Graphics Analyzer

Version 5.7

User Guide

arm

101545_0507_00_en

Graphics Analyzer

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0505-00	14 February 2020	Non-Confidential	New document for v5.5.
0506-00	21 August 2020	Non-Confidential	New document for v5.6.
0507-00	20 November 2020	Non-Confidential	New document for v5.7.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact *terms@arm.com*.

Contents
Graphics Analyzer User Guide

Preface

This preface introduces the *Graphics Analyzer User Guide*.

It contains the following:

About this book

This document describes how to install and use Arm® Graphics Analyzer to examine an application running on an Android or Linux target device.

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 Before you begin

Follow the steps in this section to prepare your applications ready to trace using Graphics Analyzer.

Chapter 3 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 extra content.

Chapter 4 Analyzing your trace

Learn about the different ways you can analyze your trace in more detail.

Chapter 5 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 6 Integration with Arm Streamline

The Graphics Analyzer interceptor library generates Streamline annotations and chart information.

Chapter 7 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.

`mono`space

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

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

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

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

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

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_0507_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® Developer*.
- *Arm® Documentation*.
- *Technical Support*.
- *Arm® 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 Graphics and Gaming Community*.

It contains the following sections:

1.1 Introduction to Graphics Analyzer

Graphics Analyzer enables you to explore the graphical output of your Android application, and identify scenes that might be causing performance problems. It captures every OpenGL ES or Vulkan API call that your application makes as it runs on a target device.

Use Graphics Analyzer to evaluate the impact of each draw call and look for opportunities to optimize performance, for example:

- To check that the level of detail is appropriate, look at the frame buffer output alongside object geometry
- To check for pixels that are unnecessarily shaded multiple times, capture the level of overdraw in a scene
- Find the most expensive shaders and where they are used in a scene.
- Check texture formats and the level of compression used.

To see Graphics Analyzer in action, *watch this video*.

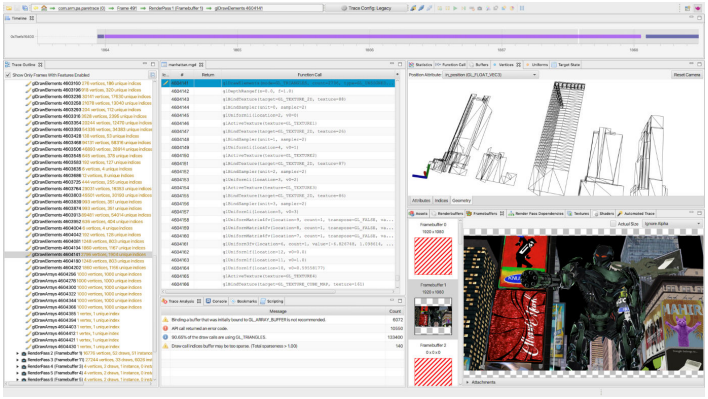


Figure 1-1 The Graphics Analyzer user interface

Examine API behavior

Graphics Analyzer shows you all the function calls the application makes as it runs on your connected device. This information helps you to understand exactly what the application requested from the graphics system and what the system returned. You can then investigate how and why the system state changed over time.

At each function call, see the currently allocated buffer objects, shaders, and textures your application is using, and their properties.

Graphics Analyzer reports problems such as improper use of the API. For example, passing illegal arguments, or any issues that are known to adversely impact performance.

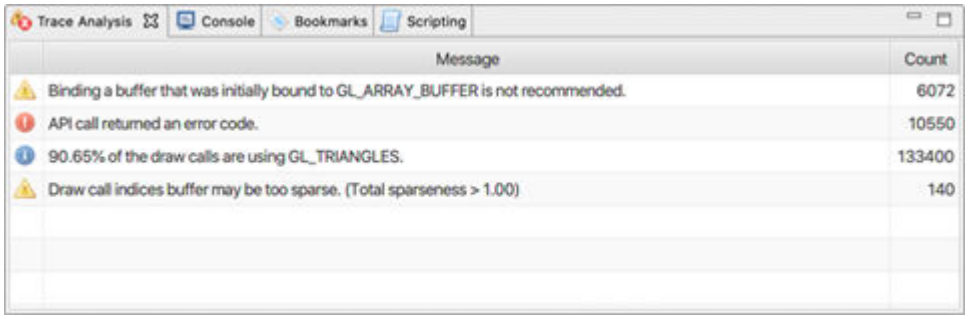


Figure 1-2 Trace Analysis view

Frame buffer outputs

See the visual output to each frame buffer, alongside the color, depth, and stencil attachments. Step through the draw calls to see how the frame is composed.

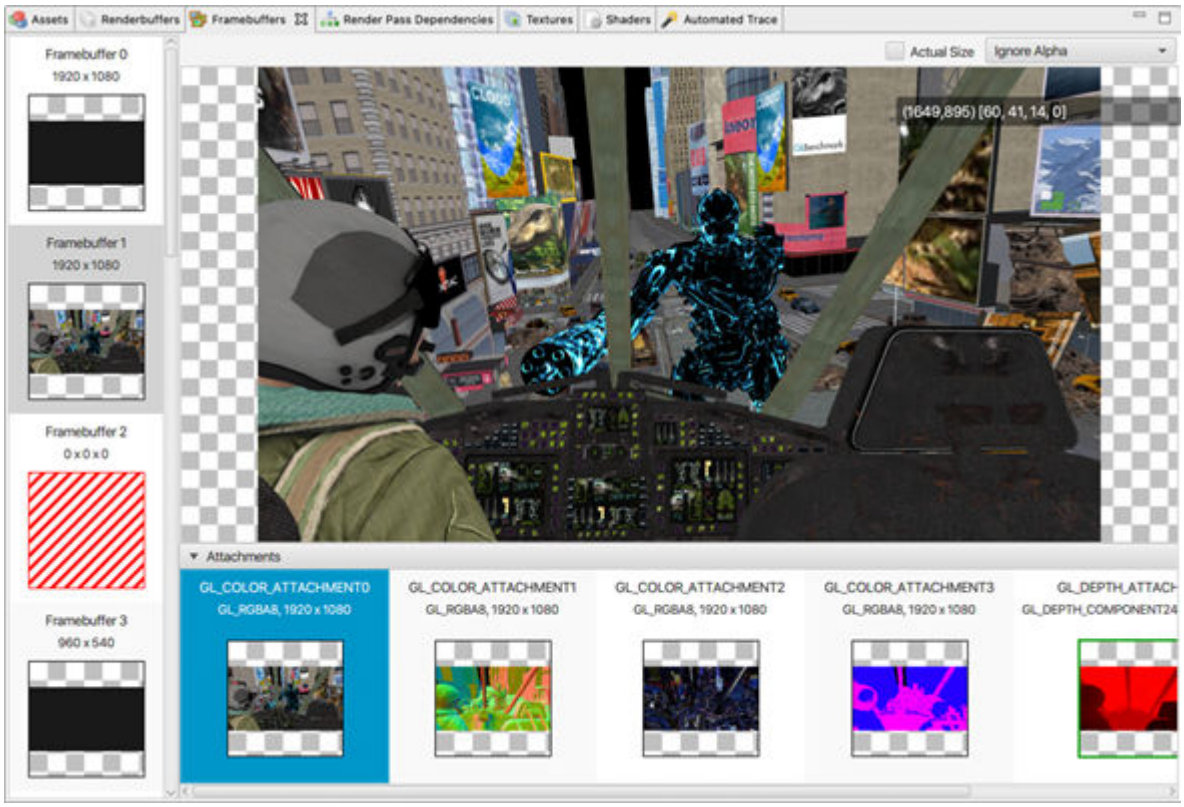


Figure 1-3 Framebuffer view with color and depth attachments

Geometry

Viewing the geometry alongside the frame buffer output helps you evaluate whether the level of detail is appropriate for the object in the scene. Step through each draw call to see the impact on the scene and whether you could improve performance by:

- Reducing the complexity of meshes.
- Using level of detail meshes for objects that are further away from the camera.

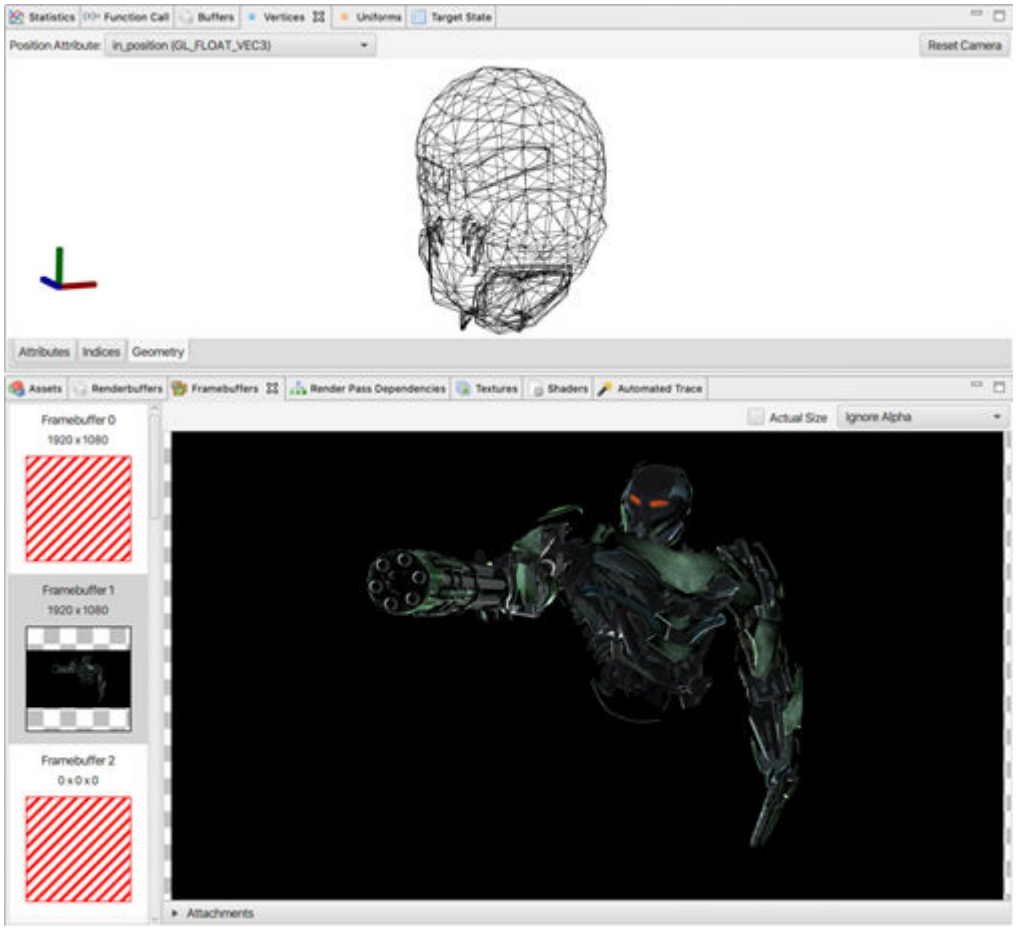


Figure 1-4 Geometry view

Fragment count analysis

If a scene has too many fragments to process, or if fragments are too expensive to process efficiently, you might see slow performance. In Graphics Analyzer, you can analyze the number of fragments processed by each shader, alongside approximate cycle cost and register usage.

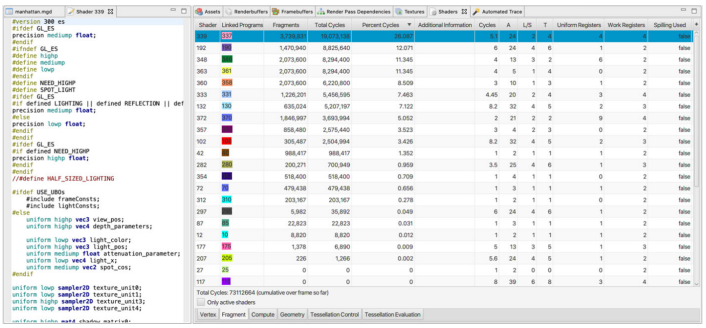


Figure 1-5 Fragment count analysis alongside shader program source

Overdraw

If content has a high degree of overdraw it can perform poorly, because of the cumulative cost of shading multiple layers. Poor performance can occur even if the layers are simple, especially for devices running at high resolutions and frame rates. To show overdraw in Graphics Analyzer, areas of the scene are overlaid with white, where multiple fragments are shaded per output pixel. The whiter the area, the more overdraw is present.

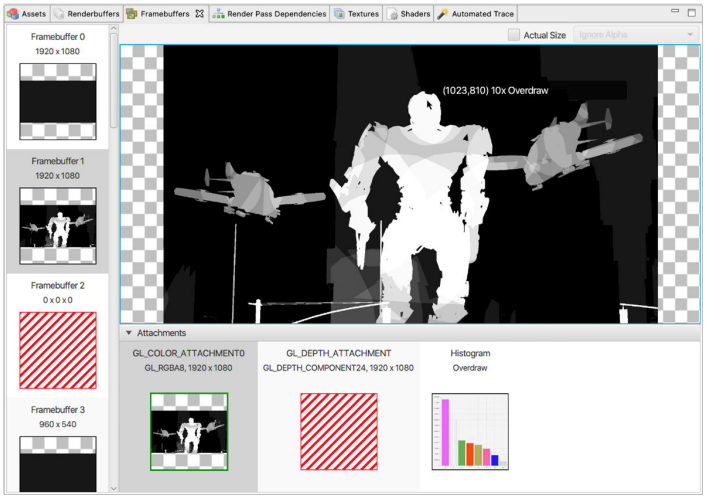


Figure 1-6 Framebuffer view showing overdraw analysis

Shader analysis

See which shaders are used in each part of the scene. The shader map helps you to detect problems where incorrect shaders have been assigned. It also helps you find the most expensive shaders that have the greatest impact on game performance. These shaders can be targeted for optimization, either by reducing their complexity, or by reducing the number of fragments they must process.

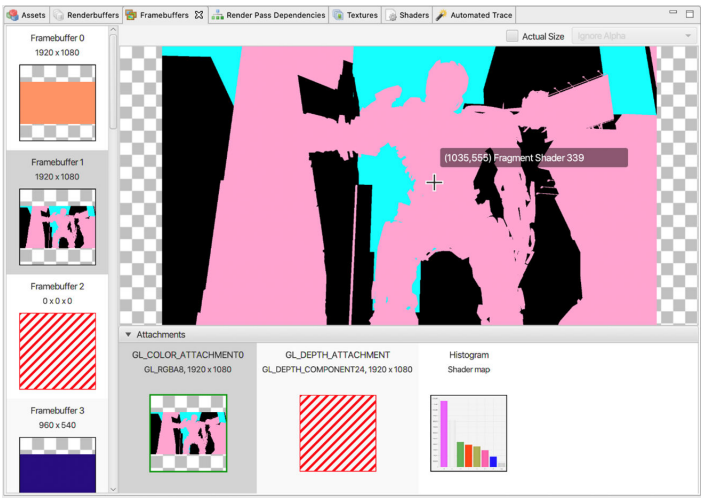


Figure 1-7 Framebuffer view showing shader map analysis

Get started

Capturing data with Graphics Analyzer is easy. Your device must be in developer mode and have USB debugging enabled. Graphics Analyzer connects to the device through USB, and traces your application as it runs on the device. When you come to a problem area, pause Graphics Analyzer and capture extra frame data, such as the frame buffer output, shader map, or level of overdraw.

See *Chapter 3 Getting started* on page 3-32 for full details.

1.2 Installation package

Graphics Analyzer is available for the Arm Mobile Studio or the Arm Development Studio product suites. The installation package contains everything that you need to start investigating GPU applications.

Download *Arm Mobile Studio* or *Arm Development Studio*, depending on the 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:

Arm Mobile Studio users

`android/arm/`

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

Arm Development Studio users

`linux/arm/32-bit_softfloat/`

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

`linux/arm/32-bit_hardfloat/`

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

`linux/arm/64-bit/`

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

Licenses

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 what each license includes:

- *Arm Mobile Studio Editions*.
- *Arm Development Studio Editions*.

For instructions on adding your license, see:

- *Licensing Arm Mobile Studio*.
- *Licensing Arm Development Studio*.

Chapter 2
Before you begin

Follow the steps in this section to prepare your applications ready to trace using Graphics Analyzer.

It contains the following sections:

2.1 Host system requirements

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

- If you are tracing a Linux target, it can connect to the target using TCP/IP, for online analysis, and has port 5002 open.
- At least 8GB of system RAM is available. Arm recommends that you have at least 12GB of RAM available.

This section contains the following subsections:

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 or `g` for gigabytes.
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`, or `g`.

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 **Edit** > **Preferences** and selecting an existing directory for the **Custom temporary storage directory** field.

2.2 Target system requirements

Graphics Analyzer supports tracing on specific Khronos Group APIs.

For Android 8 and above:

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

Note

A list of the recommended Android devices that support Graphics Analyzer is available from the Arm Mobile Studio *Supported Devices* page on Developer.

For platforms other than Android:

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

2.3 Prerequisites

To run Graphics Analyzer on an unrooted Android target device, ensure that the host and target are configured correctly.

Configure your host machine

- Install the *Android Debug Bridge (ADB)*, which is available with the *Android Software Development Kit (SDK)*.
- Edit the `PATH` environment variable on the host machine to include the path to the Android SDK platform tools directory.
- The application that you want to trace must be debuggable.
- To modify the application that you want to analyze, you must have access to the source code.

Configure your device

- Enable *Developer Mode*, then enable USB debugging on the target device, by selecting **Settings > Developer options**.
- If **Device Manager** does not find your device, test the adb connection. Connect the target device to your host machine, then run the `adb devices` command in a terminal window. If the adb connection is good, you will see the ID of your device with no permission errors, and you can run `adb shell` without issues. See the *Android Studio User Guide* for more information.
- The Android device must be running Android 8.0 or above.

2.4 Preparing non-debuggable applications

To trace non-debuggable applications, you must install a Vulkan or OpenGL ES layer driver, manually install and configure `aga-daemon` to collate information from different traced applications, then use the **Device Manager** to connect to your device.

These instructions are applicable to Vulkan and OpenGL ES Android 10 applications only. Tracing of non-debuggable applications on older Android versions is not supported.

Note

Loading layer drivers into non-debuggable applications is only possible using Android eng or userdebug builds of the operating system. It is not possible in release builds of the operating system.

Procedure

1. Install the layer driver appropriate to which API you are using:
   - To allow Vulkan applications to be traced using the Vulkan API layers system, you must install the interceptor as a global Vulkan layer on your device:
     1. Copy `libVK_layer_aga.so` onto the device then into the app folder:

        ```
        adb push libVK_layer_aga.so /data/local/tmp
        adb shell chmod 777 /data/local/tmp/libVK_layer_aga.so
        adb shell cp /data/local/tmp/libVK_layer_aga.so /data/data/<package_name>
        ```

     2. Enable layers:

        ```
        adb shell settings put global enable_gpu_debug_layers 1
        adb shell settings put global gpu_debug_app <package_name>
        adb shell settings put global gpu_debug_layers VK_LAYER_ARM_AGA
        ```

     See *Install Vulkan validation layers on Android* for more information.
   - To allow OpenGL ES applications to be traced, you must install the interceptor as a global OpenGL ES layer on your device:
     1. Copy `libGLES_layer_aga.so` onto the device then into the app folder:

        ```
        adb push libGLES_layer_aga.so /data/local/tmp
        adb shell chmod 777 /data/local/tmp/libGLES_layer_aga.so
        adb shell cp /data/local/tmp/libGLES_layer_aga.so /data/data/<package_name>
        ```

     2. Enable layers:

        ```
        adb shell settings put global enable_gpu_debug_layers 1
        adb shell settings put global gpu_debug_app <package_name>
        adb shell settings put global gpu_debug_layers_gles  libGLES_layer_aga.so
        ```

     See *Install OpenGL ES layers on Android* for more information.
2. Open a shell on your host machine.
3. Navigate to the Graphics Analyzer installation directory.
4. Copy the daemon to your target device:

   ```
   adb push target/android/arm/aga-daemon /data/local/tmp
   ```

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

   ```
   adb shell
   su
   ```

6. Install the daemon on the device, and grant appropriate permissions:

   ```
   cp /data/local/tmp/aga-daemon /system/bin
   chmod 777 /system/bin/aga-daemon
   ```

7. Enter the following commands on the host machine:

   ```
   adb forward tcp:5002 tcp:5002
   adb shell aga-daemon
   ```

Next Steps

Open Graphics Analyzer and the **Device Manager**, then open the **Linux/IP** tab to connect to the default IP address 127.0.0.1 and port 5002. See *3.2 Tracing Android applications* on page 3-34 for more information.

2.5 Linux

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

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

This section contains the following subsections:

2.5.1 Prerequisites

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

- A network connection to a host running the Graphics Analyzer GUI.
- The target must permit TCP/IP communication on port 5002.

2.5.2 Install Graphics Analyzer on a Linux target

To install Graphics Analyzer on a Linux target, take the following steps.

1. Navigate to `...linux/arm/` and then to the `32-bit_softfloat`, `32-bit_hardfloat`, or `64-bit` directory, according to your configuration.

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

   Note
   - The Linux interceptor supports Armv7, Armv8, and Intel 64-bit 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.
   - 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 set the execute permission bit by running the following command from inside the directory that you copied `aga-daemon` into:

   ```
   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. Copy `libinterceptor.so` to anywhere on your target device, and rename it to `libVK_layer_aga.so`.

Note

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

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

Note

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

2.5.3 Connect the host and the target

To connect Graphics Analyzer on your host device to the target device you want to trace, 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. Only close it when you have finished tracing all the applications.
2. Connect to `aga-daemon` running on the device using the Device Manager. 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 *3.3 Tracing Linux devices and IP address of target devices* on page 3-37 for more information.
3. Start the application that you want to trace by following the instructions in *2.5.4 Trace an OpenGL ES or OpenCL application* on page 2-22, or in *2.5.5 Trace a Vulkan application* on page 2-22.

2.5.4 Trace an OpenGL ES or OpenCL application

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

To preload the library, 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 *7.1 Intercepting without using LD_PRELOAD* on page 7-103 for an alternative method.

If you have more than one version of your graphics driver on your system and are having issues, see *7.2 Multiple drivers installed on the system* on page 7-104 for more information.

2.5.5 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. You must also tell it 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 `VK_DEVICE_LAYERS` environment variable to be set to allow Graphics Analyzer to trace all Vulkan function calls in the application.

2.5.6 Uninstall Graphics Analyzer

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

2.6 Chrome OS

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

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

This section contains the following subsections:

2.6.1 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.

2.6.2 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.

Tracing an Android application on Chrome OS is mostly the same as for other unrooted Android devices:

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:

   ```
   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 *3.2 Tracing Android applications* on page 3-34.

2.6.3 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 *3.3 Tracing Linux devices and IP address of target devices* on page 3-37.
7. Launch a new SSH session.
8. Run your Linux application and preload the interceptor library by defining the `LD_PRELOAD` environment variable to point at this library. For example:

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

Note

- If you are unable to use `LD_PRELOAD` on your system, see *7.1 Intercepting without using LD_PRELOAD* on page 7-103 for an alternative method.
- If you have more than one version of your graphics driver on your system and are having issues, see *7.2 Multiple drivers installed on the system* on page 7-104 for more information.

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

2.6.4 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 reboot, 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 *2.6.3 Trace a Linux application on Chrome OS* on page 2-24.
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:

```
sudo stop ui
```

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

```
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
```

Note

If you are unable to use `LD_PRELOAD` on your system, see *7.1 Intercepting without using LD_PRELOAD* on page 7-103 for an alternative method.

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 Chrome does not launch correctly, disconnect the Graphics Analyzer client and launch Chrome again. After it has launched, connect Graphics Analyzer and trace Chrome.

2.7 webOS

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

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

This section contains the following subsections:

2.7.1 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 "{}"
```

2.7.2 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 *1.2 Installation package* on page 1-14.

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.

2.7.3 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`

2.7.4 Trace a QML application

`/usr/bin/qml-runner` interprets QML applications.

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 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.

Tip

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`.

2.7.5 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. The following steps change this executable so that the application 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.

Tip

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

Example 2-1 webOS main menu (com.webos.app.home)

This application 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:

```
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:

```
mv /usr/bin/ga/libinterceptor.so /usr/bin/ga/libinterceptor.so.removed
```

2.8 Troubleshooting

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

This section contains the following subsection:

2.8.1 No trace is visible

The interceptor component on the target reports through `logcat` on Android. If no trace is found, then Arm recommends that you review the `logcat` trace.

In general, ensure the following:

- On Linux, ensure that the interceptor library is in your `PRELOAD` path.
- 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 or Vulkan calls.

Chapter 3
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 extra content.

It contains the following sections:

3.1 Open Graphics Analyzer

Follow these instructions to open Graphics Analyzer.

- On Windows, click **Start**, expand the studio folder, then select Graphics Analyzer.
- On Linux, run the command:

```
<install_directory>/gui/aga &
```

- On macOS, press Cmd+Space, type `Graphics Analyzer`, then press Enter.

To load one of the supplied sample traces, select **File** > **Open**, or click the **Open**  icon, and navigate to `<install_directory>/samples/traces/`. The various application windows fill with information from the loaded trace which you can examine.

3.2 Tracing Android applications

Connecting to your Android devices is automated through the **Device Manager**. To set up your device for tracing applications, the **Device Manager** detects any Android devices that are connected to the host. It then automatically installs the Graphics Analyzer components when the tracing session starts.

Prerequisites

- Connect your target device to a USB port on your host machine.
- Enable *Developer Mode*, then enable USB debugging on the target device, by selecting **Settings** > **Developer options**.
- Ensure you have installed a debuggable build of your application.

Procedure

1. Open Graphics Analyzer.
2. Click **Open the Device Manager**  .
3. In the **Android** tab, choose which Android device you want to connect to.
4. Choose the application that you want to trace from the list of debuggable packages.

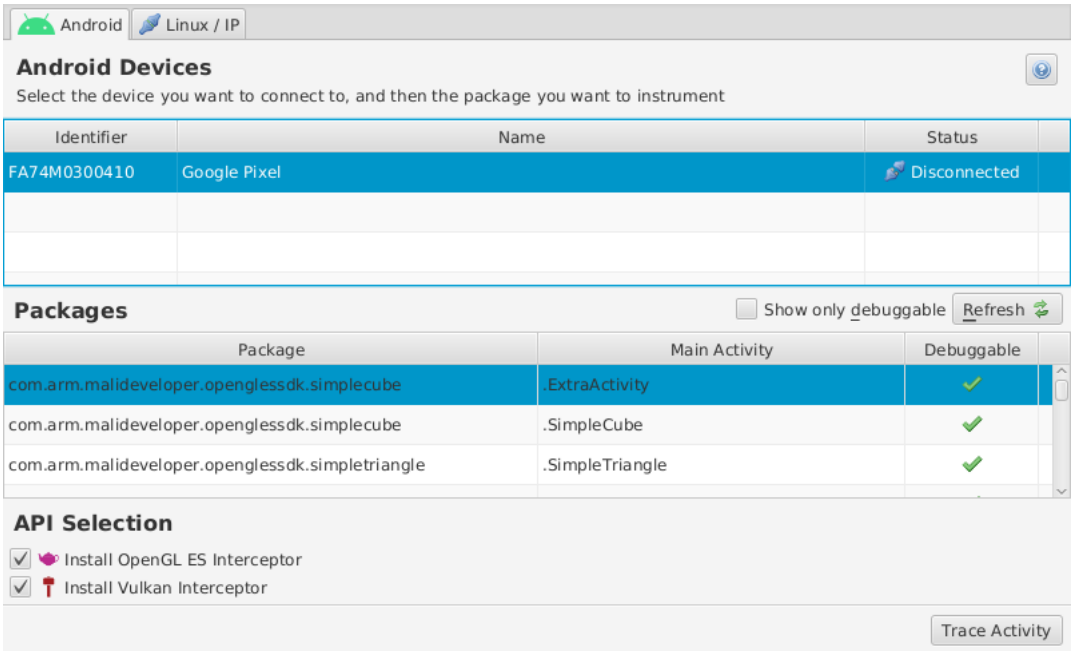


Figure 3-1 Connect to your Android device in the Device Manager

5. Select the required APIs.

   Note

   Installing both interceptors can affect performance of your application. If you experience such issues, try only installing one interceptor.

6. Click **Trace Activity** for the **Device Manager** to install layers and start the daemon automatically. **Results:** Graphics Analyzer connects to your device and installs the layer driver and daemon application that it uses to communicate with it. When the connection is established, the **Device Manager** closes and the live trace is shown.
7. Optionally, select a preset configuration, or choose which API assets are captured.

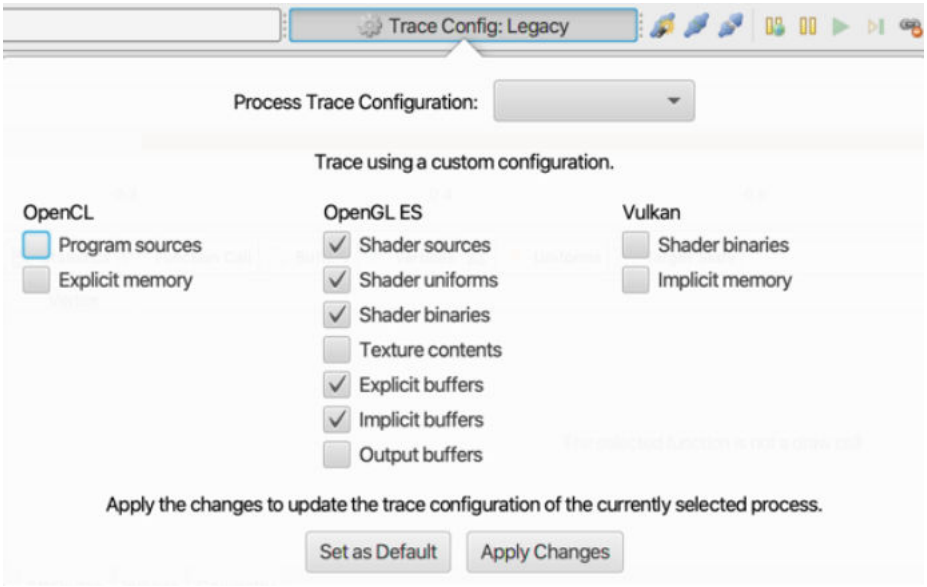


Figure 3-2 Trace Config dialog

As you enable more asset types, the application runs slower, more memory is required, and the generated trace file is larger. See *3.4 Configure tracing assets* on page 3-40 for more information.

8. Perform your test scenario on the device. Graphics Analyzer displays the trace data it receives from the device.
9. When you see a problem area in the trace data:
   - Use the pause, step, and play buttons to locate a frame that you want to analyze more closely. See *3.5 Pause, step frames, and resume* on page 3-41.
   - To capture the frame buffer output at the current frame, click the camera icon. See *3.6 Capturing frame buffer content* on page 3-42.
   - Capture extra frame data by enabling overdraw, shader map, or fragment count modes, then click the camera icon to collect the data. Learn more about these modes in *Chapter 4 Analyzing your trace* on page 4-44.
10. When you are ready to stop tracing, click **Stop tracing**.
    Results: The frames are listed in the **Trace Outline** view. To identify the type of frame capture you performed, an icon is shown next to the frames when you have captured extra data.
11. To show only the frames where you have captured extra data, select the **Show Only Frames With Features Enabled** check box.

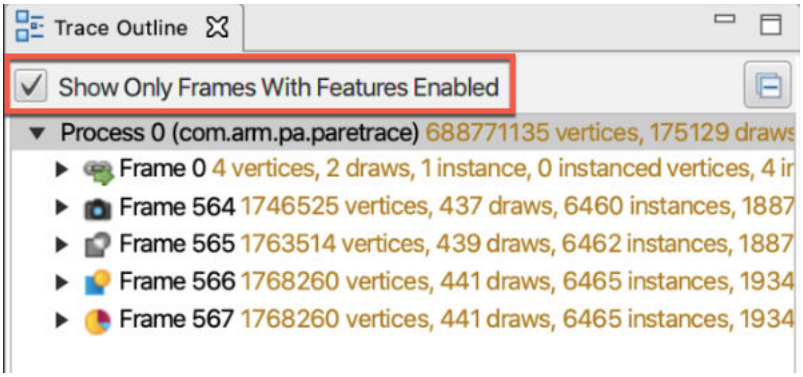


Figure 3-3 Show only frames with features enabled

12. Expand a frame to see the render passes and draw calls within it. Select frames, renderpasses, and draw calls to explore their data using the different data views. See *Chapter 5 The Graphics Analyzer interface* on page 5-56 for more information about the views.

Next Steps

To save or export the trace file, use the options under the **File** menu.

3.3 Tracing Linux devices and IP address of target devices

Connecting to a Linux device or IP address is automated through the **Device Manager**. The **Device Manager** detects any Linux devices on the network that has `aga-daemon` running on it. When a connection is established, Graphics Analyzer installs the layer driver and daemon application that it uses to communicate with it ready to trace your application.

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

Prerequisites

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

- A running OpenGL ES, 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.

Procedure

1. Open Graphics Analyzer.
2. Click **Open the Device Manager**  .
3. Select the **Linux/IP** tab.
   Results: The **Device Manager** automatically shows any Linux devices on the same local network and subnet as the host that are running the `aga-daemon` application. Alternatively, you can connect to the IP address of the device.
4. Connect to your device:

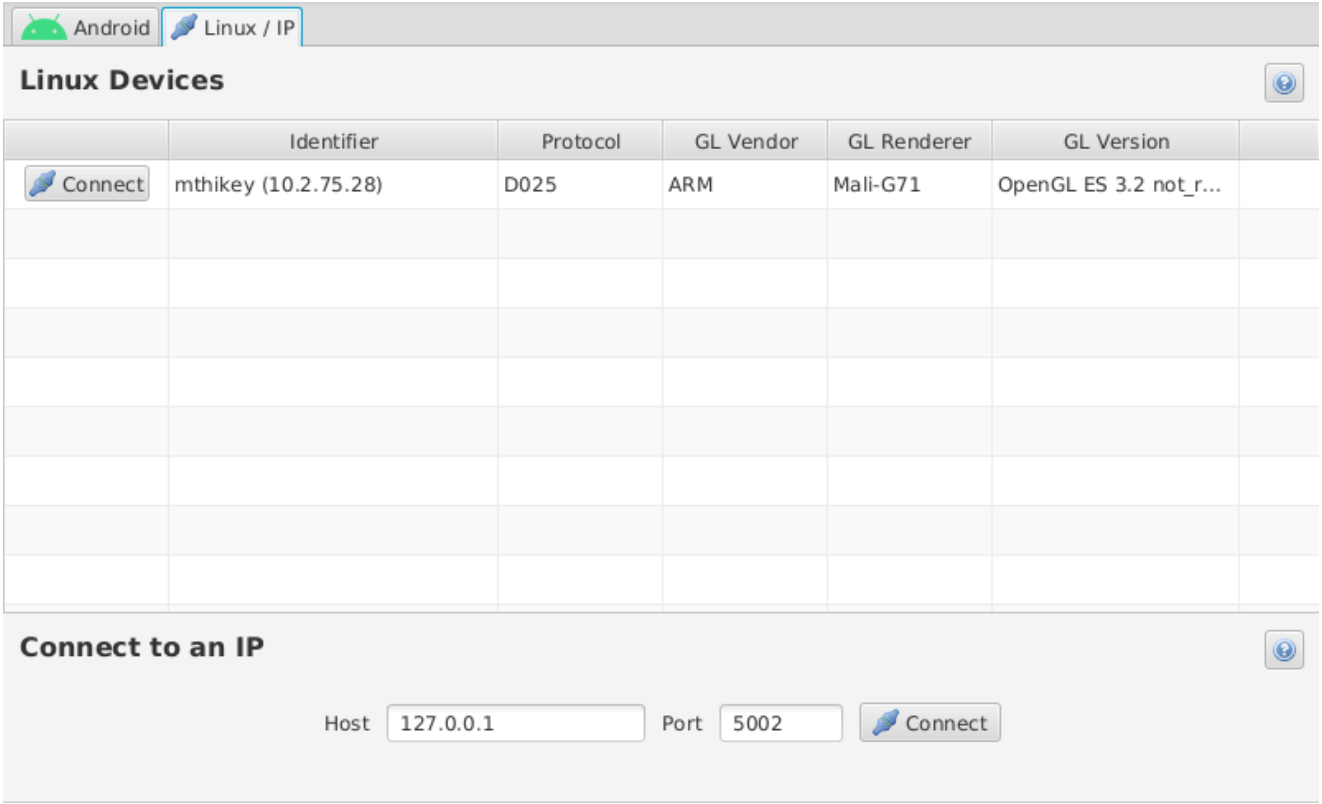


Figure 3-4 The Device Manager dialog

- Select a Linux device. If no Linux devices have been detected, you can directly connect to the device using the IP address instead.
- Enter the IP address of the device, and the port that the daemon is running on. Then click **Connect**.

Note

— The default port is 5002.

— The **Device Manager** obtains the GL Vendor, GL Renderer, and GL Version strings of the device by dynamically loading the `libGLESv2` library. If this library could not be found or loaded, the strings appear as `Unknown`.

Results: Graphics Analyzer connects to your device and installs the layer driver and daemon application that it uses to communicate with it. When the connection is established, the **Device Manager** closes and the live trace is shown.

5. Optionally, select a preset configuration, or choose which API assets are captured.

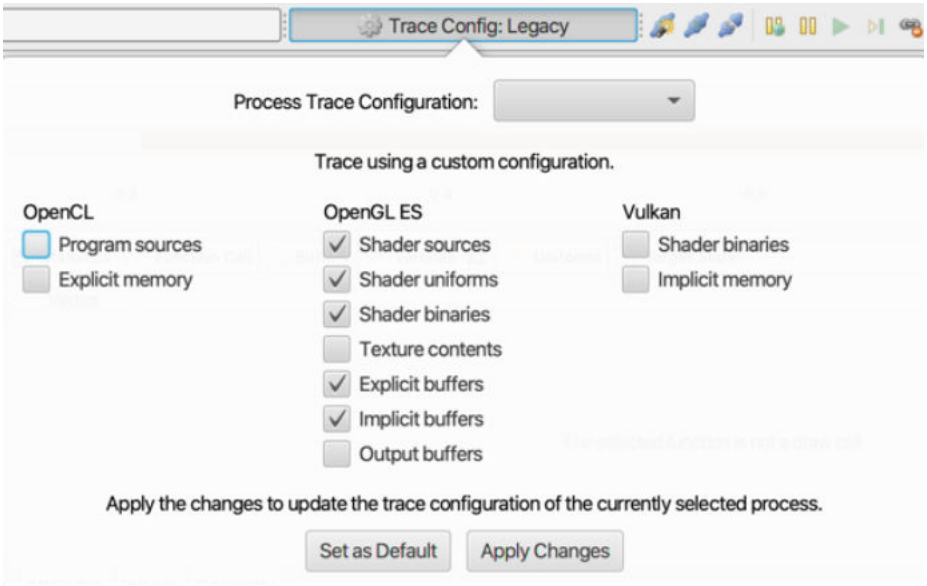


Figure 3-5 Trace Config dialog

The more asset types you enable, the slower the application runs, the more memory is required, and the larger the generated trace file is. See *3.4 Configure tracing assets* on page 3-40 for more information.

6. Perform your test scenario on the device. Graphics Analyzer displays the trace data it receives from the device.
7. When you see a problem area in the trace data:
   - Use the pause, step, and play buttons to locate a frame that you want to analyze more closely. See *3.5 Pause, step frames, and resume* on page 3-41.
   - Click the camera icon to capture the frame buffer output at the current frame. See *3.6 Capturing frame buffer content* on page 3-42.
   - Capture extra frame data by enabling Overdraw, Shader map or Fragment count modes, then click the camera icon to collect the data. Learn more about these modes in *Chapter 4 Analyzing your trace* on page 4-44.
8. When you are ready to stop tracing, click **Stop tracing**.
   Results: The frames are listed in the **Trace Outline** view. To identify the type of frame capture you performed, an icon is shown next to the frames when you have captured extra data.
9. To show only the frames where you have captured extra data, select the **Show Only Frames With Features Enabled** check box.

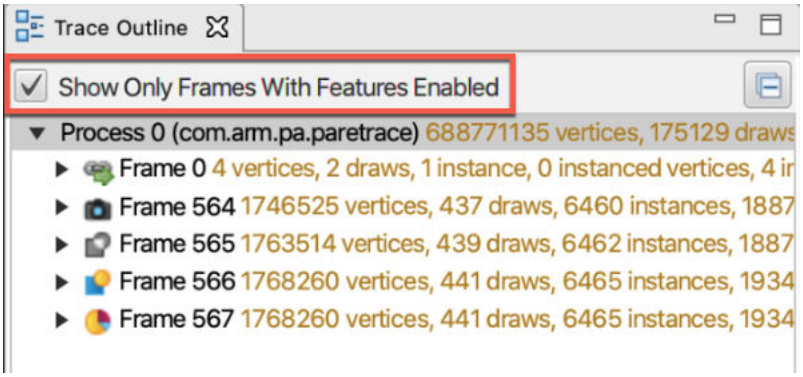


Figure 3-6 Show only frames with features enabled

10. Expand a frame to see the render passes and draw calls within it. Select frames, renderpasses, and draw calls to explore their data using the different data views. See *Chapter 5 The Graphics Analyzer interface* on page 5-56 for more information about the views.

Next Steps

To save or export the trace file, use the options under the **File** menu.

3.4 Configure tracing assets

When you have established a connection to a target device, configure which types of API assets are sent to the host when API function calls are traced.

For example, if you do not want to examine the contents of the textures and buffers being used in your OpenGL ES application, use the **Trace Config** dialog to disable these asset types and speed up tracing of your application.

The more types of assets you enable, the more information is visible in Graphics Analyzer. However, the application being traced runs more slowly, the generated trace file is larger, and Graphics Analyzer uses more memory.

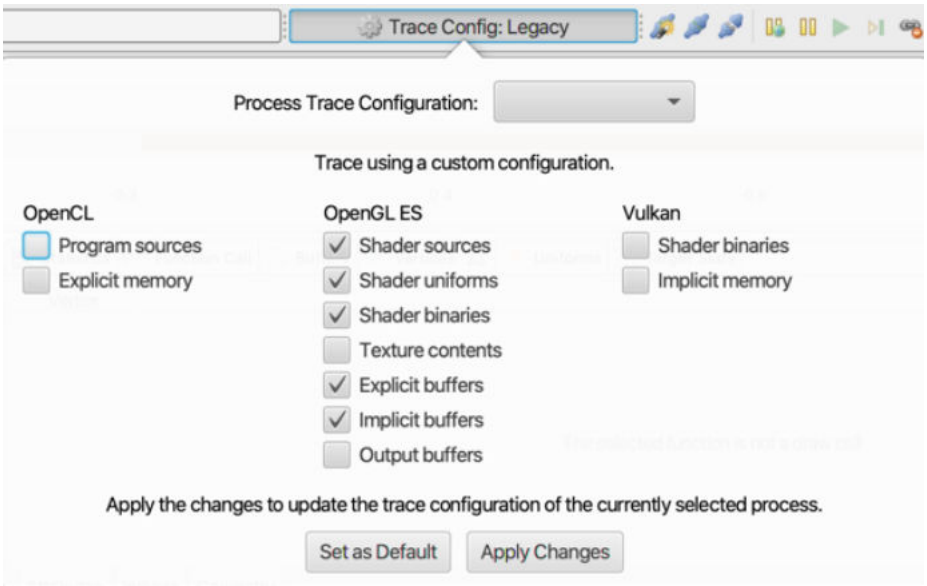


Figure 3-7 Trace Config dialog

You can change the configuration at any time, even after the process has started. Any function calls traced in the application after the new configuration has been set will use the new configuration.

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

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

Note

Use the **Legacy** configuration to send assets that are equivalent to the default types that were sent in versions of Graphics Analyzer before the **Trace Config** dialog existed.

A distinction is made in the **Trace Config** dialog between explicit and implicit memory:

- Explicit memory is memory that a function call uploads explicitly. 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 the application can then modify.

3.5 Pause, step frames, and resume

Graphics Analyzer displays the trace data as it receives it from the device. When you get to a problem area, use the pause, step and play buttons to locate a frame that you want to analyze more closely.

To pause the currently selected process, press the  button. The process is stopped 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.

To pause all connected processes, press the  button. The processes are stopped at the next `eglSwapBuffers()` call to allow you to examine the result.

When a process is paused, you can render individual frames for the selected process by stepping with the  button.

To resume the selected process, press the  button.

Note

Only the threads that are calling the graphics API functions are paused.

3.6 Capturing frame buffer content

To see the on-screen output sent to the frame buffer, capture frame data of the output from the graphics system on a draw-by-draw basis.

Procedure

1. To take a snapshot of the frame buffer content following each draw call in the next full frame of the application, click **Capture** .

   Note

   Taking this snapshot involves considerable work in composing and transferring data, which slows down the application.

2. Open the **Trace Outline** view. Any frames that include extra captured data are shown with an icon, to identify the type of frame capture you performed. Any regular frame now has a icon to show that it is a captured frame.

   Note

   Some limitations exist for frame capture of Vulkan applications. For more information, see *5.11 Vulkan Frame Capture view* on page 5-71.

3. Optionally, you can use special capture modes to capture extra frame data, such as frame buffer attachments, or to modify the frame buffer content in some way. To enable a capture mode, toggle the capture mode icon in the Graphics Analyzer toolbar, then click **Capture** to trigger the frame capture.
4. To stop tracing, click **Stop tracing** .

3.7 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, click  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. The draw calls for the four shapes have different values set for the stencil buffer write mask, with the stencil pass operation set to `GL_REPLACE`.

Figure 3-8 Color attachment

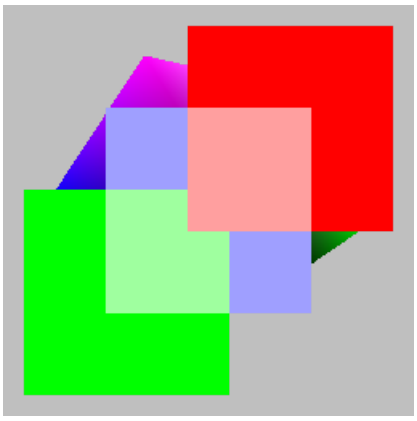


Figure 3-9 Depth attachment

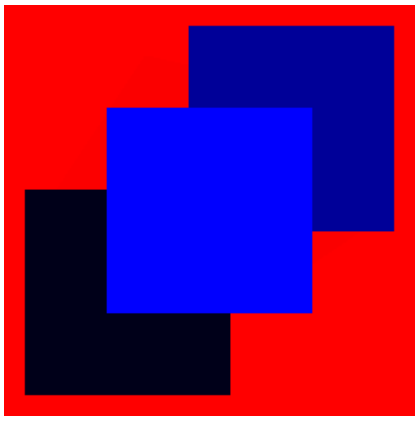


Figure 3-10 Stencil attachment

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.

Limitations on capturing all frame buffer attachments

- It is not possible to capture the depth or stencil attachments for FBO 0 on an OpenGL ES 2.0-only configuration.
- Only a low-resolution capture of the depth buffer is possible when:
  - The configuration does not support depth texture sampling.
  - The device only has OpenGL ES 2.0 available and the depth attachment is a renderbuffer.

Chapter 4
Analyzing your trace

Learn about the different ways you can analyze your trace in more detail.

It contains the following sections:

4.1 Analyzing overdraw

Overdraw occurs in graphics applications where scenes are built using multiple layers of objects that overlap, and are rendered in a back-to-front order. High levels of overdraw can cause poor performance on some devices, where pixels are unnecessarily shaded multiple times. Graphics Analyzer shows the level of overdraw in a scene, to help you diagnose rendering order issues, and find opportunities to optimize performance.

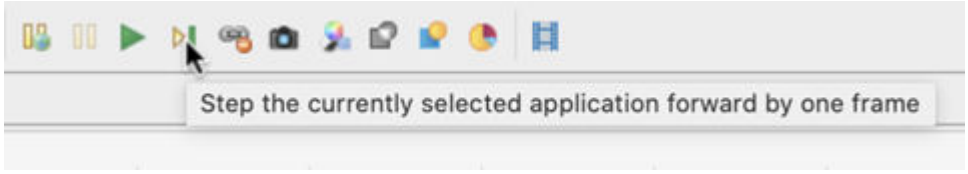
When Graphics Analyzer captures a frame with overdraw mode enabled, the fragment shader in the target application is replaced with an almost transparent white fragment shader. Each time a pixel is rendered to the frame buffer, the alpha value is increased using an additive blend. Therefore, as more overdraw happens in an area, the whiter the final image appears.

Use overdraw mode to check that opaque objects are being rendered in a front-to-back order. This order enables Mali GPUs to use early ZS testing to disable fragment shading for objects that are hidden by objects that are closer to the camera, or by stencil masks.

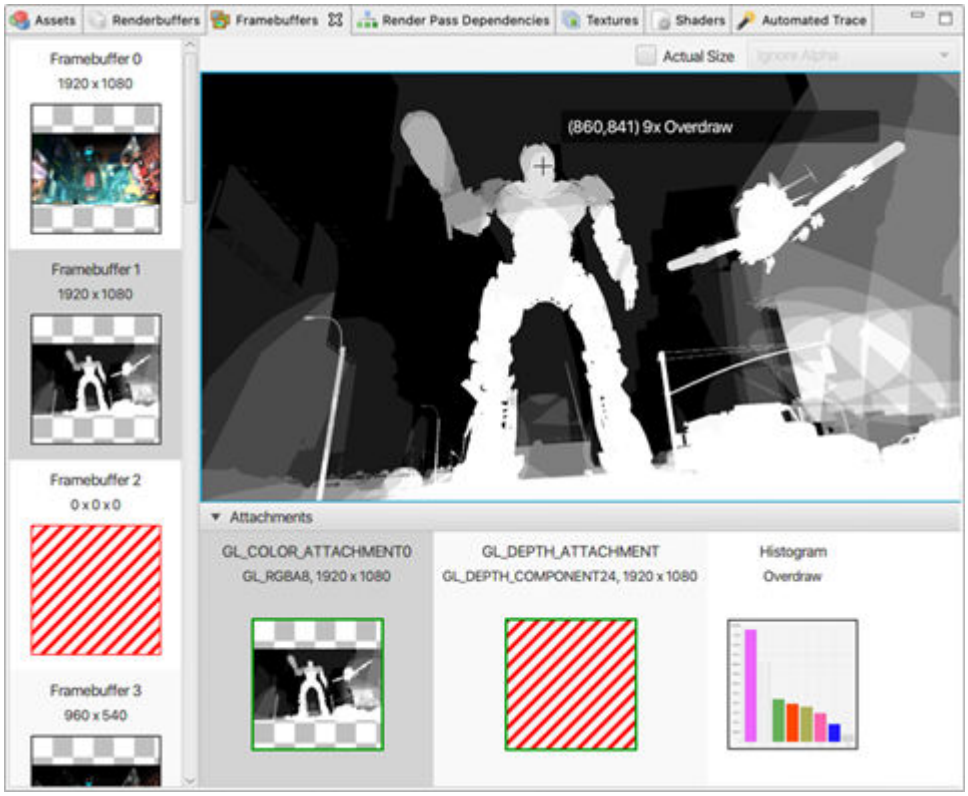
If the scene uses transparency, some level of overdraw is expected, as objects must be rendered back-to-front. Use transparency carefully in mobile applications, as the performance cost of shading multiple layers can be significant.

Procedure

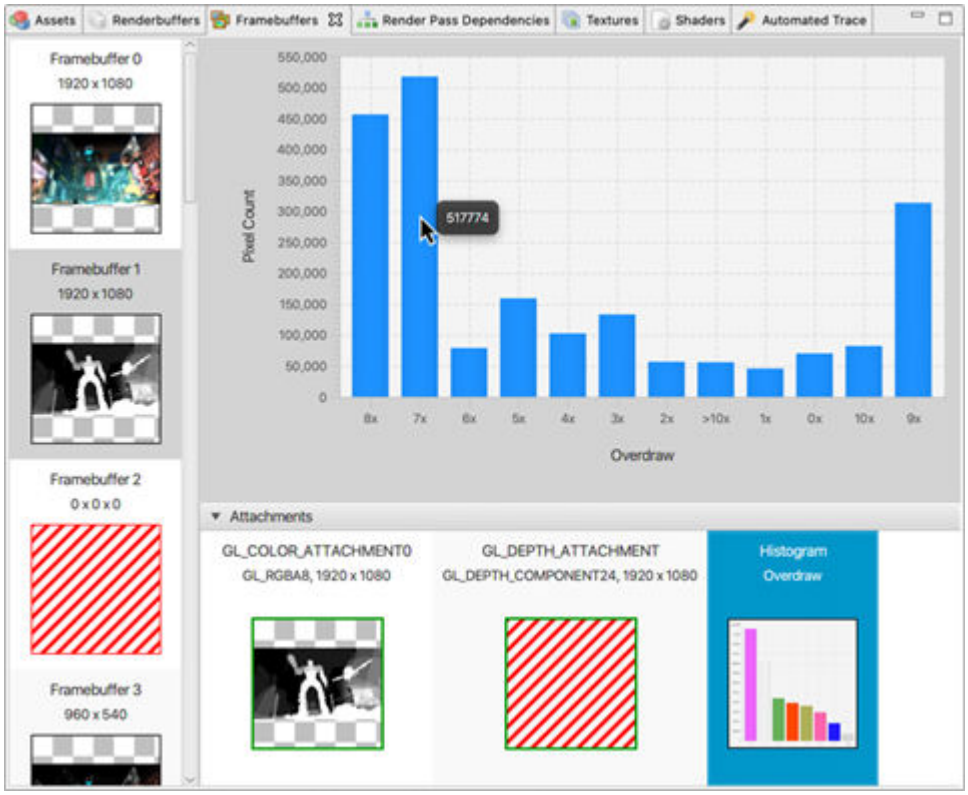
1. Connect Graphics Analyzer to the application on your device as described in *Get Started with Graphics Analyzer*, and start capturing a trace.
2. Perform your test scenario in the application on the device. To locate the right frame, just before you get to the scene you want to analyze, use the pause and step buttons.



3. Click the **Toggle Overdraw Mode** icon to enable overdraw mode, then click the **Capture** icon to capture the next frame.
4. To stop tracing click the **Stop tracing** icon.
   Results: The frames are listed in the **Trace Outline** view. Any frames where you have captured overdraw are shown with the icon.
5. Expand a frame to see the renderpasses and draw calls within it. Step through the draw calls and evaluate the frame buffer output to see how each drawn object affects the level of overdraw in the scene. Hover over different areas of the scene to see the level of overdraw at that point.



6. View the histogram to see the number of pixels being shaded at each level of overdraw in the scene.



Next Steps

When you have identified an area in the scene with high overdraw, refer to the following topics to get advice on how to reduce it.

- For optimization advice for high overdraw on the Arm Developer website, see *Optimization advice for high overdraw*.
- For blending advice in the Mali GPU best practises guide, see *Blending advice for Mali GPUs*.

4.2 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 4-1 Original image

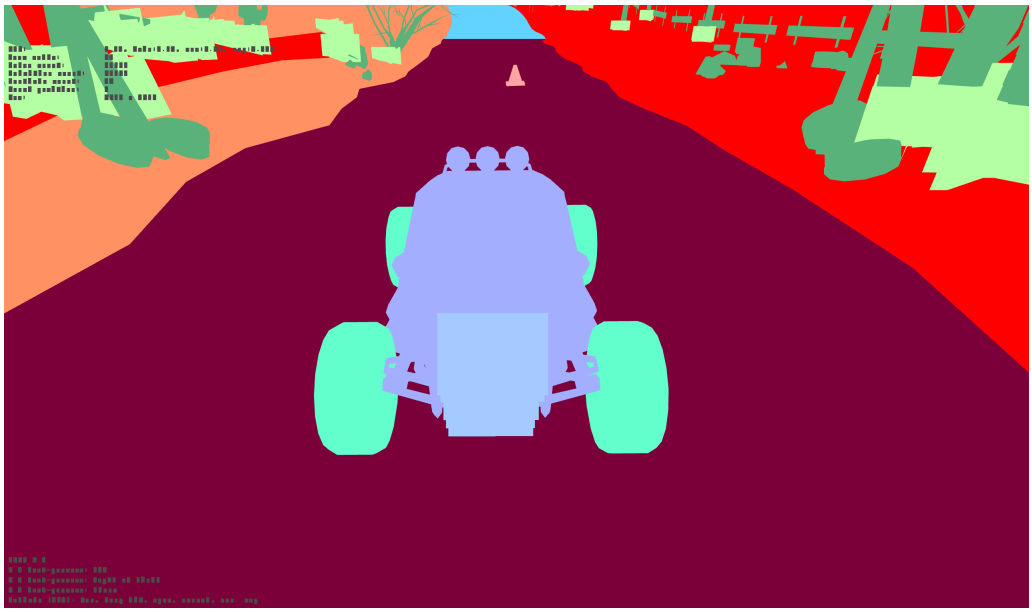


Figure 4-2 Image with shader map feature turned on

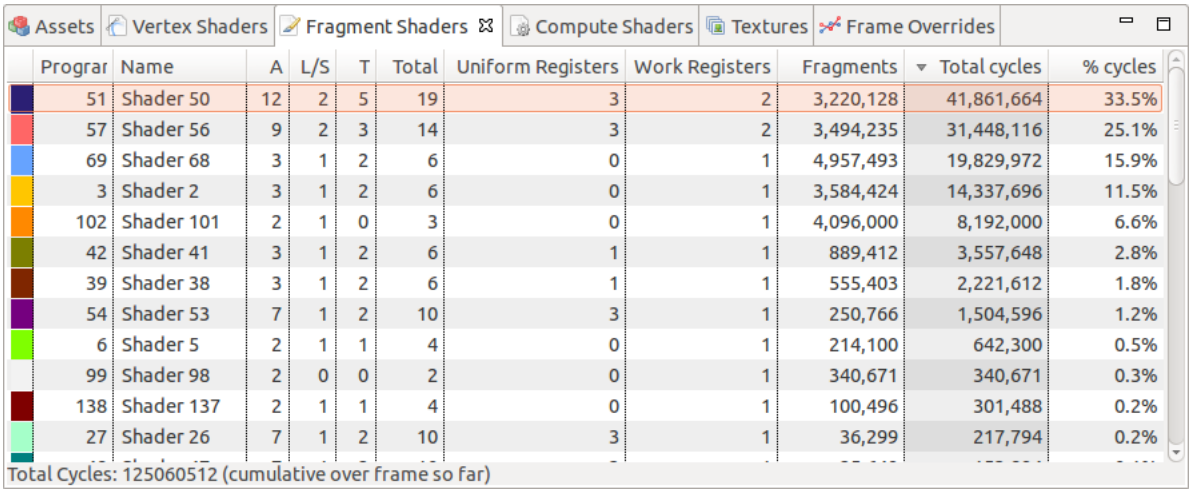
There are 100 unique colors that Graphics Analyzer can assign to shader programs, after which programs have duplicate colors. You can identify which program corresponds to each color by putting the cursor on a frame buffer image that was captured in shader map mode. The active shader is identified above the image.

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

4.3 Analyzing the fragment count

Graphics Analyzer can count the number of fragments that a shader processes 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 . 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:



Prograr	Name	A	L/S	T	Total	Uniform Registers	Work Registers	Fragments	Total cycles	% cycles
51	Shader 50	12	2	5	19	3	2	3,220,128	41,861,664	33.5%
57	Shader 56	9	2	3	14	3	2	3,494,235	31,448,116	25.1%
69	Shader 68	3	1	2	6	0	1	4,957,493	19,829,972	15.9%
3	Shader 2	3	1	2	6	0	1	3,584,424	14,337,696	11.5%
102	Shader 101	2	1	0	3	0	1	4,096,000	8,192,000	6.6%
42	Shader 41	3	1	2	6	1	1	889,412	3,557,648	2.8%
39	Shader 38	3	1	2	6	1	1	555,403	2,221,612	1.8%
54	Shader 53	7	1	2	10	3	1	250,766	1,504,596	1.2%
6	Shader 5	2	1	1	4	0	1	214,100	642,300	0.5%
99	Shader 98	2	0	0	2	0	1	340,671	340,671	0.3%
138	Shader 137	2	1	1	4	0	1	100,496	301,488	0.2%
27	Shader 26	7	1	2	10	3	1	36,299	217,794	0.2%

Figure 4-3 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.
- 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.

4.4 Debugging an OpenCL application

Graphics Analyzer supports tracing OpenCL applications on Linux, 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 views that are supported for OpenCL. For more information, see *5.1 Perspectives* on page 5-58.

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. Therefore 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.

4.5 Using GPUVerify to validate OpenCL kernels

GPUVerify is a tool that can 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. Click **Edit** > **Preferences** and fill in the **Path to GPUVerify** field, as shown here:

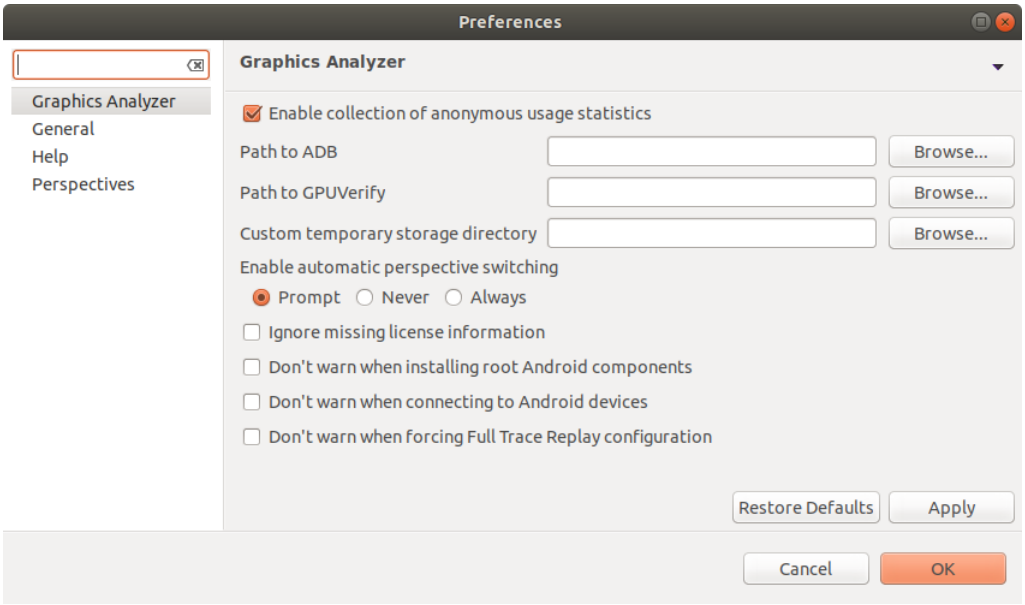


Figure 4-4 Preferences dialog

If you enter an invalid path in this field, **Apply** and **OK** are grayed out.

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 the OpenCL kernels.
- Any run-time parameters that are known at this point, for example the local and global workgroup sizes.

To run any of the kernels that were found in a trace through GPUVerify, select a function call in the trace with live kernel objects, then select **Debug** > **Launch GPUVerify**. You are presented with the following dialog:

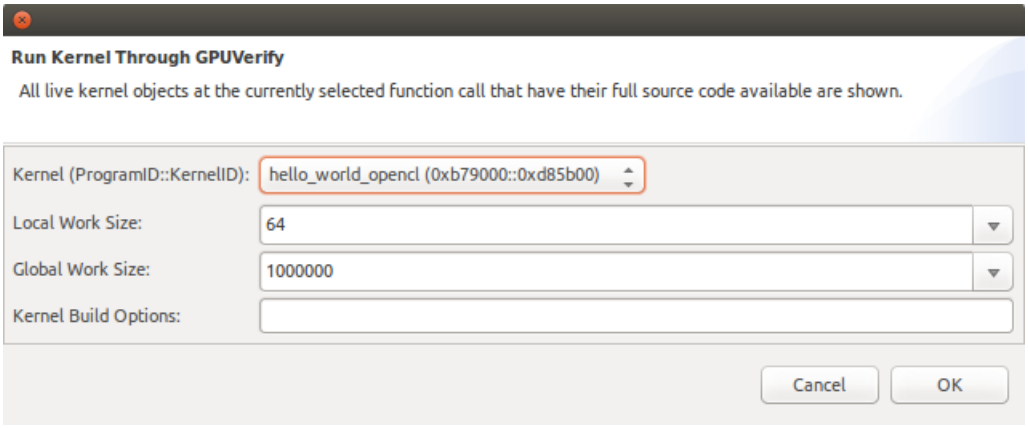


Figure 4-5 Run kernel through GPUVerify dialog

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. You can pick the local and global work sizes from the drop-down lists, which are populated from the enqueue history of the kernel. Alternatively enter them manually as 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.

4.6 Comparing state between function calls

When investigating an application trace, you can generate a difference report to compare the API state between two function calls to examine what has changed.

Procedure

1. To compare changes in state between two functions in the trace, use the **Ctrl** key (Windows or Linux hosts) or **Cmd** key (OS X hosts) to either:
   - In the **Trace** view, select two function calls.
   - In the **Trace Outline** view, select two draw calls.
2. Right-click on the selected calls, then select **Generate Diff** from the popup menu.
   Results:
   The report shows the difference between states for the two selected function calls. It is a table of the items that are different, or have changed at some point between the two functions.

	Original value	Current value	Related functions
GL_CURRENT_PROGRAM	0	1	182, 170, 153
GL_ELEMENT_ARRAY_BUFFER_BINDING	0	0	89, 79, 63, 59
GL_VIEWPORT	[0, 0, width, height]	[0, 0, 2560, 1504]	180
GL_TEXTURE_BINDING_2D_ARRAY	0	0	326, 282, 276, 194, 107... and 1 more function
GL_PIXEL_UNPACK_BUFFER_BINDING	0	0	278, 188, 119, 115, 111
GL_ACTIVE_TEXTURE	GL_TEXTURE0	GL_TEXTURE0	280
GL_COLOR_CLEAR_VALUE	[0, 0, 0, 0]	[0.5, 0.5, 0.5, 1.0]	172
GL_ARRAY_BUFFER_BINDING	0	0	87, 77, 55, 51
GL_UNIFORM_BUFFER_BINDING	0	0	322, 284, 71, 67
GL_CULL_FACE	GL_FALSE	GL_TRUE	178
GL_DEPTH_TEST	GL_FALSE	GL_TRUE	176
EGL_current_display	<unknown>	0x74720ac8	46
EGL_current_read_surface	<unknown>	0x769ce8b8	46
EGL_current_context	<unknown>	0x75544fd8	46
EGL_current_draw_surface	<unknown>	0x769ce8b8	46

Figure 4-6 Difference Report view

   - Values that have changed but have since reverted to the original value are highlighted in light blue.
   - 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, any subcomponents that have not changed are shown with gray text.
3. The final column in the report, labeled **Related functions**, lists the numerical id of function calls in the trace that modified the particular state item. Right-click on one of the rows, then select **Jump to related function** and select a related function from the context menu.
4. To manually compare sets of changes, you can open multiple difference reports at a time. The results of the state comparison are persistent until the window closes.

4.7 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 *5.20 Bookmarks view* on page 5-86 and the *5.2 Trace view* on page 5-59.

Chapter 5
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:

5.1 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. This perspective 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.

5.2 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. It is not how much time the GPU spends doing the work that the function call requests.

  ———————————

- The list of arguments sent to the call.

  ————— **Note** —————

  This list is truncated to save space. For a complete list of the arguments, see *5.6 Function Call view* on page 5-65.

  ———————————

- 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 *5.2.1 Add a bookmark to a function call* on page 5-60.

————— **Note** —————

Some columns in this table might initially be hidden. To enable or disable columns, click  .

———————————

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. A call to `eglSwapBuffers()` delimits each frame. 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 *4.6 Comparing state between function calls* on page 4-54. 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. Click  and  to jump between the search results.

To close the search and hide the results, click  or press Esc.

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 *5.22 Filtering and searching in Graphics Analyzer* on page 5-89 for more information.

This section contains the following subsection:

5.2.1 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 *4.7 Bookmarks* on page 4-55.

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.

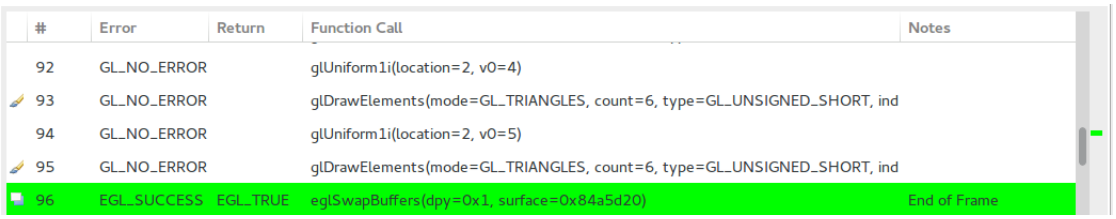


Figure 5-1 Bookmarked function call with a note

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

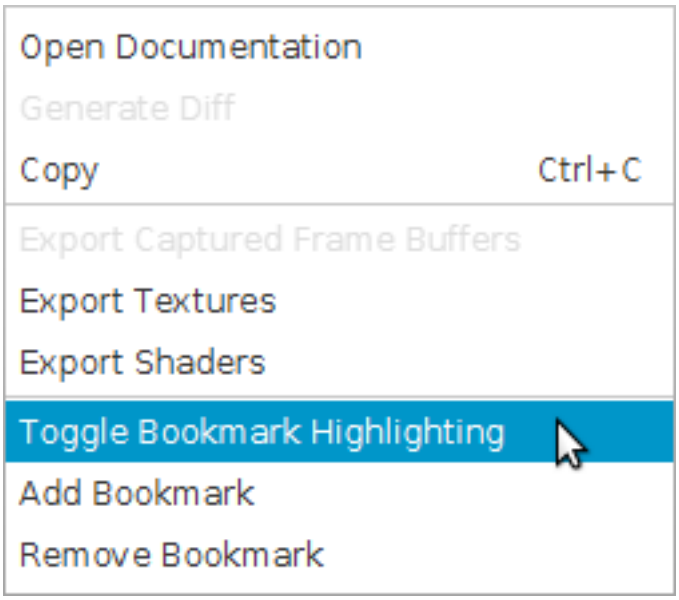


Figure 5-2 Toggling bookmark highlighting

View and manipulate all bookmarks in the *5.20 Bookmarks view* on page 5-86.

Note

To remove a bookmark, right-click on the function call and select **Remove Bookmark**.

5.3 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. Selecting a process causes the Trace view to only display 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 indicates where the item would be.

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
- Fragment Count
- Overdraw Mode
- Shader Map Mode

To collapse all the items in the tree, click  **Collapse All**.

Right-clicking on items in the tree allows you to generate a diff report between two items and export frame buffers. See *4.6 Comparing state between function calls* on page 4-54 and *5.12.1 Exporting assets* on page 5-73 for more information.

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

5.4 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 view shows you a high-level view of API function call activity for each process that was traced.
  - Select a process by clicking it.
  - Select the first frame of a process by double-clicking the process.
- When a frame or render pass has been selected, the Render Pass Timeline view is shown. This view shows you each context in the selected process and each render pass in each context.
  - Select a render pass by clicking it.
  - Select the first draw call in a render pass by double-clicking the 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.
  - Select the first draw call in a render pass by clicking the render pass.
  - Select a draw call by clicking it.

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

- Scroll the X-axis by holding down the left mouse button and dragging left or right.
- Zoom in or out of the X-axis by scrolling up or down using the scroll wheel. Alternatively hold down the right mouse button and drag up or down.
- Display a tooltip showing context-sensitive information about an element, including the axes, by hovering over the element in the chart.

5.5 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 parent of the item 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 a slice selects that item in the trace.
- Double-clicking a slice selects the first child of that item 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 Graphics Analyzer to process this data, turn off the SELinux permissions by running `setenforce 0` on your device. If this command produces an error, try the following command instead:

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

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

  ```
  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 a slice displays information about the memory contained in that trace and the percentage of total memory used in that frame.
- Double-clicking 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.

5.6 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 tab can be useful because, due to the limited size of the *5.2 Trace view* on page 5-59, 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`. As 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.

5.7 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. To gain a more detailed view of the problem, if available, hover your cursor over the problem report.

5.8 Target State view

This view shows the state of the underlying system at a time following the function selected in the Trace view. It updates 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 *5.1 Perspectives* on page 5-58.

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

If a state item has been changed anywhere in the trace, 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 *5.22 Filtering and searching in Graphics Analyzer* on page 5-89 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.

5.9 Buffers view

This view shows information about the currently allocated buffer objects.

You can filter the list of buffer objects by usage, or, for OpenGL ES 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 *5.22 Filtering and searching in Graphics Analyzer* on page 5-89 for more information.

5.10 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 *3.6 Capturing frame buffer content* on page 3-42 for more information about capturing frames.

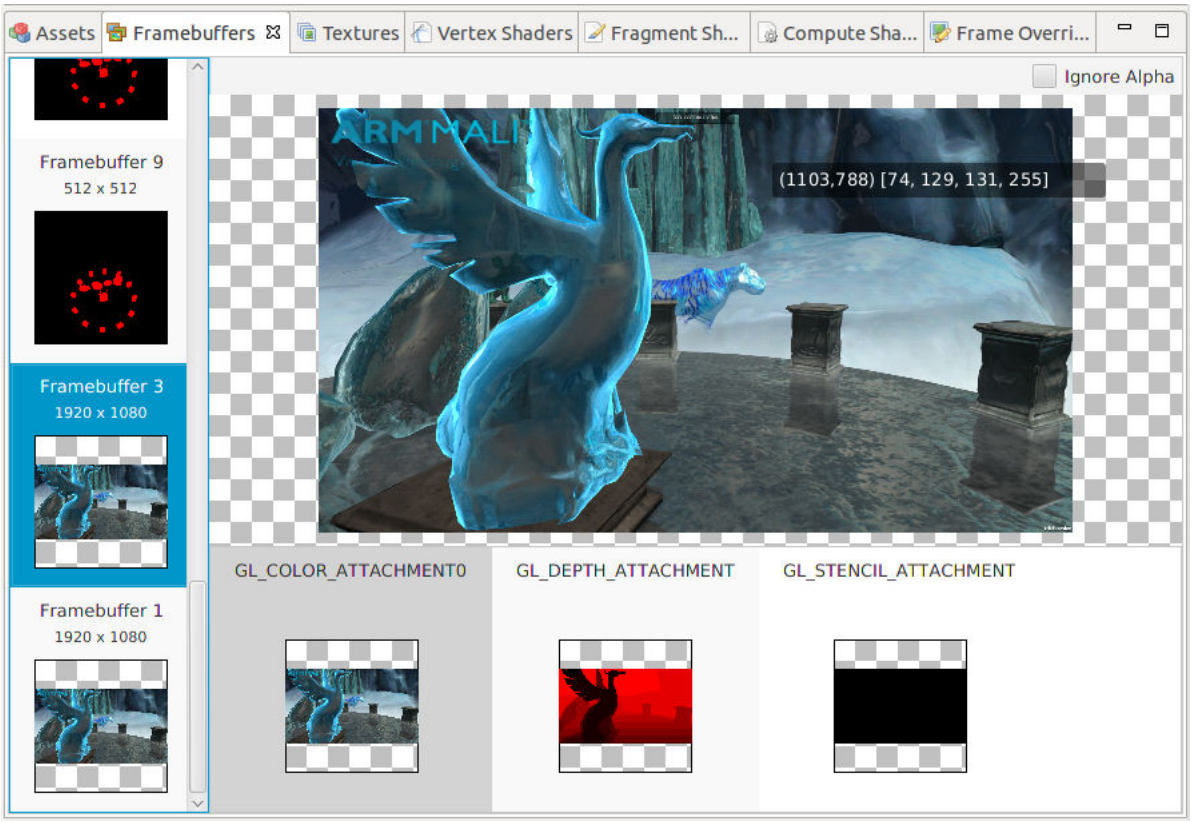


Figure 5-3 Selecting a framebuffer in OpenGL ES Framebuffers view

Selecting a frame buffer in the left-hand list brings up a list of the attachments used in that frame buffer in the lower list. If relevant, it also brings up histograms displaying the overdraw or shader map data. To bring up a larger view of an attachment, select it in the lower list. 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 or fully transparent) to white (maximum alpha or fully opaque).

For a captured frame, you can step through the sequence of draw calls one at a time and observe how the final scene is constructed.

5.11 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 *3.6 Capturing frame buffer content* on page 3-42 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. These 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.

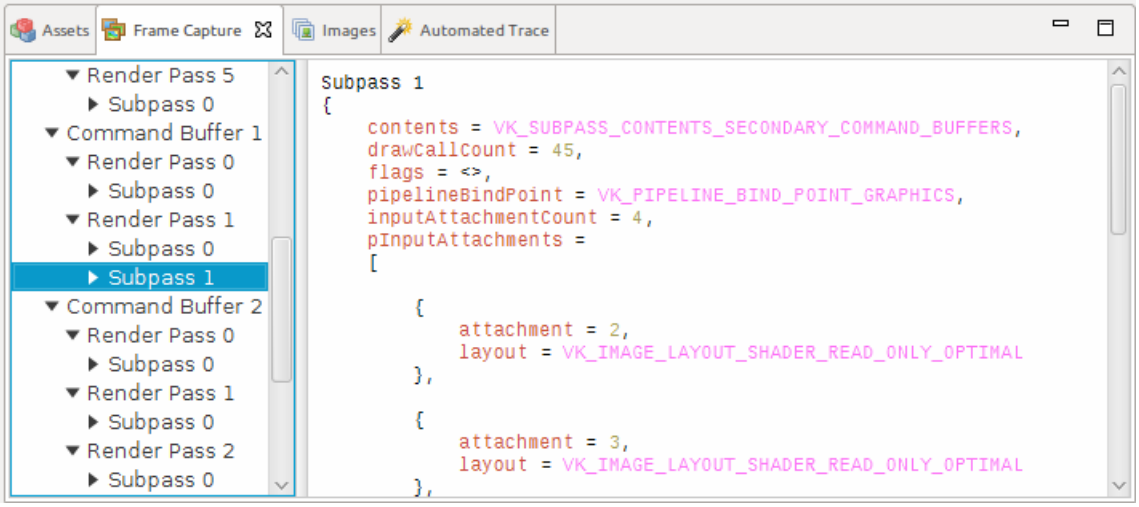


Figure 5-4 Selecting a subpass in Vulkan Frame Capture view

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.

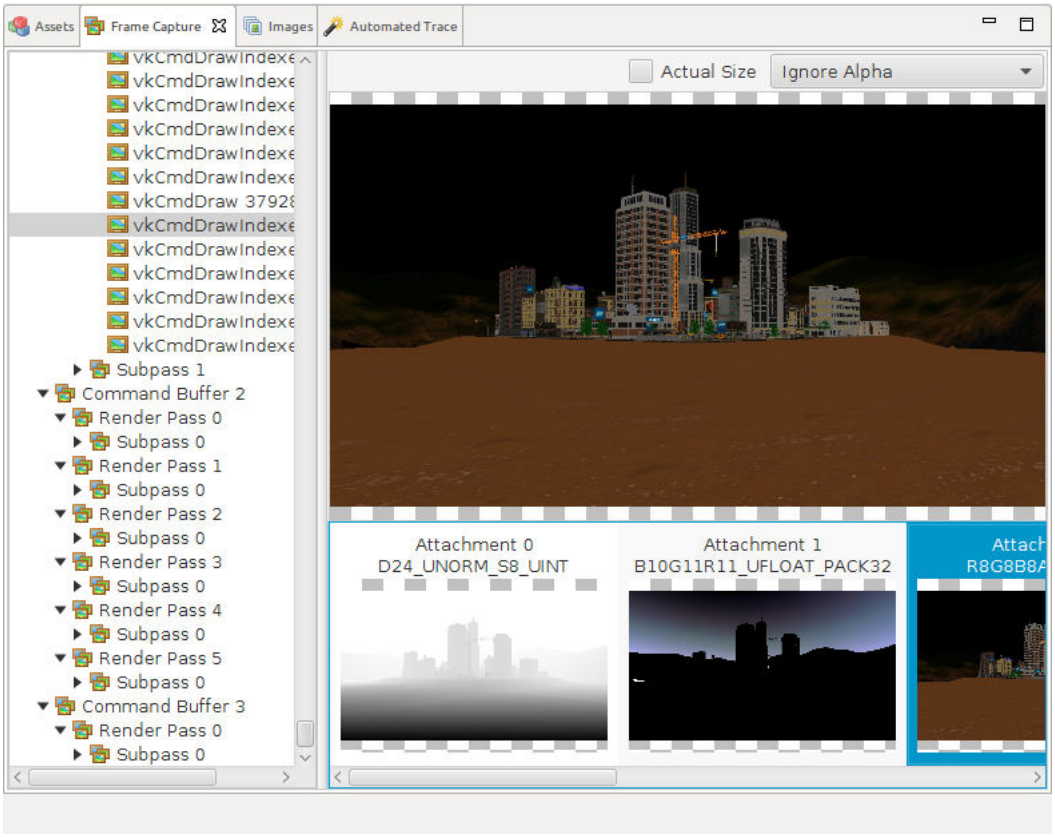


Figure 5-5 Selecting a draw call in Vulkan Frame Capture view

The  icon is displayed next to tree items when frame buffer attachment data has been received. You can view captured frame buffer attachments when the host receives the data. It is therefore not necessary to wait until the **Capturing frame** dialog completes 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 or fully transparent) to white (maximum alpha or 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. First the minimum and maximum floating-point values in the entire image are calculated for each channel. Then the values for each channel are scaled to 0-255 according to these calculated minimum and maximum values. The tone mapper ignores channels that have a value of 0. These channels remain at 0 in the displayed image.

5.12 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.

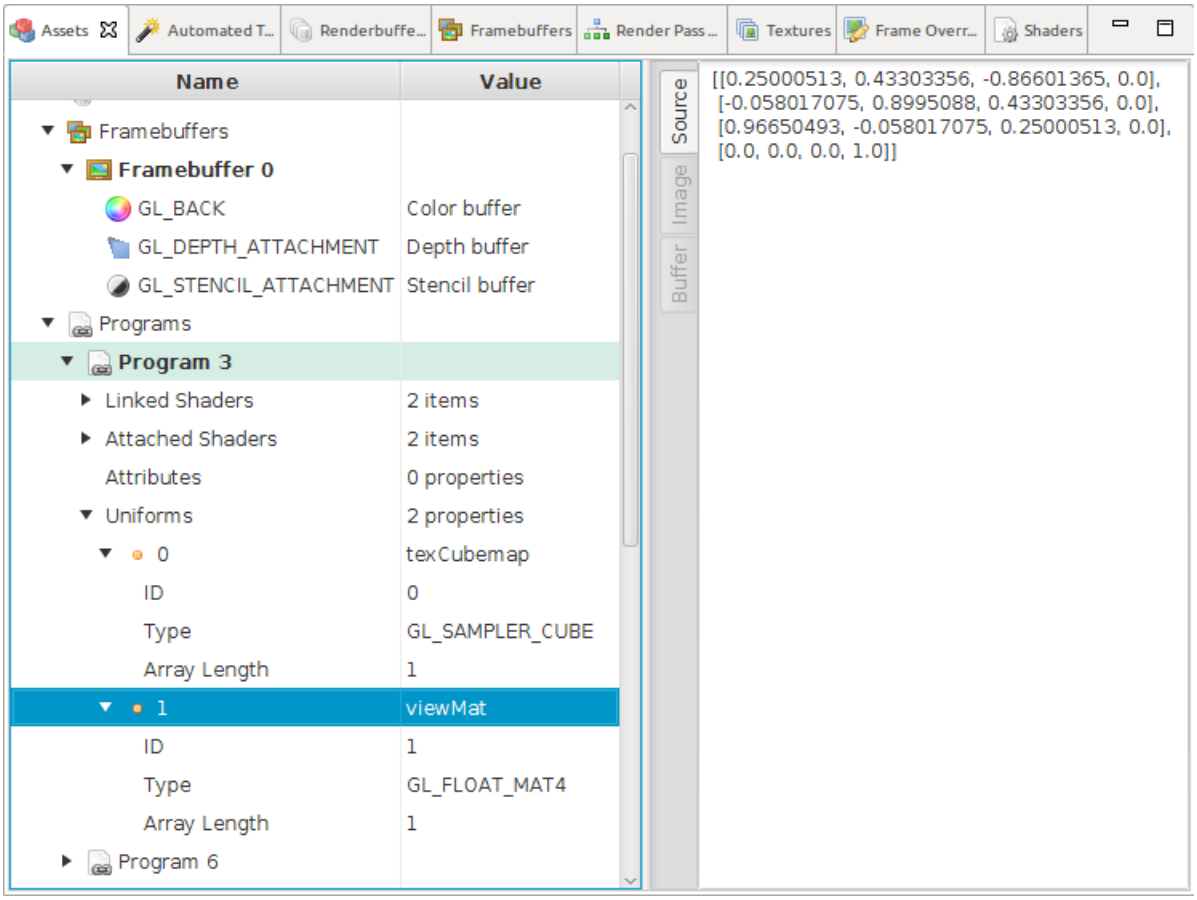


Figure 5-6 Assets view

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:

- *5.12.1 Exporting assets* on page 5-73.

5.12.1 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**.

5.13 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-T880 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. For example, 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 *4.3 Analyzing the fragment count* on page 4-50 for more information.

Active shaders are shown in bold type.

5.14 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_RGBA` with type `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT_4_4_4_4`
- `GL_RGB` with type `GL_UNSIGNED_BYTE`
- `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 or fully transparent) to white (maximum alpha or fully opaque).

5.15 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 or fully transparent) to white (maximum alpha or fully opaque).

5.16 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:

5.16.1 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.

5.16.2 Indices tab

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

5.16.3 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 was drawn and also to inspect the geometry for defects. You can see if the geometry is incorrect due to missing or unexpected extra vertices. Or you can see if the geometry is too dense, which might lead to performance problems.

If you use the Geometry view with the Framebuffers view, you can see where in the scene the geometry was drawn. Seeing the position of the geometry 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 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 the attribute 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. Alternatively, to move left and right press A and D, and to move up and down press Q and E.

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. Most 3D model editors and viewers can load these files. 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.

———————————

5.17 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.

5.18 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

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:

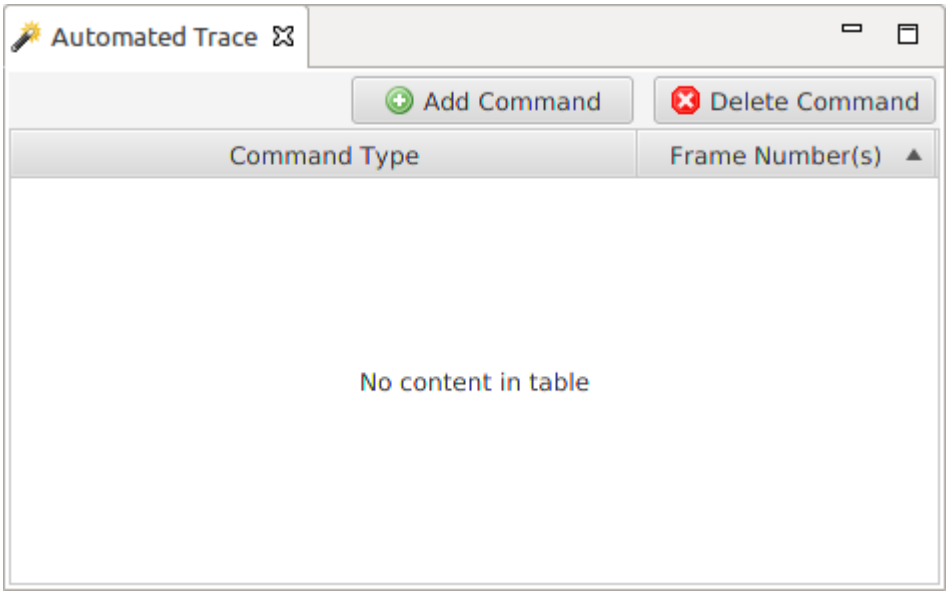


Figure 5-7 Automated Trace view

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

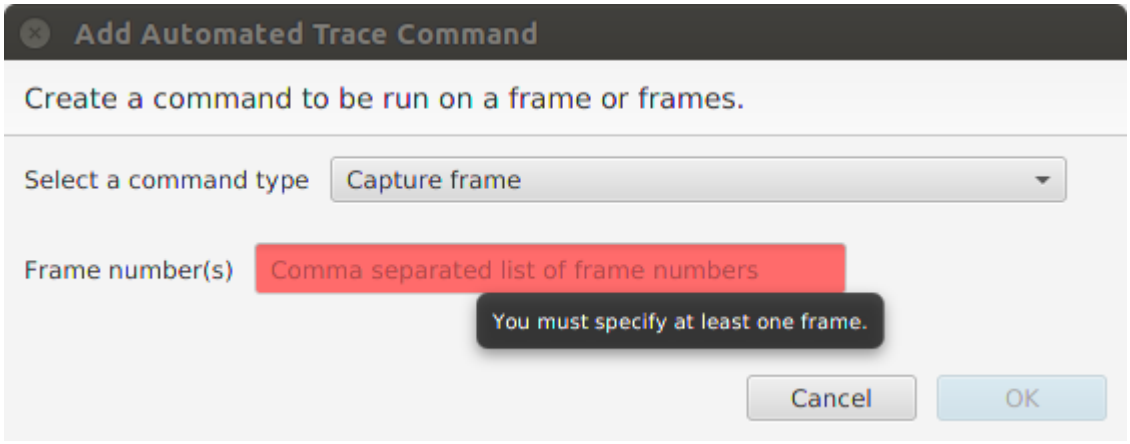


Figure 5-8 Add Automated Trace Command dialog

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:

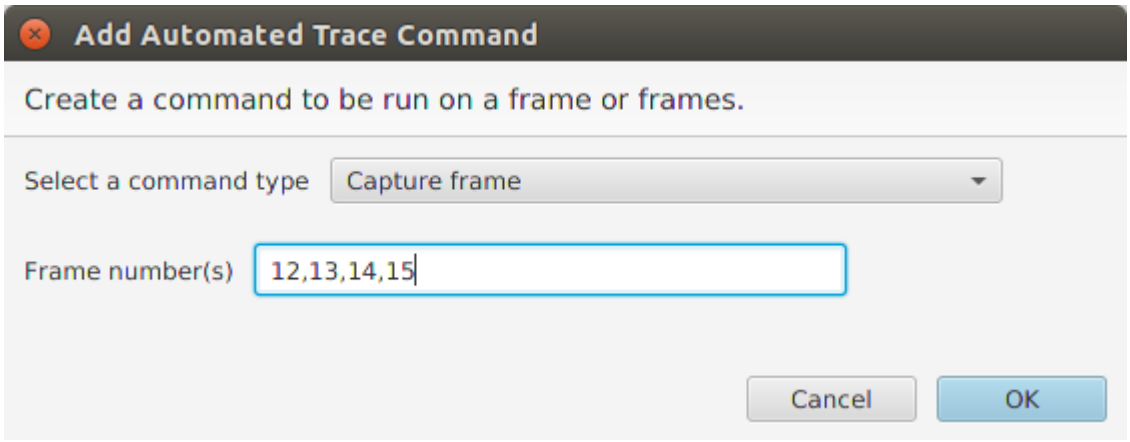


Figure 5-9 Specifying a list of frame numbers

You can then add more commands and remove existing ones:

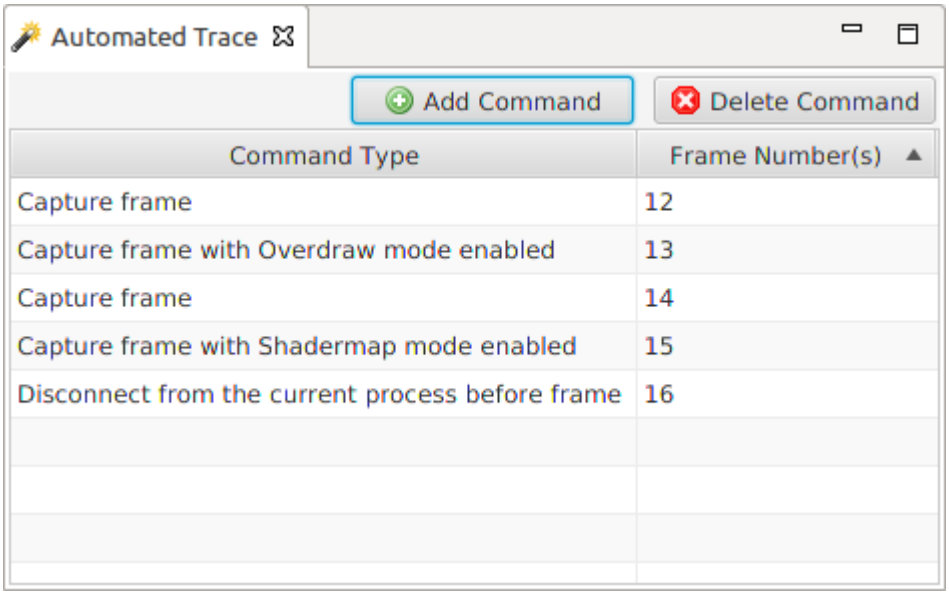


Figure 5-10 Selecting command types

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

If you send a play, step, or capture command in a frame, or in the frame before it, automated trace commands for that frame are ignored.

5.19 Render Pass Dependencies view

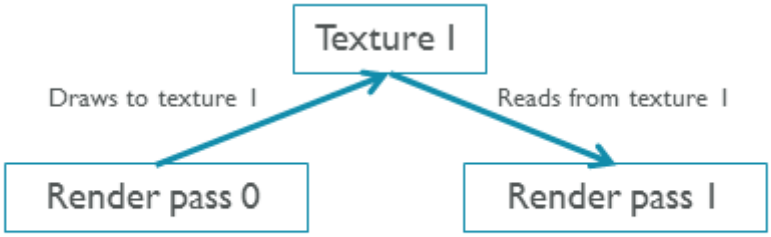
Graphics Analyzer can work out what dependencies there are between different render passes in a selected frame.

Note

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

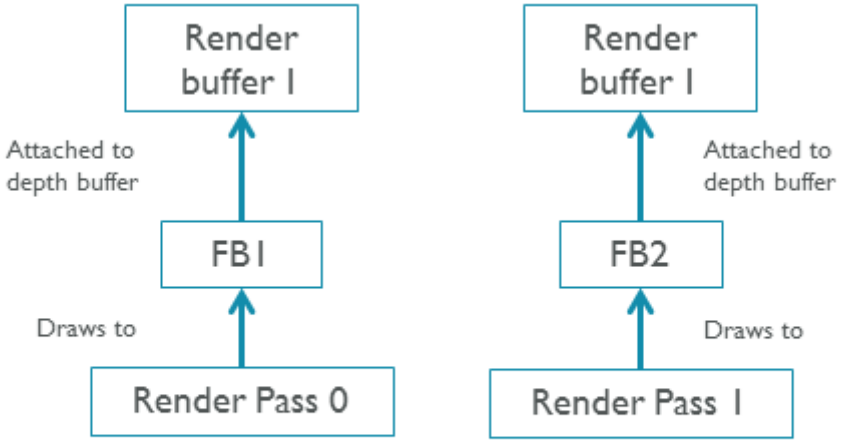
The different types of dependencies it can detect are:

- 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:



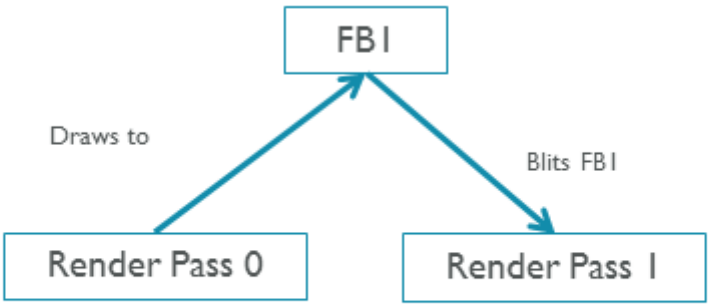
Render pass 1 has a dependency on render pass 0 due to texture 1

- 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, both render pass 0 and render pass 1 have render buffer 1 attached to their depth target:



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:



Render pass 1 has a dependency on render pass 0 due to a blit of FB1

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 depends on another render pass is shown in the tree. Expanding the render pass tells you:

- Which render pass it depends 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 depends on Render Pass 26 in the *previous* frame (Frame 0). Render Pass 4 depends 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:

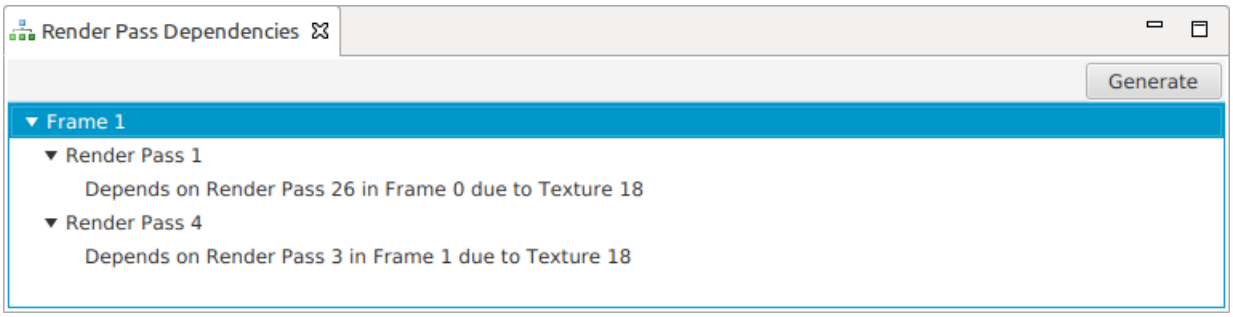


Figure 5-11 Render Pass Dependencies view showing dependencies

5.20 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 *4.7 Bookmarks* on page 4-55.

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.

5.21 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.

5.22 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`. To anchor your expressions, 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*

5.23 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 *5.24 Target-side headless mode* on page 5-92.

Note

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

This section contains the following subsections:

5.23.1 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 export assets, run the `aga-headless` script from the Graphics Analyzer installation directory. Pass in the location of the trace file to export assets from. You must also provide the output directory with the `--export-output-directory` switch. This directory is used to write the exported shaders and source code, and must 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, which 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. Join 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 looks similar to this example, using Unix formatting convention:

```
aga-headless/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.

5.23.2 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 `aga-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. If it is your first time running headless mode, or if you need reminding how to configure the trace, use this method 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 2 Before you begin* on page 2-15.

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 file is where 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**, 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.

5.24 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 *5.23 Host-side headless mode* on page 5-90.

Note

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

This section contains the following subsections:

5.24.1 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 application loading the interceptor has the Android `WRITE_EXTERNAL_STORAGE` permission, or `/data/data/{package-name}` if the permission is not available.

Place the default configuration file in `$EXTERNAL_STORAGE/aga-headless.conf` if the application 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 application manifest contains the `WRITE_EXTERNAL_STORAGE` permission and that the application has that permission enabled in Settings. Without this permission, it is not possible to set the configuration file for the application or write the trace files, as internal application 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 these policies are implemented by checking if there are any error messages using `adb logcat -s audit`. You can solve the issue by disabling SELinux on your rooted device.

- The `WRITE_EXTERNAL_STORAGE` permission also implicitly grants `READ_EXTERNAL_STORAGE`, so if this permission is available, the interceptor always looks for the config file in `$EXTERNAL_STORAGE/aga-headless.conf`.
- On most devices `$EXTERNAL_STORAGE` is `/sdcard`.

Linux

Trace files are saved to the `$HOME/traces/` directory by default.

Place the configuration file in `$HOME/aga-headless.conf`.

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

5.24.2 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 5-1 Configuration file keys reference

Key	Required?	Type	Accepted values	Description	Default value
`name`	Yes	String	A valid process name, or package name on Android.	Name of the process.	None
`config`	No	String	`fullTrace`, `everything`, `balanced`, `functionsOnly`, `legacy`	Preset config name.	`legacy`
`customConfig`	No	Object	A valid `customConfig` object, see *Custom config reference* on page 5-93.	Custom configuration of resources per API.	None. Use preset as-is with no changes.
`traceDirectory`	No	String	A valid filesystem path.	Directory used for creating headless trace files.	None. Use default, see *5.24.1 File locations* on page 5-92.
`frameCaptures`	No	Object	A valid `frameCaptures` object, see *Frame captures reference* on page 5-94.	Allows setting frame captures frame numbers and modes.	None
`disconnectBeforeFrame`	No	Integer	A valid frame number.	Number of the frame to disconnect and disable tracing before.	None

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 5-95.

Note

Custom configuration resource toggles are always applied after the preset. A preset can therefore 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 5-2 customConfig object keys

Key	Type
`cl`	OpenCL config object. See *OpenCL config object reference* on page 5-94.
`gles`	OpenGL ES config object. See *OpenGL ES config object reference* on page 5-94.
`vulkan`	Vulkan config object. See *Vulkan config object reference* on page 5-94.

OpenCL config object reference

Key	Type
`programSources`	Boolean
`explicitMemory`	Boolean

OpenGL ES config object reference

Key	Type
`shaderSources`	Boolean
`shaderUniforms`	Boolean
`shaderBinaries`	Boolean
`textureContents`	Boolean
`explicitBuffers`	Boolean
`implicitBuffers`	Boolean
`outputBuffers`	Boolean

Vulkan config object reference

Key	Type
`shaderBinaries`	Boolean
`implicitMemory`	Boolean

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 5-95.

Table 5-3 frameCaptures object keys

Key	Type
`default`	Integer array
`overdraw`	Integer array
`fragmentCount`	Integer array
`shaderMap`	Integer array
`allAttachments`	Integer array

Example configuration file

This topic shows an example of a headless configuration file.

```
{
  "processes": [
    {
      "name": "com.example.application",
      "customConfig": {
        "cl": {
          "programSources": false,
          "explicitMemory": false
        },
        "gles": {
          "shaderSources": false,
          "shaderUniforms": false,
          "shaderBinaries": 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/"
    }
  ]
}
```

5.24.3 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. The configuration file can contain multiple configurations for different processes.

The daemon accepts a series of arguments. For details, see *5.24.4 Arguments accepted by the daemon* on page 5-96. 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:

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

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

5.24.4 Arguments accepted by the daemon

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

Note

- If `--HeadlessMode` is specified, `--name` must also be specified, otherwise the daemon does not start.
- For explicit resource arguments, specified as 1 or 0, if the argument is not present, the resource setting from the preset is used.
- Comma-separated argument values have the format 1,[2…] with no spaces. For example, 1,2,4,8.

Table 5-4 Arguments for starting the daemon

Argument	Description	Default value	Required?
`--HeadlessMode`	Start the daemon in headless mode.	`false`	No, but required for the other options to apply.
`--name/-n {`*`process_name`*`}`	Name of the process.	None	Yes, if `--HeadlessMode` is present.
`--config/-c {`*`preset_name`*`}`	Config preset. See accepted values for the `config` key in *5.24.2 Headless configuration file reference* on page 5-93.	`legacy`	No
`--traceDirectory/-o {`*`directory`*`}`	Output directory for trace files.	None. Uses the default, see *5.24.1 File locations* on page 5-92.	No
`--disconnectBeforeFrame/-d {`*`frame_number`*`}`	Frame number before which to stop tracing.	None	No

Table 5-4 Arguments for starting the daemon (continued)

Argument	Description	Default value	Required?
`--frameCaptures.[CaptureMode] {comma-separated_list_of_frame_numbers }`	Frame captures for the given mode, where `[CaptureMode]` is one of the frame capture keys. See *Frame captures reference* on page 5-94), given as a comma-separated list.	None	No
`--cl.[Resource] {1/0}`	Explicitly enable (1) or disable (0) a resource, where `[Resource]` is one of the CL resource keys. See *OpenCL config object reference* on page 5-94.	None	No
`--gles.[Resource] {1/0}`	Explicitly enable (1) or disable (0) a resource, where `[Resource]` is one of the OpenGL ES resource keys. See *OpenGL ES config object reference* on page 5-94.	None	No
`--vulkan.[Resource] {1/0}`	Explicitly enable (1) or disable (0) a resource, where `[Resource]` is one of the Vulkan resource keys. See *Vulkan config object reference* on page 5-94.	None	No

Chapter 6
Integration with Arm Streamline

The Graphics Analyzer interceptor library generates Streamline annotations and chart information.

When profiling an application with 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:

6.1 Installation

Install 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 Streamline.

To set up Graphics Analyzer, follow the instructions in *Chapter 2 Before you begin* on page 2-15.

6.2 Using Streamline annotations

The details panel and charts in Streamline display a range of extra information.

Charts

The interceptor provides five charts that track draw calls, frame rate, and vertices.

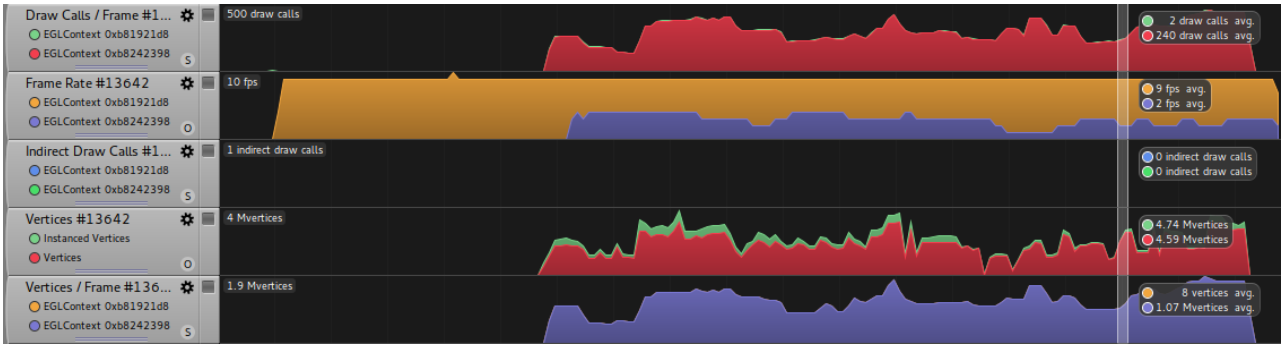


Figure 6-1 Charts in Streamline.

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.

Figure 6-2 Graphics Analyzer Activity in the details panel.

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.

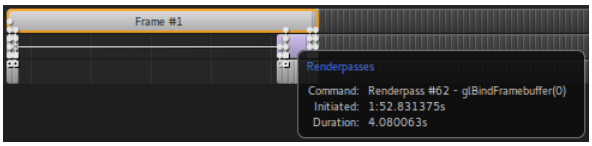


Figure 6-3 Tooltip displaying render pass information

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.

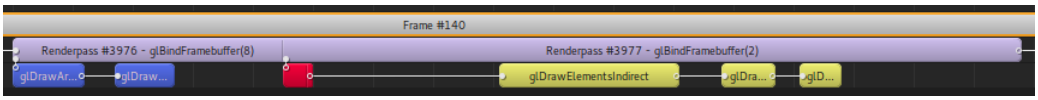


Figure 6-4 Color coded API calls.

API Call Marker Colors

Red	Flushing calls such as `glFlush`
Green	End of frame calls such as `eglSwapBuffers`
Blue	Direct draw commands such as `glDrawArrays`
Yellow	Indirect draw commands such as `glDrawArraysIndirect`

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.

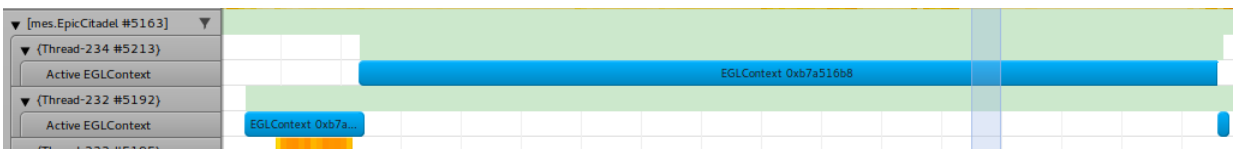


Figure 6-5 Active EGL contexts in the details panel.

Chapter 7
Known issues

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

It contains the following sections:

7.1 Intercepting without using LD_PRELOAD

Sometimes it might not be possible to use `LD_PRELOAD`, for example if `LD_PRELOAD` is already being used for another purpose.

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

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.

7.2 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.

Note

Non-Android support is only available in Arm Development Studio, and requires a valid license. See *Arm Development Studio Editions* for more information.

If so, it might not be possible to use the standard `LD_PRELOAD` approach on its own.

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 *7.1 Intercepting without using LD_PRELOAD* on page 7-103 for more information.

7.3 Application crashes while tracing

Graphics Analyzer uses shader replacement to instrument the application and provide debug visualizations of captured frames. To avoid the application using previously compiled and cached shader or program binaries, which cannot be instrumented, the installed Graphics Analyzer layer forces all calls that load previously compiled shaders and programs to fail.

This forced failure includes functions such as `glShaderBinary()` or `glProgramBinary()`. If an application does not handle these functions failing cleanly, falling back to a new source compilation, the application might fail to start or might render incorrectly.

You can work around this issue by doing one of the following, depending on the application:

- Clear the application storage cache, which clears the automatic caching on Android.
- Delete all application data, which clears any application managed caching.
- Uninstall and re-install the application, forcing a fresh compile on first load.

Note

Calls to `glShaderBinary()` or `glProgramBinary()` can fail at any time on a device, for example if a system update installs a newer graphics driver. Applications should therefore always fall back to compiling from source when binary loads fail.

Appendix A
Analytics

It contains the following sections:

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
- OpenGL 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.

  ————————————