the popup menu to compare the two draw calls instead.

Searching

To find a particular function call, or set of calls, Graphics Analyzer includes a search feature. You can open the search dialog by pressing Ctrl+F with the Trace view selected, or by selecting Edit > Find API call... from the main menu. Type your search string in the search box and Graphics Analyzer highlights matching calls in the trace. Use the and buttons to jump between the search results.

Press the button or the Esc key to close the search and hide the results.
By default, the search only looks at the function call name. If you want to search the function call parameters as well, select the Include parameters option. With this option selected, Graphics Analyzer searches the functions exactly as they appear in the Function Call column of this view.

See 6.23 Filtering and searching in Graphics Analyzer on page 6-113 for more information.

This section contains the following subsection:
• 6.2.1 Add a bookmark to a function call on page 6-83.

6.2.1 Add a bookmark to a function call

In the Trace view, the Bookmark Notes column lets you view and edit bookmarks for each function call.

Note
For general information on bookmarks, see Bookmarks on page 5-76.

Procedure
• To add a bookmark with a note, double-click on the Bookmark Notes column for the function call and enter your note.
• To add an empty bookmark, right-click on the function call and select Add Bookmark.

The bookmarked function call turns green in the trace and a green marker appears on the right-hand side of the trace. The highlighting and markers allow quick navigation to bookmarked function calls.

![Figure 6-1 Bookmarked function call with a note](image)

You can disable the highlighting and markers by right-clicking anywhere in the Trace view window and selecting Toggle Bookmark Highlighting.

![Figure 6-2 Toggling bookmark highlighting](image)
View and manipulate all bookmarks in the Bookmarks view on page 6-109.

Note

To remove a bookmark, right-click on the function call and select Remove Bookmark.
6.3 Trace Outline view

The Trace Outline view shows a summary tree view of the function calls made by your application as it is running. Use this view to easily navigate through the trace.

The top-level items in the tree are processes. If you have traced more than one process on the target system, you can switch between them by selecting them in this view. This causes the Trace view to display only calls for that process.

Function calls are further grouped depending on the API in use. For example, OpenGL ES calls are grouped into frames and render passes. For each item in the tree, including the groupings, the name, index, and extra interesting information for that item are shown. For certain items, extra information can be found in the tooltip for the item.

Selecting an item in the tree causes other views to be updated to provide information about that item. Selections that are made in the Trace view, Breadcrumb bar, or the Statistics view Charts tab cause the selection in the Trace Outline view to update. If the item selected in another view is not an item in the Trace Outline view, a selection line is shown to indicate where the item would be in relation to other items.

When you have a trace with many frames, you can use the Show Only Frames With Features Enabled option to quickly find interesting frames. With this mode turned on, only frames that meet one of the following criteria are shown:

- Frame Capture
- Frame Replay
- Fragment Count
- Overdraw Mode
- Shader Map Mode

The (Collapse All) button can be used to collapse all of the items in the tree.

Right-clicking on items in the tree allows you to generate a diff report between two items and export frame buffers. See 5.18 Comparing state between function calls on page 5-75 and 6.12.1 Exporting assets on page 6-96 for more information.

Each call that is executed in a different EGL context is highlighted using a different color.
6.4 Timeline view

The Timeline view shows a graphical representation of the function calls in your application.

There are three types of Timeline view:

- When you first open a trace, the Process Timeline view is shown. This shows you a high-level view of API function call activity for each process that was traced.
  — Single click a process to select that process.
  — Double-click a process to select the first frame in that process.
- When a frame or render pass has been selected, the Render Pass Timeline view is shown. This shows you each context in the selected process and each render pass in each context.
  — Single click a render pass to select that render pass.
  — Double-click a render pass to select the first draw call in that render pass.
- When a draw call has been selected, the Draw Call Timeline view is shown. This view is identical to the render pass view, except individual draw calls are visible.
  — Single click a render pass to select the first draw call in that render pass.
  — Single click a draw call to select that draw call.

All three types of Timeline view can be navigated in the same way:

- The X-axis can be scrolled by holding down the left mouse button and dragging left or right.
- The X-axis can be zoomed in or out by scrolling up or down using the scroll wheel, or by holding down the right mouse button and dragging up or down.
- Hovering over any element in the chart, including the axes, displays a tooltip showing context-sensitive information about that element.
6.5 Statistics view

Three tabs are available in this window, General, Charts, and Memory.

General

This tab gives statistics and averages for the currently selected process. Use this tab to gain an overview of the state of your application as it ran.

Charts

This tab shows charts for various statistics for the currently selected item type. For example, selecting a process shows you statistics about all the processes in the current trace. The currently selected item is highlighted in the chart and the item's parent is shown as the chart title. The following actions are available for the charts:

- Hovering over a slice of the pie chart allows you to see more information about the slice, including its value.
- Clicking on a slice selects that item in the trace.
- Double-clicking on a slice selects that item's first child in the trace.

Note

Some items do not support all the available statistics. For example, Render Passes do not support the number of render passes statistic.

Memory

This tab shows information on memory usage for each frame. This data can be produced on Mali-T600 or later based devices, but the vendor might choose not to enable this feature. To see if this feature is supported, check if your device contains one of the following files:

- /sys/kernel/debug/mali0/ctx/*/mem_profile
- /sys/kernel/debug/mali/mem/*

If the files are present and non-empty, then your device is supported.

Note

- To allow this data to be processed by Graphics Analyzer, turn off the SELinux permissions by running `setenforce 0` on your device. If this command produces an error, try the following command instead:

  ```sh
  supolicy --live 'permissive untrusted_app'
  ```

- You might need to mount the Linux debugfs mount point for this feature to work, using:

  ```sh
  mount -t debugfs /sys/kernel/debug /sys/kernel/debug
  ```

When selecting a frame, a pie chart showing the memory allocated by each channel is shown. Each channel is a driver-defined heap for a different type of object:

- Hovering over a slice of the pie chart shows you the memory contained in that channel.
- Clicking on a slice displays information about the memory contained in that trace and the percentage of total memory used in that frame.
- Double-clicking on a slice displays a histogram showing more details about the memory allocations.

The histogram shows the number of memory allocations made for each memory range. Hovering over each bar shows the total amount of memory this range contains.
6.6 Function Call view

This view has three sub tabs, Arguments, Additional Information, and Documentation. All the tabs show data for the selected function call. Therefore, the data is visible only when a function call is selected in the trace.

Arguments

This tab shows the unabridged arguments for the selected function call alongside their values. This can be useful because, due to the limited size of the 6.2 Trace view on page 6-82, the values of arguments are truncated in that view to save space.

Additional Information

This tab shows any additional information that is available about the currently selected function call. Only functions that are shown in the outline view have additional information.

Depending on the type of function call selected, different information might be shown. This view is useful when using indirect draw calls such as glDrawElementsIndirect. Since those calls use a command struct as a parameter, it is impossible to see what parameters were passed to the call in the normal trace view. It simply shows the value of the struct pointer. However, the struct is parsed and displayed in this view.

Documentation

This tab shows the Khronos documentation page for the selected function call, if it is available.
6.7 Trace Analysis view

This view shows problems or interesting information related to API calls as they were made. These problems can include improper use of the API, for example passing illegal arguments, or issues known to adversely impact performance.

Use this view to improve the quality and performance of your application.

Selecting an entry in this view highlights the offending issues within the trace view. Hover your cursor over the problem report to gain a more detailed view of the problem, if available.
6.8 Target State view

This view shows the state of the underlying system at a time following the function selected in the Trace view, and is updated as the trace selection changes. Use this view to explore how the system state changes over time and the causes of that change.

Note

This view is only available in the OpenGL ES + EGL perspective by default. For more information, see 6.1 Perspectives on page 6-81.

The initial state is shown as an entry in normal type. If the relevant API standard defines an initial state, then this is shown, otherwise <unknown> appears instead.

If a state item has been changed anywhere in the trace then it is highlighted in light green. A state item that is not currently its default value is highlighted in dark green. Any read-only constant states such as GL_MAX_DRAW_BUFFERS are highlighted in yellow. States that have never been changed in the trace are shown in white.

If a state has been changed, you can find the function call that changed it by selecting Select Previous Change Location from the right click menu on the state item. The function call that next changes a given state item can be located in a similar way.

You can use the Filter box to filter the states and values in the view. For example, type texture into the box and the view only shows states that have texture in the name or in the value. See 6.23 Filtering and searching in Graphics Analyzer on page 6-113 for more information.

In addition to the Filter box, there are several filtering modes available:

All states
   No additional filtering is applied.

States that have been modified
   Only show the states that have been changed in this trace.

States that have not been modified
   Only states that never change value in the trace are shown.

States that are not currently their defaults
   Only states that, at the currently selected function call, are not at their default values are shown.

States changed by this function
   Only states changed by the currently selected function call are shown.

Read-only states
   Only read-only constant states such as GL_MAX_DRAW_BUFFERS are shown.
6.9 Buffers view

This view shows information about the currently allocated buffer objects.

The list of buffer objects can be filtered by usage, or, for GLES only, by the last bound target column. The bottom part of the view shows the size of the currently selected buffer objects. If no buffer object is selected, the size of all the displayed buffer objects is shown.

To filter the view to show only the buffer objects with a given usage or last bound target, use the Filter box. For example, type transform into the box and the view only shows you the buffer objects that have transform in the usage field. See 6.23 Filtering and searching in Graphics Analyzer on page 6-113 for more information.
6.10 OpenGL ES Framebuffers view

After you have captured a frame using the Capture Frame button, you can find the results in the Framebuffers view. Use this view to gain an insight into what the graphics system has produced as a result of your application calls.

Note

This view is only available in the OpenGL ES + EGL perspective by default.

See 5.6 Capturing frame buffer content on page 5-58 for more information about capturing frames.

Figure 6-3 Selecting a framebuffer in OpenGL ES Framebuffers view

Select a frame buffer in the left-hand list to bring up a list of the attachments used in that frame buffer in the lower list, and histograms displaying the overdraw or shader map data, if relevant. Select an attachment to bring up a larger view of that attachment. For multiview rendering, the views are displayed as separate selectable elements in the attachments list. You can then mouse over the attachment for additional information. Double-clicking the main image opens the attachment in an external editor.

In certain situations, you might want to view the frame buffer with different alpha options. The following alpha modes are available:

**Use Alpha**

Does normal alpha blending using the alpha values in the frame buffer.

**Ignore Alpha**

Ignores the alpha values in the frame buffer and sets the alpha value for each pixel to its maximum, that is, fully opaque.
Visualize Alpha

Ignores the color information in the frame buffer and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

For a captured frame, it is possible to step through the sequence of draw calls for that frame one at a time and observe how the final scene is constructed.
6.11 Vulkan Frame Capture view

You can capture frame buffers for Vulkan applications using the Capture Frame button. For all calls to vkQueueSubmit within the captured frame, Graphics Analyzer captures each color, resolve, and depth/stencil frame buffer attachment that has been modified by each draw call inside each submitted command buffer.

Note

This view is only available in the Vulkan perspective by default.

See 5.6 Capturing frame buffer content on page 5-58 for more information about capturing frames.

Frame buffer attachment images that were created with anything other than the VK_SAMPLE_COUNT_1_BIT as the samples parameter are not captured. However, if the multisampled image has a corresponding resolve attachment in the render pass, the resolve attachment is captured after each draw call.

The Overdraw, Shadermap, and Fragment Count capture modes are unsupported for Vulkan applications, and the buttons are disabled when tracing an application that contains only Vulkan function calls.

Also, the Capture All Framebuffer Attachments button is disabled, because by default all frame buffer attachments are captured.

When a call to vkQueueSubmit has been selected, the tree of draw calls within the vkQueueSubmit is displayed in the Frame Capture view. Selecting any tree item shows information about that tree item.

![Figure 6-4 Selecting a subpass in Vulkan Frame Capture view](image)

If the currently selected call to vkQueueSubmit is within a captured frame, selecting a draw call displays the contents of all captured frame buffer attachments after that draw call was executed.
The 🔄 icon is displayed next to tree items when frame buffer attachment data has been received. Captured frame buffer attachments can be viewed when the data has been received by the host, so it is not necessary to wait until the Capturing frame dialog has completed to start examining the data.

In certain situations, you might want to view the frame buffer attachment with different alpha options. The following alpha modes are available:

**Ignore Alpha**
- Ignores the alpha values in the attachment and sets the alpha value for each pixel to its maximum, in other words, fully opaque.

**Use Alpha**
- Does normal alpha blending using the alpha values in the attachment.

**Visualize Alpha Channel**
- Ignores the color information in the attachment and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

The stencil component of combined depth/stencil attachments is visualized in the alpha channel of the displayed image.

High dynamic range (HDR) image formats such as VK_FORMAT_B10G11R11_UFLOAT_PACK32 are tone mapped for display in the UI by calculating the minimum and maximum floating-point values in the entire image for each channel, and then scaling the values for each channel to 0-255 according to these calculated minimum and maximum values. Channels that have a value of 0 are ignored by the tone mapper and remain at 0 in the displayed image.
6.12 **Assets view**

The Assets view shows the tree of created assets and their properties at the currently selected function call, for all supported APIs. Use this view to explore the API objects that your application has created and is using.

In the tab pane on the right side of the Assets view, Source, Image, and Buffer tabs are available. When certain assets, such as shaders, textures, or buffers, are selected, the appropriate tab that can preview the content becomes selectable and displays the asset data.

Under certain conditions, assets might be displayed differently in the assets view:

- Assets that are highlighted in green were created in the currently selected function call.
- Assets that are highlighted in light green were modified in the currently selected function call.
- Assets that are considered active, such as the currently bound OpenGL ES frame buffer, are displayed in bold type.

The right-click context menu for assets allows you to navigate to the API call that created or previously modified the asset.

This section contains the following subsection:

- **6.12.1 Exporting assets**

6.12.1 **Exporting assets**

Assets can be exported from your application trace to disk.

You can export frame buffers in the following ways:
• Selecting **File > Export All Captured Framebuffers**. This option is only enabled if OpenGL ES function calls are present in the application trace.

• Selecting the frames, render targets, or draw calls containing the captured frame buffers that you want to export from the Trace Outline view, right-clicking, and selecting **Export Selected Captured Framebuffers**. This option is grayed out if your selection does not contain any captured frame buffers.

You can export textures in the following ways:

• Right-clicking on a single function call and selecting **Export Textures**. This method exports all textures that existed at the time of that function call.

• Selecting the textures that you want to export from the Textures view, right-clicking, and selecting **Export Textures**.

You can export shaders in the following ways:

• Right-clicking on a single function call and selecting **Export Shaders**. This method exports all shaders that existed at the time of that function call.

• Selecting the shaders that you want to export from the Shaders view, right-clicking, and selecting **Export Shaders**.
Shaders view

This view is an alternative tabular view of all currently loaded shaders.

Note

This view is only available in the OpenGL ES + EGL perspective by default.

For each shader, various cycle counts are shown allowing you to identify which shaders are most costly. These cycle counts are calculated using the Mali Offline Shader Compiler for the Mali-T760 GPU.

The Cycles field in the table shows the estimate of the number of cycles each shader takes for a single invocation. This estimate might be inaccurate for any shaders that have any kind of non-linear control flow, such as a loop where the number of iterations cannot be statically determined, or if the shader contains any if statements.

\[
Cycles = \max(a, ls, t)
\]

\[
a = (\text{arithmetic shortest path} + \text{arithmetic longest path})/2
\]

\[
ls = (\text{load_store shortest path} + \text{load_store longest path})/2
\]

\[
t = (\text{texture shortest path} + \text{texture longest path})/2
\]

For fragment shaders, the Fragments column and other dependent columns are empty unless the Fragment count feature was active for the selected frame. See 5.12 Analyzing the fragment count on page 5-65 for more information.

Active shaders are shown in bold type.
6.14 Textures view

This view shows an alternative tabular view of all currently loaded textures and includes information on their size and format. Use this view to visualize the images that you have loaded into the system.

--- Note ---

This view is only available in the OpenGL ES + EGL perspective by default.

---

Loading textures is done using an external program.

--- Note ---

For larger traces, the application can take a short time to convert and display all textures.

---

The following texture formats are supported:

- `GL_COMPRESSED_RGBA8_ETC2_EAC`
- `GL_COMPRESSED_RGB8_ETC2`
- `GL_ETC1_RGB8_OES`
- `GL_LUMINANCE`
- `GL_ALPHA`
- `GL_RGBA4`
- `GL_RGB` with type `GL_UNSIGNED_BYTE`
- `GL_RGBA` with type `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT_4_4_4_4`
- `GL_COMPRESSED_RGBA_ASTC_*_KHR`
- `GL_COMPRESSED_SRGB8_ALPHA_ASTC_*_KHR`

In certain situations, you might want to view the textures with different alpha options. The following alpha modes are available:

**Use Alpha**

Does normal alpha blending using the alpha values in the texture.

**Ignore Alpha**

Ignores the alpha values in the texture and sets the alpha value for each pixel to its maximum, that is, fully opaque.

**Visualize Alpha**

Ignores the color information in the texture and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).
6.15 Images view

This view shows an alternative tabular view of all currently loaded images, including information on their size and format. Use this view to visualize the images that you have loaded into the system.

Note

This view is only available in the Vulkan perspective by default.

Loading images is done using an external program.

Note

For larger traces, the application can take a short time to convert and display all images.

The following image formats are supported:

- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32G32B32A32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_B10G10R11_UFLOAT_PACK32
- VK_FORMAT_ASTC_*_UNORM_BLOCK
- VK_FORMAT_ASTC_*_SRGB_BLOCK

The view has the following limitations:

- Only one layer is displayed for multiple-layer images
- Only base mipmap level is displayed
- Optimal tiling images content is displayed tracking buffer-to-image and image-to-image copy operations
- Image copy operations from and to buffer subregions and from and to specific image subresources are not supported

In certain situations, you might want to view the images with different alpha options. The following alpha modes are available:

Use Alpha
Does normal alpha blending using the alpha values in the image.

Ignore Alpha
Ignores the alpha values in the image and sets the alpha value for each pixel to its maximum, that is, fully opaque.

Visualize Alpha
Ignores the color information in the image and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).
6.16 Vertices view

This view has three sub tabs, Attributes, Indices, and Geometry. All the tabs show data for the selected draw call. Therefore, the data is visible only when a draw call, such as glDrawArrays or glDrawElements is selected in the trace.

Note

This view is only available in the OpenGL ES + EGL perspective by default.

This section contains the following subsections:

- 6.16.1 Attributes tab on page 6-101.
- 6.16.2 Indices tab on page 6-101.
- 6.16.3 Geometry tab on page 6-101.

6.16.1 Attributes tab

This tab shows the values of the vertex attributes that are passed to the vertex shader.

If GL_ELEMENT_ARRAY_BUFFER_BINDING or GL_ARRAY_BUFFER_BINDING is set, the corresponding buffer object is used to provide the values. The vertex indices used in this view have been sorted and duplicates removed.

6.16.2 Indices tab

This tab shows the original list of vertex indices that were passed to the draw call.

6.16.3 Geometry tab

This tab shows, as a wireframe, the geometry drawn by the draw call.

This tab allows you to get a quick idea of what the draw call was drawing and also to inspect the geometry for defects. You can see if the geometry is incorrect, due to missing or additional unexpected vertices, or if the geometry is too dense, which might lead to performance problems.

If you use the Geometry view in combination with the Framebuffers view, you can see where in the scene the geometry was drawn. This allows you to detect if the geometry is appropriate for its position in the scene. For example, if the geometry is always far away from the camera, the geometry detail can probably be lower. Or, if the complex internal geometry of an object is always occluded, it is probably not worth drawing.

To render the correct geometry, Graphics Analyzer must know which one of the shader attributes corresponds to the geometry position data. You can select this from the Position Attribute choice box. Graphics Analyzer uses the names of the attributes and their types to initially auto-select its best guess at a matching attribute.

The axes in the corner of the view show the orientation of the geometry relative to the three axes, X (red), Y (green), and Z (blue).

Note

The Geometry viewer and export function only work with the GL_TRIANGLES, GL_TRIANGLE_STRIP, and GL_POINTS draw modes. When using GL_POINTS, each point is rendered as a small tetrahedron. Primitive restart is also not supported. If you are using this feature, you might see unexpected results.
**Camera controls**

You can rotate the camera around the center of the geometry by clicking and dragging the primary mouse button in the view. For more precise rotation, the numeric keypad direction buttons 2, 4, 6, and 8 can also be used to rotate the camera.

To zoom the camera in and out, use the mouse scroll wheel or the W and S keys.

To move the camera, click and drag with the secondary mouse button or use the A and D keys to move left and right and the Q and E keys to move up and down.

To reset the position and orientation of the camera at any point, press the **Reset Camera** button.

**Exporting**

To do more in-depth analysis of the geometry, you can export it to a Wavefront .obj file. These files can be loaded by most 3D model editors and viewers. To export the geometry, right-click on the geometry viewer and select **Export to .obj**.

--- **Note** ---

- Wavefront .obj files do not support triangle strips so Graphics Analyzer converts any triangle strip data to a series of individual triangles when exporting.
- Wavefront .obj files do not support points so Graphics Analyzer converts any point data into a series of small tetrahedrons when exporting.
6.17 Uniforms view

This view shows uniform data for the active OpenGL ES shader program or programs, if program pipeline objects are in use, at the time of the selected function call.

--- Note ---

This view is only available in the OpenGL ES + EGL perspective by default.

---

For each active uniform, its index, location, type, and value are shown. If the uniform is a block, the block name and the block buffer binding are shown.
6.18 Automated Trace view

The Automated Trace view allows you to run a range of standard Graphics Analyzer commands automatically when a certain frame is encountered.

For example, you could run your application and automatically take a frame capture on frame 10, do a frame capture with overdraw mode switched on at frame 20 and do a frame capture with fragment count mode enabled at frame 30.

--- Note ---
- This feature requires a valid license.
- You can only add automated trace commands after an application has started.

To add an automated trace command, first select and pause the process that you want to add commands to and then open the **Automated Trace** view:

![Automated Trace view](image)

Select **Add Command** and the **Add Automated Trace Command** dialog opens:

![Add Automated Trace Command dialog](image)
Here you can select the type of command you want and which frames it applies to. You must specify at least one frame number to add a command. For a frame number to be considered valid, it must:

- Be greater than the current frame plus one
- Not already have an automated command associated with it

If the list of frames is invalid, the Frame number(s) text box is highlighted in red. A tooltip on the text box gives the reason.

——— Note ————

Empty frame numbers, which are represented by a series of commas with no numbers between them, and duplicate frame numbers are ignored.

When you have a valid list of frames, select the OK button:

![Add Automated Trace Command](image)

**Figure 6-9  Specifying a list of frame numbers**

You can then add more commands and remove existing ones:

![Automated Trace](image)

**Figure 6-10  Selecting command types**
When you are happy with the list, press the ➔ button. When the trace reaches frames that you have added commands to, those commands are executed.

Note

Automated trace commands for a frame are ignored if you send a play, step, or capture command in that frame, or in the frame before it.
6.19 **Render Pass Dependencies view**

Graphics Analyzer can work out what dependencies there are between different render passes in a selected frame.

--- **Note** ---

This view is only available in the OpenGL ES + EGL perspective by default.

The different types of dependencies it can detect are:

- If a render pass reads from a texture that was written to in a different render pass without being cleared. For example, render pass 0 draws to texture 1 and render pass 1 then reads from texture 1:

![Texture 1 diagram](image1)

Render pass 1 has a dependency on render pass 0 due to texture 1

- If a render pass has the same depth or stencil buffer bound as another render pass without being cleared, assuming that depth or stencil testing is enabled. For example, render pass 0 has render buffer 1 attached to its depth target and render pass 1 also has render buffer 1 attached to its depth target:

![Render buffer 1 diagram](image2)

Render pass 1 has a dependency on render pass 0 due to render buffer 1

- If a render pass does a `glBlitFramebuffer` call on a different frame buffer. For example, render pass 0 draws to frame buffer 1 and render pass 1 blits frame buffer 1 into frame buffer 2:
The Render Pass Dependencies view shows render pass dependencies for the selected frame. To generate a list of dependencies, select a frame in the Trace Outline view and press the Generate button in the Render Pass Dependencies view. Any render pass in the selected frame that is dependent on another render pass is shown in the tree. Expanding the render pass tells you:

- Which render pass it is dependent on
- Which frame that render pass is in
- Why Graphics Analyzer considers it a dependency

The dependency analysis stops at the first dependency for each render pass. To find out the next dependency in the chain, if there is one, select the frame with the earlier render pass in it and run the analysis again.

For example, in the following screenshot, two of the render passes in Frame 1, namely Render Pass 1 and Render Pass 4, have dependencies on previous render passes. Render Pass 1 is dependent on Render Pass 26 in the previous frame (Frame 0). Render Pass 4 is dependent on Render Pass 3 in the current frame (Frame 1). In both cases there are dependencies because Texture 18 is attached to the active frame buffer for the render passes:

![Figure 6-11  Render Pass Dependencies view showing dependencies](image)
6.20 Bookmarks view

This view shows bookmarks that have been added to the trace and allows you to add, remove, and edit bookmarks.

Bookmarks are links to specific function calls in the trace and can contain notes for you to add interesting information. Pressing the Add Bookmark button adds an empty bookmark to the currently selected function call in the Trace view. Pressing the Remove Bookmark button removes the selected bookmark from the Bookmarks view. You can edit a bookmark by double-clicking on the notes area next to the bookmark.

For more information, see 5.19 Bookmarks on page 5-76.

Bookmarks can also be viewed and manipulated in the Trace view.

You can jump to the function call associated with a bookmark by clicking the Go to Function button next to the bookmark.
6.21 Console view

This view contains a read-only console that Graphics Analyzer uses to present its internal log. You can attach the output from this view to bug reports and it might be helpful when making a support request.
6.22 Scripting view

The user interface might not have all the tools that you require to extract or analyze information that is retrieved from the target device. Therefore Graphics Analyzer has a Python scripting environment that allows you to directly interface with the Graphics Analyzer trace model. You can either perform an analysis natively in Python, or output the data that you need to an external file.

Note
This feature requires a valid license.

The Jython interpreter

The Scripting view contains a Jython interpreter that implements the Python 2.7.0 specification. The interpreter supports the standard Python syntax and the Python standard library.

There is one interpreter per trace file. These interpreters cannot share data. Closing the Scripting view closes all interpreters. The interpreter supports loading script code in the following forms:
- User script files that you load and run from the Scripting view.
- Python modules that you load using the import statement in a user script. Imported user modules are found by searching based on the JYTHON_PATH environment variable, not the PYTHON_PATH environment variable.

Note
After a script completes, all globally scoped declarations (imports, global variables, class definitions, function definitions, …) persist in the scripting environment.

For convenience, the interpreter is initialized with two extra global variables:

trace
A representation of the Graphics Analyzer internal model.

monitor
An interface to the progress bar underneath the interpreter input text area. You can use this interface to track progress in your scripts.

For more information about these objects, or any other Graphics Analyzer object, use the built-in Python help function to print API documentation. For example:

help(trace)

Graphics Analyzer also comes with some sample scripts in <install_directory>/samples/scripts/ that provide examples of different ways to perform analysis on a trace object.

The scripting console

The scripting console allows you to interact with the Jython interpreter.

You can use the Up and Down keys to move through a history of the commands you have previously executed.

Clicking Interrupt causes any running script and any created threads to stop.

Clear allows you to clear the output text area.

To reset the interpreter back to its original state, click Reset.
Loading scripts

The Scripting view contains an interactive interpreter, but for more complicated analysis it is easier to write a script in a separate Python file. The Scripting view also allows you to load Python scripts, or directories of Python scripts, from your file system. Graphics Analyzer only loads scripting files with the .py extension.

The script locations that you load are stored in your workspace and are persistent across runs of Graphics Analyzer.

To load a single script, click Add Script. To load a directory, click Add Directory. File-system changes to this directory are reflected in Graphics Analyzer.

To remove any top-level item, click Remove. The file and directory on the file system are not affected.

Scripts are loaded and displayed in a staging area next to the interpreter. To execute a script, either double-click it or highlight it in the staging area and either right-click and select Run or press the R key.

Performance considerations

The scripting environment is powerful, but also potentially memory intensive. The following tips might help improve the performance of your scripts:

• Holding global references to objects that you no longer need wastes memory. Delete an individual reference with the del keyword or click Reset to re-initialize the scripting environment, deleting all references.
• Only touching the parts of the model that you are interested in keeps memory usage low.
• Traversing the model forwards is faster than traversing backwards.
• Printing excessively large strings to the interpreter console can slow down the Graphics Analyzer user interface. If you must write large strings, write to an external file rather than the scripting console.
6.23 Filtering and searching in Graphics Analyzer

Filtering and searching in Graphics Analyzer uses Java regular expressions. For example, type gl_program|gl_texture into a filter or search box to match entries that contain gl_texture or gl_program. Filtering and searching in Graphics Analyzer is case insensitive.

By default, matches are performed on substrings. For example, program matches GL_PROGRAM. If you want to anchor your expressions you can use the standard regular expression boundary matchers such as ^ for the beginning of a line, and $, for the end of a line. For example, program$ matches GL_CURRENT_PROGRAM but not GL_PROGRAM_PIPELINE_BINDING.

If the filter you type is not a valid regular expression, the Filter or Search box goes red and the error is shown as a tooltip.

To learn more about Java regular expressions, see:
- Class Pattern
- Oracle regular expressions tutorial
6.24 Host-side headless mode

Headless mode allows you to do certain tasks without launching the Graphics Analyzer user interface. It allows you to automate tracing a target device, or export assets from an existing trace.

Note

This feature requires a valid license.

You can also use headless mode to trace a target device using just the target-side components. The main difference between using the host and target-side headless modes is where the trace output is stored. Host-side headless mode can only save traces to the host machine, and target-side headless mode can only save traces onto the target device. The interfaces for each headless mode are also slightly different. For more information, see 6.25 Target-side headless mode on page 6-116.

Note

Exporting assets from an existing trace file can only be performed on the host.

This section contains the following subsections:

• 6.24.1 Exporting assets on page 6-114.
• 6.24.2 Tracing a target device on page 6-115.

6.24.1 Exporting assets

You can use headless mode to export shader source code and texture assets from an OpenGL ES trace. You must provide an index in the trace from which to export.

For example, you could export the shaders that were loaded at the end of frame 15, or the textures that were loaded at function call index 500.

To do this, run the headless-aga script from the Graphics Analyzer installation directory. Pass in the location of the trace file to export assets from. You also must provide the output directory with the --export-output-directory switch. This directory is used to write the exported shaders and source code, and should be writable, otherwise the export operations fail.

You must provide the name of the process that you want to export assets from with the -p or --process switch. If your trace contains multiple processes with the same name, you must also provide the process ID of the process you want to export from, using the --process-id switch.

Provide the trace index for Graphics Analyzer to extract the assets from. Graphics Analyzer attempts to compute the OpenGL ES state at the index and export the assets that are present at that point.

You can specify the index as a function call by using these switches:

--export-function-textures
--export-function-shaders

Alternatively, you can index the trace by the frame number. This works the same way as selecting a frame in the Trace Outline. Graphics Analyzer indexes the trace to the function call at the end of the frame that you provide.

You can specify the index as a frame by using these switches:

--export-frame-textures
--export-frame-shaders

You can specify a trace index only, in which case Graphics Analyzer exports all assets of the specified type at that index. For example, --export-frame-shaders 15 would export all shaders from frame 15.

You can further provide a comma-separated list of asset IDs to export from the trace index, in which case Graphics Analyzer exports those assets only. This is done by joining the frame index to the list of asset
IDs with a colon (:) character. For example, --export-function-textures 100:1,2,3,4 would export only texture IDs 1, 2, 3, and 4 from function call index 100.

You can specify multiple frame indexes in the same command. Each index requires its own switch. Putting it all together, the final command should look similar to this example, using Unix formatting convention:

```
headless-aga /path/to/trace.mgd --export-output-directory /path/to/output-dir --process com.my.process --process-id 3089 --export-frame-textures 22 --export-function-shaders 200:1,2,3,4,5
```

This command exports the textures from frame 22 and the first five shaders from function index 200 into the output-dir directory.

### 6.24.2 Tracing a target device

You can use headless mode to trace a target device with the same configuration options that can be enabled from the GUI.

________ Note ________

To get a full list of all the command-line switches, invoke the headless script with missing or malformed arguments. For example, invoking `headless` with no arguments causes the command line to fail and triggers the headless mode help to print to the console. This help message contains an explanation of each of the command-line switches, which map to functionality from the GUI. So, if it is your first time running headless mode, or if you need a reminder of how to configure the trace, do this to get the headless mode documentation.

Headless mode assumes that the Graphics Analyzer daemon is already installed on the target device. Instructions are available in Chapter 3 Target installation on page 3-16.

Graphics Analyzer requires you to provide details about the device to connect. Use an IP address to connect using the --device-ip switch. Make sure that the aga-daemon executable is running on the target. Graphics Analyzer attempts to connect using port 5002.

You must specify a process name with --process. Headless mode does not support tracing multiple processes.

You must provide a file name with --trace-file-output. This is the file that the trace is written to when the session ends. You must provide a file path that points to a writable location and include the file name you want to use, for example /path/to/my-trace.mgd. Use the .mgd extension so that Graphics Analyzer recognizes it as a trace file. If you do not use the standard extension, it is appended to the file name automatically.

________ Note ________

If the file you provide exists, Graphics Analyzer tries to delete it when it saves the trace.

To tell Graphics Analyzer when to stop tracing, use --timeout to set a timeout in seconds, or --exit-at-frame to set a frame limit for the trace.

You can also use the command line to set the trace configuration. Use the preset configurations, such as Full Trace, which allows you to replay the trace later, using the --trace-config switch. Alternatively, you can manually tell the interceptor to collect specific data, depending on your needs. For the full list of switches and valid inputs, run headless without any arguments to generate the help message.
6.25 Target-side headless mode

Headless mode allows you to perform a trace on the target device without a connection to the host Graphics Analyzer application. In target-side headless mode, trace files are stored in a local directory on the target device.

Note
This feature requires a valid license.

There are two ways to use headless trace mode:

• Using a global configuration file
• Using the daemon in headless mode

You can also use headless mode to trace a target device from the host machine, without using the host GUI. The main difference between using the host and target-side headless modes is where the trace output is stored. The host headless mode can only save traces to the host machine, and the target headless mode can only save traces onto the target device. The interfaces for each headless mode are also slightly different. For more information, see 6.24 Host-side headless mode on page 6-114.

Note
Exporting assets from an existing trace file can only be performed on the host.

This section contains the following subsections:

• 6.25.1 File locations on page 6-116.
• 6.25.2 Headless configuration file reference on page 6-117.
• 6.25.3 Starting the daemon in headless mode on page 6-119.
• 6.25.4 Arguments accepted by the daemon on page 6-120.

6.25.1 File locations

The locations for the configuration file and the output trace file on the target device depend on the target OS.

Android

By default, trace files are saved to the $EXTERNAL_STORAGE/traces/ directory if the app loading the interceptor has the Android WRITE_EXTERNAL_STORAGE permission, or /data/data/{package-name} if the permission is not available.

The default configuration file should be placed in $EXTERNAL_STORAGE/aga-headless.conf if the app loading the interceptor library has the Android READ_EXTERNAL_STORAGE permission, or /data/data/{package-name}/aga-headless.conf if not.

Note
• If the target device is non-rooted, make sure that the app manifest contains the WRITE_EXTERNAL_STORAGE permission and that the app has that permission enabled in Settings. Without this permission, it is not possible to set the configuration file for the app or write the trace files, as internal app storage cannot be accessed without su, which requires the device to be rooted.
• Android devices might implement specific SELinux policies which prevent Graphics Analyzer from reading the headless mode configuration file. You can verify if this is the case by checking if there are any error messages using adb logcat -s audit. The issue can be solved by disabling SELinux on your rooted device.
• The WRITE_EXTERNAL_STORAGE permission also implicitly grants READ_EXTERNAL_STORAGE, so the interceptor always looks for the config file in $EXTERNAL_STORAGE/aga-headless.conf if this permission is available.
• On most devices $EXTERNAL_STORAGE is /sdcard.

Linux

Trace files are saved to the $HOME/traces/ directory by default.
The configuration file should be placed in $HOME/aga-headless.conf.

6.25.2  Headless configuration file reference

The headless configuration file is formatted as JSON. The top-level object contains a single key, processes, which is an array of JSON objects, with each object corresponding to a configuration for an individual process.

--- Important ---

Configuration file items are case-sensitive. Make sure that keys and values are typed exactly as shown.

<table>
<thead>
<tr>
<th>Key</th>
<th>Required?</th>
<th>Type</th>
<th>Accepted values</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Yes</td>
<td>String</td>
<td>A valid process name, or package name on Android</td>
<td>Name of the process</td>
<td>None</td>
</tr>
<tr>
<td>config</td>
<td>No</td>
<td>String</td>
<td>fullTrace, everything, balanced, functionsOnly, legacy</td>
<td>Preset config name</td>
<td>legacy</td>
</tr>
<tr>
<td>customConfig</td>
<td>No</td>
<td>Object</td>
<td>A valid customConfig object, see Custom config reference on page 6-117</td>
<td>Custom configuration of resources per API</td>
<td>None</td>
</tr>
<tr>
<td>traceDirectory</td>
<td>No</td>
<td>String</td>
<td>A valid filesystem path</td>
<td>Directory used for creating headless trace files</td>
<td>None</td>
</tr>
<tr>
<td>frameCaptures</td>
<td>No</td>
<td>Object</td>
<td>A valid frameCaptures object, see Frame captures reference on page 6-118</td>
<td>Allows setting frame captures frame numbers and modes</td>
<td>None</td>
</tr>
<tr>
<td>disconnectBeforeFrame</td>
<td>No</td>
<td>Integer</td>
<td>A valid frame number</td>
<td>Number of the frame to disconnect and disable tracing before</td>
<td>None</td>
</tr>
</tbody>
</table>

Custom config reference

The customConfig object contains keys for each API. Values are objects containing keys specifying whether resources are enabled for that API. All keys are optional.

For an example of how to format this object, see Example configuration file on page 6-119.

--- Note ---

Custom configuration resource toggles are always applied after the preset. This allows a preset to be applied first and then modified by applying the changes made by the custom preset on top. For example,
after applying the Everything preset and a custom config with gles.shaderSources disabled, the resulting config has all resources except OpenGL ES shader sources enabled.

Table 6-2  customConfig object keys

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl</td>
<td>OpenCL config object. See OpenCL config object reference on page 6-118.</td>
</tr>
<tr>
<td>gles</td>
<td>OpenGL ES config object. See OpenGL ES config object reference on page 6-118.</td>
</tr>
<tr>
<td>vulkan</td>
<td>Vulkan config object. See Vulkan config object reference on page 6-118.</td>
</tr>
</tbody>
</table>

OpenCL config object reference

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>programSources</td>
<td>Boolean</td>
</tr>
<tr>
<td>explicitMemory</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

OpenGL ES config object reference

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaderSources</td>
<td>Boolean</td>
</tr>
<tr>
<td>shaderUniforms</td>
<td>Boolean</td>
</tr>
<tr>
<td>shaderBinaries</td>
<td>Boolean</td>
</tr>
<tr>
<td>textureContents</td>
<td>Boolean</td>
</tr>
<tr>
<td>explicitBuffers</td>
<td>Boolean</td>
</tr>
<tr>
<td>implicitBuffers</td>
<td>Boolean</td>
</tr>
<tr>
<td>outputBuffers</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Vulkan config object reference

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaderBinaries</td>
<td>Boolean</td>
</tr>
<tr>
<td>implicitMemory</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Frame captures reference

The frameCaptures configuration object contains keys for each frame capture mode. All keys are optional.

For an example of how to format this object, see Example configuration file on page 6-119.

Table 6-3  frameCaptures object keys

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Integer array</td>
</tr>
<tr>
<td>overdrow</td>
<td>Integer array</td>
</tr>
<tr>
<td>fragmentCount</td>
<td>Integer array</td>
</tr>
</tbody>
</table>
Table 6-3  frameCaptures object keys (continued)

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaderMap</td>
<td>Integer array</td>
</tr>
<tr>
<td>allAttachments</td>
<td>Integer array</td>
</tr>
</tbody>
</table>

**Example configuration file**

This topic shows an example of a headless configuration file.

```json
{
  "processes": [
    {
      "name": "com.example.application",
      "customConfig": {
        "cl": {
          "programSources": false,
          "explicitMemory": false
        },
        "gles": {
          "shaderSources": false,
          "shaderUniforms": false,
          "textureContents": false,
          "explicitBuffers": false,
          "implicitBuffers": false,
          "outputBuffers": false
        },
        "vulkan": {
          "shaderBinaries": false,
          "implicitMemory": false
        }
      }
    },
    {
      "name": "cube",
      "config": "fullTrace",
      "customConfig": {
        "gles": {
          "outputBuffers": true
        }
      },
      "frameCaptures": {
        "default": [50],
        "overdraw": [65, 81],
        "allAttachments": [22],
        "shaderMap": [25, 33],
        "fragmentCount": [70, 73, 76, 80, 90]
      },
      "disconnectBeforeFrame": 50
    },
    {
      "name": "com.sample.teapot",
      "traceDirectory": "/some/path/"
    }
  ]
}
```

**6.25.3 Starting the daemon in headless mode**

As an alternative to using a headless configuration file, you can start the daemon in headless mode by passing arguments to it.

--- Note ---

A limitation of using the daemon arguments instead of a headless configuration file is that only one process, specified using `--name`, can be configured at a time, whereas the configuration file can contain multiple configurations for different processes.

---

The daemon accepts a series of arguments. For details, see 6.25.4 Arguments accepted by the daemon on page 6-120. If the `--HeadlessMode` argument is present, the daemon starts in headless mode.
Otherwise, it starts as normal, ignoring the rest of the headless configuration arguments. The daemon
does not reset the headless mode configuration until you kill the daemon or you connect to the host.
The process for starting the daemon in headless mode depends on the target device. However, the
argument names are the same for both methods.

For a Linux device, start the daemon executable, passing extra arguments as required. For example:

```bash
./aga-daemon --HeadlessMode --name cube
```

To stop the daemon, kill it like any other Linux process. This clears any headless configurations that
were passed in as an argument.

### 6.25.4 Arguments accepted by the daemon

You can pass these arguments to the daemon when starting it in headless mode.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Default value</th>
<th>Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--HeadlessMode</code></td>
<td>Start the daemon in headless mode</td>
<td><code>false</code></td>
<td>No, but required for the other options to apply</td>
</tr>
<tr>
<td><code>--name/-n {process_name}</code></td>
<td>Name of the process</td>
<td>None</td>
<td>Yes, if <code>--HeadlessMode</code> is present</td>
</tr>
<tr>
<td><code>--config/-c {preset_name}</code></td>
<td>Config preset. See accepted values for the <code>config</code> key in 6.25.2 Headless configuration file reference on page 6-117.</td>
<td><code>legacy</code></td>
<td>No</td>
</tr>
<tr>
<td><code>--traceDirectory/-o {directory}</code></td>
<td>Output directory for trace files</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td><code>--disconnectBeforeFrame/-d {frame_number}</code></td>
<td>Frame number before which to stop tracing</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td><code>--frameCaptures.[CaptureMode] {comma-separated_list_of_frame_numbers}</code></td>
<td>Frame captures for the given mode, where <code>[CaptureMode]</code> is one of the frame capture keys. See Frame captures reference on page 6-118, given as a comma-separated list.</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td><code>--cl.[Resource] {1/0}</code></td>
<td>Explicitly enable (1) or disable (0) a resource, where <code>[Resource]</code> is one of the CL resource keys. See OpenCL config object reference on page 6-118</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
<td>Default value</td>
<td>Required?</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>--gles.[Resource] {1/0}</td>
<td>Explicitly enable (1) or disable (0) a resource, where <code>[Resource]</code> is one of the OpenGL ES resource keys. See <em>OpenGL ES config object reference on page 6-118.</em></td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>--vulkan.[Resource] {1/0}</td>
<td>Explicitly enable (1) or disable (0) a resource, where <code>[Resource]</code> is one of the Vulkan resource keys. See <em>Vulkan config object reference on page 6-118.</em></td>
<td>None</td>
<td>No</td>
</tr>
</tbody>
</table>
Chapter 7
Integration with Arm Streamline

The Graphics Analyzer interceptor library generates Arm Streamline annotations and chart information. When profiling an application with Arm Streamline, and provided the interceptor is installed and being used, the following additional information is now available:

- Charts showing:
  - Frames per second
  - Direct and indirect draw calls per frame
  - Vertices and instanced vertices
  - Vertices per frame
- A per process activity view, which shows:
  - Active contexts
  - Frames within each context
  - Render passes within each frame
  - Important calls per render pass, including draw calls, frame end calls, and flushing calls
- The **Heat Map** and **Core Map** views show the active EGL contexts for threads of intercepted processes.

It contains the following sections:
- *7.1 Installation on page 7-123.*
- *7.2 Using Arm Streamline annotations on page 7-124.*
### 7.1 Installation

Install Arm Streamline and set up your host machine and target device.

See *Getting started with Streamline* in the *Arm Streamline User Guide* for more information about Arm Streamline.

To set up Graphics Analyzer, follow the instructions in *Chapter 3 Target installation on page 3-16*. 
7.2 Using Arm Streamline annotations

The details panel and charts in Arm Streamline display a range of extra information.

Charts

The interceptor provides five charts that track draw calls, frame rate, and vertices.

Draw Calls / Frame

This chart shows, for each EGL context, the number of draw calls per frame. Selecting a range with the caliper tool gives you the average for that period. This chart is stacked so the total height indicates the total number of draw calls at any given time.

Frame Rate

This chart shows, for each EGL context, the average frame in frames-per-second. The frame rate, $r$, is calculated using a simple rolling average over the last six frames. Selecting a range using the caliper tool shows an average value for that period.

Indirect Draw Calls

This chart shows, for each EGL context, the number of indirect draw calls. This information indicates how much extra work the GPU might be doing, as it is not possible to determine the number of vertices or instanced vertices for these draw calls. Selecting a range using the caliper tool shows the total value for that period. This chart is stacked so the total height indicates the total number of indirect draw calls at any given time.

Vertices

This chart shows, as a global total for the application, the number of vertices and instanced vertices sent with all direct draw calls. The two series are overlaid so that the height of the instanced vertices series shows the total number of vertices processed by the vertex shader. It is possible to select a range using the caliper tool and see the total number of vertices and instanced vertices for that period. For programs not using instanced rendering, the two series are the same.

Vertices / Frame

This chart shows, for each EGL context, the number of vertices per frame. Selecting a range using the caliper tool shows an average value for that period. This chart is stacked so the total height indicates the total number of vertices at any given time.

Graphics Analyzer Activity View

A new view is available from the mode menu in the details panel for each process that was traced using the Graphics Analyzer interceptor. This view shows the following:

- Active contexts on each thread.
- Each frame within a context.
- Each render pass within a frame.
- Interesting API calls within a render pass.
It is possible to select a frame, render pass or call item and see its relationship with other items. Selecting a frame highlights all render passes within that frame and all calls associated with each render pass. Selecting a render pass highlights the chain of render passes and calls for a given frame so far. Selecting a call highlights all previous calls within a render pass.

Information such as the time spent in the driver for an item is available by hovering over the item. Render passes also give an indication of the reason for the render pass. For example, `eglSwapBuffers` for the end of frame, or `glBindFramebuffer(fboID)` indicating that you changed the bound draw FBO.

For more detailed information, zoom in to a level where it is possible to see individual API calls. Calls are color coded to indicate the type of call they are.

### API Call Marker Colors

<table>
<thead>
<tr>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Flushing calls such as <code>glFlush</code></td>
</tr>
<tr>
<td>Green</td>
<td>End of frame calls such as <code>eglSwapBuffers</code></td>
</tr>
<tr>
<td>Blue</td>
<td>Direct draw commands such as <code>glDrawArrays</code></td>
</tr>
<tr>
<td>Yellow</td>
<td>Indirect draw commands such as <code>glDrawArraysIndirect</code></td>
</tr>
</tbody>
</table>

### Heat Map and Core Map Annotations

The **Heat Map** and **Core Map** views show active EGL contexts for each rendering thread. The length of each bar indicates the duration that that context was active between `eglMakeCurrent` calls.
Chapter 8
Known issues

This chapter describes some known issues in this release of Graphics Analyzer.

It contains the following sections:
- 8.1 OpenGL ES extensions on page 8-127.
- 8.2 Shading language version on page 8-128.
- 8.3 Shader compiler on page 8-129.
- 8.4 API asset coverage on page 8-130.
- 8.5 Memory on page 8-131.
- 8.6 Partial support for earlier trace versions on page 8-132.
- 8.7 Graphics Analyzer becomes unresponsive when closed while dynamic help is open on page 8-133.
- 8.8 Issues viewing Khronos reference pages on page 8-134.
- 8.9 Intercepting without using LD_PRELOAD on page 8-135.
- 8.10 Multiple drivers installed on the system on page 8-136.
- 8.11 Daydream VR compositor skips right eye execution on capturing a frame on page 8-137.
8.1 OpenGL ES extensions

In general, known OpenGL ES extensions appear in the trace, but do not have an effect on the trace or asset state.
8.2 Shading language version

The host application supports syntax-highlighting of shaders from OpenGL ES 3.1 and earlier. There is support for OpenCL kernel source but this is limited to a plain text view.
8.3 Shader compiler

The system uses a built-in version of the Mali Offline Shader Compiler to determine shader cycle counts. These cycle counts are based on compilation for a Mali-T880 GPU.
8.4 API asset coverage

Graphics Analyzer has limited support for most API extension enumerations. Not all asset types are covered at present, and not all texture types and formats are currently supported.
8.5 Memory

Capturing for long periods of time or attempting to capture many frame buffer or other images is expensive on memory. If a memory shortage is detected, an active trace might be terminated to protect the system.

See 2.1.1 Increase the available memory on page 2-14 for advice.
8.6 Partial support for earlier trace versions

As new features are added to Graphics Analyzer, the data format of the saved traces is updated. We aim to support opening and visualizing traces that were captured with an earlier version of the tool but cannot guarantee full functionality for older traces. The best solution is to update the Graphics Analyzer software on both target and host and re-trace the application.
8.7 Graphics Analyzer becomes unresponsive when closed while dynamic help is open

If you try to close Graphics Analyzer while the dynamic help view is open, Graphics Analyzer does not close and becomes unresponsive.

To avoid this issue, ensure that you close the dynamic help view before closing Graphics Analyzer. The application then closes normally.

This issue only affects Linux.
8.8 Issues viewing Khronos reference pages

Depending on your platform, you might or might not be able to correctly view the Khronos reference pages when double-clicking functions. This is particularly an issue on Windows which uses Internet Explorer. It is outside of our control, as it is an incompatibility between the Khronos reference page format and the platform default browser.
8.9 Intercepting without using LD_PRELOAD

In some cases, it might not be possible to use LD_PRELOAD, for example if LD_PRELOAD is already being used for another purpose.

In such cases, you must define both LD_LIBRARY_PATH and MGD_LIBRARY_PATH as follows:

```
LD_LIBRARY_PATH=/path/to/intercept/dir/:$LD_LIBRARY_PATH
MGD_LIBRARY_PATH=/path/to/original/drivers/dir/
```

In this example, /path/to/intercept/dir/ is the directory on the target where the installation files were copied to. This directory must contain libinterceptor.so, and include symlinks to libinterceptor.so named libEGL.so, libGLESv2.so and libGLESv1_CM.so.

You can set up the required symlinks for libinterceptor.so as follows:

```
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libEGL.so
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libGLESv1_CM.so
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libGLESv2.so
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libOpenCL.so
```

The directory /path/to/original/drivers/dir/ should contain the pre-existing libGLESv2.so and libEGL.so files from the graphics driver installation.

LD_PRELOAD does not need to be defined when using this method.

When a graphics application runs, the Graphics Analyzer interceptor libraries are loaded from the LD_LIBRARY_PATH first. These interceptor libraries dynamically load the original graphics libraries from the MGD_LIBRARY_PATH location, as required.

--- Important ---

You might find that the original Mali drivers pointed to by MGD_LIBRARY_PATH are small shim libraries that do not export any entry points, but instead depend on libmali.so. If so, the interceptor fails to correctly load the driver libraries unless MGD_LIBRARY_PATH also contains libmali.so. If this is not the case, you can either point MGD_LIBRARY_PATH to the location of libmali.so, regardless of whether that location also contains the libEGL or libGLES libraries, or you can point MGD_LIBRARY_PATH to a location that contains symlinks to libmali.so instead.
8.10 Multiple drivers installed on the system

More than one version of the Mali driver might be installed on your device. For example, if you aim to use both X11 and FBDEV on the same Linux platform.

If so, it might not be possible to use the standard LD_PRELOAD approach on its own. See 3.4.4 Trace an OpenGL ES, EGL, or OpenCL application on page 3-35 for details.

Instead, you must use that approach as normal while defining the MGD_LIBRARY_PATH environment variable as follows:

```
MGD_LIBRARY_PATH=/path/to/original/drivers/dir/
```

The /path/to/original/drivers/dir/ contains the pre-existing libGLESv2.so and libEGL.so files from the graphics driver installation.

When a graphics application runs, the Graphics Analyzer interceptor libraries are preloaded as normal. The interceptor libraries then dynamically load the original graphics libraries from the MGD_LIBRARY_PATH location as required.

See 8.9 Intercepting without using LD_PRELOAD on page 8-135 for more information.
8.11 Daydream VR compositor skips right eye execution on capturing a frame

When you capture a frame of a Daydream application, the VR compositor might skip the right eye execution. This is because capturing the frame buffer content takes longer than the expected operation duration.

In this case, frames only contain:
• The draw calls made by the application to prepare the eye buffers.
• The left eye draw calls to FB0 made by the VR compositor.

One way to make the VR compositor draw both eyes on capturing is to disable distortion and asynchronous reprojection. From the Daydream application, check Settings > Developer options > Force Undistorted Rendering.

To enable Developer options, tap Build Version seven times from the Daydream application settings. For more information, see The Controller Emulator.

Alternatively, you can disable the asynchronous reprojection from your application using setAsyncReprojectionEnabled(boolean). For details, see GvrLayout.
Appendix A
Analytics

It contains the following sections:

- *A.1 Analytics information* on page Appx-A-139.
- *A.2 Disable analytics data collection* on page Appx-A-140.
A.1 Analytics information

Arm collects anonymous information about the usage of our products to help us improve our products and your experience with them.

Product usage analytics contain information such as system information, settings, and usage of specific features of the product. You can enable or disable the feature in the product settings. Product usage analytics do not include any personal information.

Host information includes:

- Operating system name, version, language, architecture, and locale
- Number of CPUs
- Amount of physical memory
- Screen resolution
- Processor and GPU type
- Java environment

Product information includes:

- Build ID, version, and edition
- License information

Feature information includes:

- OS architecture of the target device
- GPU vendor of the target device
- GL renderer of the target device
- Open GL version running on the target device
- Is the target device running a rooted Android?
- Number of times a trace is taken on the system
A.2 Disable analytics data collection

Use these options to disable the collection of product usage analytics data.

Procedure

- Set the environment variable `ARM_DISABLE_ANALYTICS` to any value, including 0 or an empty string, to disable analytics collection for all tools running in that environment.
- The command-line option `--disable_analytics` disables analytics collection for that single tool invocation.
- Uncheck the option Edit > Preferences > Product usage analytics > Allow collection of product usage analytics.

Note

The preference is not persistent across tool versions. If you uncheck the option on a particular Graphics Analyzer version and then install a newer version, the preference reverts to the default.