

Mali Graphics Debugger v4.9.2

User Guide

© Arm Ltd. 2017

arm MALI

Visual Computing

Contents

Chapter 1: Introduction.....	7
Installation Package.....	8
Components.....	8
This Release.....	8
 Chapter 2: Minimum Requirements.....	 9
Host System Requirements.....	10
Target System Requirements.....	10
Increasing Available Memory.....	10
Temporary Storage.....	10
 Chapter 3: Host Installation.....	 11
Linux.....	12
Windows.....	12
Mac OS X.....	12
Target Deliverables.....	12
Licensing.....	13
 Chapter 4: Target Installation.....	 15
Linux.....	16
Prerequisites.....	16
Installation.....	16
Run Instructions.....	17
Tracing an OpenGL ES, EGL, or OpenCL Application.....	17
Tracing a Vulkan Application.....	17
Uninstalling.....	17
Android.....	17
Rooted Android.....	17
Unrooted Android.....	23
Tracing an Android Application.....	27
Chrome OS.....	27
Prerequisites.....	27
Tracing an Android Application on Chrome OS.....	28
Tracing a Linux Application on Chrome OS.....	28
Tracing Chrome on Chrome OS.....	29
Troubleshooting.....	29
Target Devices With No cp Support.....	29
No Trace Visible.....	30
Frame Capture Doesn't Work on Android 4.4 and Above.....	30
Socket Permission Errors.....	30
 Chapter 5: Device Manager.....	 31
Android Devices.....	32
Installing MGD Components.....	32
Uninstalling MGD Components.....	33
Connecting to Your Device.....	33
Using the ADB Tasks Log.....	33
Linux Devices.....	33
Connect to an IP.....	34

Chapter 6: Getting Started with MGD..... 35

Running the Host GUI.....	36
Using the Mali Graphics Debugger on Your Application.....	36
Tracing an Application.....	36
Process Configuration.....	36
Dealing With Multiple Processes.....	37
Pausing, Stepping Frames, and Resume.....	37
Capturing Framebuffer Content.....	38
Capturing All Framebuffer Attachments.....	38
Capturing All Framebuffer Attachments Limitations.....	39
Analyzing Overdraw.....	39
Analyzing the Shadermap.....	39
Overdraw and Shadermap Limitations.....	40
Analyzing the Fragment Count.....	40
Fragment Count Limitations.....	41
Replaying a Frame.....	41
Frame Replay Limitations.....	41
Frame Overrides.....	42
Debugging an OpenCL Application.....	45
Using GPUVerify to Validate OpenCL Kernels.....	46
Comparing State Between Function Calls.....	47
Bookmarks.....	48
Dealing With VR Applications.....	48
Tracing an Application That Is Already Running.....	49

Chapter 7: Exploring Your Application..... 51

Perspectives.....	52
Trace View.....	52
Bookmarks.....	53
Trace Outline View.....	54
Timeline View.....	55
Statistics View.....	55
Function Call View.....	56
Trace Analysis View.....	56
Target State View.....	56
Buffers View.....	57
OpenGL ES Framebuffers View.....	57
Vulkan Frame Capture View.....	58
Assets View.....	60
Asset Exporting.....	61
Shaders View.....	62
Textures View.....	62
Images View.....	63
Vertices View.....	63
Attributes Tab.....	63
Indices Tab.....	64
Geometry Tab.....	64
Uniforms View.....	64
Automated Trace View.....	65
Render Pass Dependencies View.....	67
Bookmarks View.....	68
Console View.....	69
Full Trace Replay.....	69
Scripting View.....	70
Filtering and Searching in MGD.....	71

Chapter 8: Integration with DS-5 Streamline.....	73
Installation.....	74
Using Streamline Annotations.....	74
Chapter 9: About the Implementation.....	77
Khronos Conformance.....	78
Chapter 10: Known Issues.....	79
OpenGL ES Extensions.....	80
Shading Language Version.....	80
Shader Compiler.....	80
Performance.....	80
API Asset Coverage.....	80
Memory.....	80
Partial Support for Earlier Trace Versions.....	80
GTK Warnings when closing MGD (Ubuntu only).....	80
Issues With Texture and Image Output on 64-bit Linux Hosts.....	81
Issues Viewing Khronos Reference Pages.....	81
Intercepting Without Using LD_PRELOAD.....	81
Multiple Drivers Installed on the System.....	82
Daydream VR compositor skips right eye execution on capturing a frame.....	82
Device Manager does not work correctly on Huawei Mate 9, Mate 10 and Mate 10 Pro.....	82
Chapter 11: Changes from Previous Versions.....	83
Changes between version 4.9.0 and 4.9.2.....	85
Changes between version 4.8.1 and 4.9.0.....	85
Changes between version 4.8.0 and 4.8.1.....	85
Changes between version 4.7.0 and 4.8.0.....	85
Changes between version 4.6.0 and 4.7.0.....	86
Changes between version 4.5.0 and 4.6.0.....	87
Changes between version 4.4.1 and 4.5.0.....	87
Changes between version 4.4.0 and 4.4.1.....	87
Changes between version 4.3.0 and 4.4.0.....	88
Changes between version 4.2.0 and 4.3.0.....	88
Changes between version 4.1.0 and 4.2.0.....	89
Changes between version 4.0.0 and 4.1.0.....	90
Changes between version 3.5.1 and 4.0.0.....	90
Changes between version 3.5.0 and 3.5.1.....	91
Changes between version 3.4.0 and 3.5.0.....	91
Changes between version 3.3.0 and 3.4.0.....	92
Changes between version 3.2.0 and 3.3.0.....	92
Changes between version 3.1.0 and 3.2.0.....	92
Changes between version 3.0.0 and 3.1.0.....	93
Changes between version 2.1.0 and 3.0.0.....	94
Changes between version 2.0.2 and 2.1.0.....	95
Changes between version 2.0.1 and 2.0.2.....	95
Changes between version 2.0.0 and 2.0.1.....	95
Changes between version 1.3.2 and 2.0.0.....	96
Changes between version 1.3.0 and 1.3.2.....	97
Changes between version 1.2.2 and 1.3.0.....	97
Changes between version 1.2.1 and 1.2.2.....	98
Changes between version 1.2.0 and 1.2.1.....	99
Changes between version 1.1.0 and 1.2.0.....	99
Changes between version 1.0.2 and 1.1.0.....	102
Changes between version 1.0.1 and 1.0.2.....	102

Changes between version 1.0.0 and 1.0.1..... 103

Changes in version 1.0.0..... 104

Chapter 12: Support..... 105

Chapter 13: Legal..... 107

Proprietary Notice..... 108

Analytics Information We Track..... 108

Chapter 1

Introduction

Topics:

- [Installation Package](#)
- [Components](#)
- [This Release](#)

The Mali Graphics Debugger is a tool to help OpenGL[®] ES, EGL[™], OpenCL[™], and Vulkan[™] developers get the best out of their applications through analysis at the API level. The tool allows the user to observe OpenGL ES, EGL, OpenCL, and Vulkan API call arguments and return values, and to interact with a running target application so as to investigate the effect of individual calls on the target. Attempted misuse of the API is highlighted, as are recommendations for improvement on a Mali-based system. Trace information may also be captured to a file on one system and be analyzed later. The state of the underlying GPU subsystem is observable at any point.

This document describes how to install and use the Mali Graphics Debugger on Windows, Linux and Mac OS X[®]. It describes how to examine an application running on a Linux or Android[™] target.

Installation Package

The installation package for the Mali Graphics Debugger contains everything you need to start investigating GPU applications on a desktop computer. Full support is available for:

- OpenGL ES 2.0, 3.0, 3.1, and 3.2
- EGL 1.4
- Vulkan 1.0

Limited support is available for:

- OpenCL 1.0, 1.1, and 1.2

You should have downloaded the installation package appropriate to your platform. The Mali Graphics Debugger is available for Windows, Linux and Mac OS X, and comes in both 32-bit and 64-bit variants. Typically, you should download the package that most closely resembles your host environment.

Components

The installation package contains three main components: the GUI application, the target intercept components and Mali Graphics Debugger sample traces.

This Release

This is a patch release following the 4.9.0 release of the tool, it adds the features and fixes the issues listed below:

- Upload traces to the target for Full Trace Replay.
- Fix unrooted interceptor issues with some devices.

This release also includes a small amount of bug fixes. See the [full change log](#) for details.

Chapter

2

Minimum Requirements

Topics:

- *Host System Requirements*
 - *Target System Requirements*
 - *Increasing Available Memory*
 - *Temporary Storage*
-

Host System Requirements

The host should be able to connect to the target via TCP/IP (for online analysis) and should have port 5002 open.

The installation requires around 350MB of disk space.

Up to 2GB of RAM will be claimed by the installed application as it runs. If available, more memory may be allocated to the application, which will allow larger traces to be accommodated.

Up-to-date operating system components and graphics drivers are recommended.

Target System Requirements

In general, the Mali Graphics Debugger should work on any Linux system that supports one or more of the following:

- OpenGL ES 2.0, 3.0, 3.1, or 3.2 **and** EGL 1.4
- OpenCL 1.0, 1.1, and 1.2

The system should be able to substitute the existing OpenGL ES, EGL, and OpenCL system libraries with the supplied interceptor libraries. The specific method for doing this is platform-specific.

In addition, this version of MGD has been tested with:

- Android™ 5.0 and 5.1 on a consumer tablet
- Android 6.0 on a consumer phone
- Android 7.0 and 7.1 on an internal development board

Previous versions of MGD have been tested with:

- Android 4.2, 4.3, and 4.4 on a consumer tablet

Other platforms should also work correctly although these have not been tested.

The professional edition of MGD is required for support of target systems using non-Mali GPUs.

Increasing Available Memory

Increasing the amount of memory made available to the application will increase the maximum trace size that may be accommodated. To do this, locate the *mgd.ini* file within the application installation and open it with a text editor. 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 will claim when running, with a trailing 'm' for megabytes. This number can be increased to match the capabilities of your system. Please note the format for the argument and ensure that your modifications follow the same format exactly (i.e. no spaces, trailing lower-case 'm'). The number you enter must be a multiple of 4.

Temporary Storage

Depending on the complexity of the application being traced, the Mali Graphics Debugger can require a large amount of temporary disk storage. By default, the system temporary storage directory is used. If the free space available in this directory is too little, MGD will display a warning and stop caching data to the directory, increasing memory usage.

The temporary storage directory can be changed by clicking on **Edit > Preferences** and selecting an existing directory to be used in the **Custom temporary storage directory** field.

Chapter

3

Host Installation

Topics:

- [Linux](#)
 - [Windows](#)
 - [Mac OS X](#)
 - [Target Deliverables](#)
 - [Licensing](#)
-

Linux

On Linux, the Mali Graphics Debugger is provided as a gzipped tar archive that can be extracted to any location on disk. This package can be extracted using any modern version (i.e. > 1.13) of GNU tar:

```
tar xvzf Mali_Graphics_Debugger_v<version>.<build>_Linux_<arch>.tgz
```

The GUI executable is 'mgd' within the gui directory

Windows

On Windows, the Mali Graphics Debugger is provided as an executable installer package. To install the software, run the installer and follow the instructions on screen. A link to the installed GUI will be added to the Start menu.

Mac OS X

On Mac OS X, the Mali Graphics Debugger is provided as a dmg package. It can be mounted by double-clicking on it. The GUI application is in the `gui` directory and can be launched directly from the dmg package or copied to the `Applications` on the local file system for easy access.

On Mac OS X version 10.9 (Mavericks) and later, the system may prevent you from running MGD as the package is not signed. The workaround is to hold the Ctrl key down when opening the app.

If you get the message "'Mali Graphics Debugger' is damaged and can't be opened", then you will need to launch the Terminal and temporarily disable the security assessment policy subsystem:

```
sudo spctl --master-disable
```

After you launch the app successfully, you can re-enable it with:

```
sudo spctl --master-enable
```

Target Deliverables

The extracted directory hierarchy on the host will contain a 'target' directory containing the following additional directories:

- linux/soft_float
Containing daemon and libraries for Linux-based Armv7/8 target devices.
- linux/hard_float
Containing daemon and libraries for Linux-based Armv7/8 Hard Float target devices.
- linux/arm64
Containing daemon and libraries for Linux-based Armv8 (64-bit) target devices.
- android/arm
Containing daemon and libraries for Android-based Armv8 (64-bit) and Armv7 (32-bit) target devices.
- android/intel
Containing daemon and libraries for Android-based x86-64 (64-bit) and x86 (32-bit) target devices.



Note: You will be required to manually add the libraries to your execution environment, see [Target Installation](#) on page 15 for details.

Licensing

Certain MGD features require a valid Arm DS-5 license. To set up a license, follow the documentation within DS-5. Using the license within MGD depends on how you obtained MGD:

- **As part of the DS-5 Suite**

You can launch MGD as usual and the license will be automatically picked up.

- **As a standalone product**

You must launch MGD from the DS-5 shell (Linux) or command prompt (Windows).

- On Linux run: `<ds5-install-directory>/bin/suite_exec <path-to-mgd-executable>`
- On Windows:
 1. Start the DS-5 Command Prompt as normal
 2. Navigate to the MGD installation folder within the prompt
 3. Run the *mgd.exe* executable

Chapter

4

Target Installation

Topics:

- [Linux](#)
- [Android](#)
- [Chrome OS](#)
- [Troubleshooting](#)

The Mali Graphics Debugger has two target components that interact to collect and transmit trace information from your application to the GUI on the host. These two target components are:

- The **Interceptor** library, which intercepts calls made by your application to one of the supported libraries and collects information about each call as it is made on the underlying system. The interceptor library needs to be loaded before your application starts. This library also doubles as a Vulkan layer, allowing Vulkan applications to be traced using the Vulkan API's layers system.
- The **Daemon** application, which collects the trace data from all running applications being intercepted and sends this data to the host. All data is transferred using TCP/IP on port 5002.

The method for using these components varies by target platform.

For Vulkan applications, the installation instructions in this section describe the simplest method of getting MGD running with your application using the layers system. However, the Vulkan layers system is quite flexible, and there may be multiple valid methods for installing the MGD layer on your target platform that may suit your needs better. For more details, see the [Vulkan Loader Specification and Architecture Overview](#).

We do not recommend that the Validation Layers are used at the same time as the MGD Layer. The Validation Layers are designed to validate a Vulkan application (i.e. detect invalid use of the Vulkan API), and MGD is designed to ensure that a Vulkan application that has already been validated does what it is expected to do.

Due to these different goals, using the MGD Layer at the same time as the Validation Layers may result in unexpected behavior.

Linux

Prerequisites

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

Installation

Navigate to the `[MGD installation directory]/target/linux` directory, and enter either the `soft_float` (soft float implementation), `hard_float` (hard float implementation) or `arm64` (Armv8 implementation) directory according to the configuration of your system.

Inside each of these directories, there should be the following files:

- `libinterceptor.so`
- `mgddaemon`



Important:

The Linux interceptor only supports Armv7/8 target architectures.

You must make sure you use the correct libraries for your target architecture. If you are running on Armv7, you must use either the soft float libraries or the hard float libraries, depending on the requirements of your system.

If you are running on Armv8 (64-bit) and intend to trace a 64-bit application, you must use the Armv8 libraries.

If you are running on Armv8 but are intending to trace a 32-bit application, you must use the appropriate Armv7 libraries, which will most likely be the hard float libraries.

The 64-bit build of the daemon can be used when tracing both 64-bit and 32-bit applications.

Installing the Daemon

Copy `mgddaemon` to anywhere on your target device, and set the execute permission bit on the file. This can be done by running the following command from inside the directory where `mgddaemon` was copied:

```
chmod +x mgddaemon
```

Installing the OpenGL ES, EGL, and OpenCL Interceptor

Copy `libinterceptor.so` to anywhere on your target device.

Installing the Vulkan Layer



Important: Tracing of Vulkan applications is supported on all Linux targets except for Arm soft-float.

The MGD interceptor also doubles as a Vulkan layer. In order to use it as such, you need to copy `libinterceptor.so` to anywhere on your target device, and rename it to `libVkLayerMGD.so`.

The `[MGD installation directory]/target/linux` directory contains the `VK_LAYER_ARM_MGD.json` manifest file. This file must be copied into the same directory as `libVkLayerMGD.so`.



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

Run Instructions

In order to connect MGD running on your host device to the target device you wish to trace on, the daemon application must be running on the target device.

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

```
./mgddaemon
```

The daemon can handle multiple applications starting and stopping, and should only be closed once you have finished tracing all the applications to be traced.

2. Connect to `mgddaemon` running on the device using the Device Manager (see [Linux Devices](#) on page 33). If the Device Manager detects a running instance of `mgddaemon` on the local network, it will give 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 [Connect to an IP](#) on page 34).
3. Start the application you want to trace by following the instructions in [Tracing an OpenGL ES, EGL, or OpenCL Application](#) on page 17, or in [Tracing a Vulkan Application](#) on page 17.

Tracing an OpenGL ES, EGL, or OpenCL Application

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

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

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

If you are unable to use `LD_PRELOAD` on your system there is an alternative; see [Intercepting Without Using LD_PRELOAD](#) on page 81 for more information.

If you have more than one version of your graphics driver on your system and are having issues, see [Multiple Drivers Installed on the System](#) on page 82 for more information.

Tracing a Vulkan Application

To trace your Vulkan application, you need to tell the Vulkan loader the location of the MGD layer and manifest that you copied on to your target, and the name of the MGD layer to load as both an instance and a device layer. For example:

```
VK_LAYER_PATH=/path/to/mgd/layer/ VK_INSTANCE_LAYERS=VK_LAYER_ARM_MGD  
VK_DEVICE_LAYERS=VK_LAYER_ARM_MGD ./your_vulkan_app
```



Note: The concept of "device layers" has been deprecated. However, certain older drivers may still require the `VK_DEVICE_LAYERS` environment variable to be set in order to allow MGD to trace all Vulkan function calls in the application.

Uninstalling

To uninstall, simply remove the files which were copied on to the target platform.

Android

Rooted Android

Prerequisites

- Android Software Development Kit (SDK) installed on the host machine.
- `PATH` should include the path to the `adb` binary.

- A valid ADB connection to the target device (i.e. `adb devices` returns the ID of your device with no permission errors and you can run `adb shell` without issues). See [the Android device documentation](#) for more help and information.
- The target device should be rooted and allow modification of the `/system` partition. If that is not possible, MGD provides an alternative, but it has certain limitations and application startup performance will suffer. To install MGD on an unrooted device, see [Prerequisites](#) on page 23.
- The target device should permit TCP/IP communication on port 5002.
- For Android targets you will need at least 11MB free on the `/system` partition, plus an additional 14 MB if you are also installing the 64-bit interceptor. Certain devices, such as the Nexus 10, may ship with very little free space on the system partition, and it may not be possible to install MGD without manually freeing up space in the system partition.

To check free space in the system partition, run the following command from a host command line:

```
adb shell df /system
```

Installation

Tracing applications using MGD on a rooted Android device can be performed without any modification of the target applications.

We recommend using the Device Manager (see [Android Devices](#) on page 32) to automatically install the MGD Android Application, the interceptor, and the Vulkan layer. However, instructions for installing each of these components manually is still provided.

Installing the Daemon



Attention: The `mgddaemon` application for Android has been deprecated in favor of the MGD Android Application. This can be automatically installed using the Device Manager (see [Android Devices](#) on page 32), or manually installed by following the instructions in [Installing the MGD Android Application](#) on page 24. Unlike for the MGD Android App, the Device Manager is not able to automatically launch and connect to the `mgddaemon` application.

1. Open a terminal on your host device, and navigate to the `[MGD installation directory]/target/android/` directory.
2. Copy the daemon on to your target device. If you are using an Arm based Mali Android device, run the following command in your terminal:

```
adb push arm/mgddaemon /sdcard/
```

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

```
adb push intel/mgddaemon /sdcard/
```

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

```
adb shell
su
```

4. To make sure your system partition is writable, run:

```
mount -o remount /system
```

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

```
cp /sdcard/mgddaemon /system/bin/mgddaemon
chmod 777 /system/bin/mgddaemon
```

Installing the OpenGL ES, EGL, and OpenCL Interceptor on 32-bit devices



Attention:

In 2017, Google rolled out a new driver interface as part of an initiative called [Project Treble](#). If you try to install the interceptor onto the `/system` partition on a device with Treble enabled,

you may render your device unable to boot. The device drivers will have been moved to the /vendor partition, but otherwise the structure is the same.

If possible, we recommend that you use the Device Manager to automate the installation (see [Android Devices](#) on page 32). Otherwise, you can follow the instructions below, but you **must** install the interceptor onto the /vendor partition instead of the /system partition.

You can check whether your Android device is Treble enabled using ADB with the command `adb shell getprop ro.treble.enabled`. If it returns "true", then your device is Treble enabled.

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/ directory.
2. Copy the 32-bit interceptor on to your target device. If you are using an Arm based Mali Android device, run the following command in your terminal:

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

Otherwise, if you are using an x86 based Mali Android device, run the following command in your terminal:

```
adb push intel/rooted/x86/libGLES_mgd.so /sdcard/
```

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

```
adb shell
su
```

4. To make sure your system partition is writable, run:

```
mount -o rw,remount /system
```

5. Install the interceptor by running:

```
cp /sdcard/libGLES_mgd.so /system/lib/egl/libGLES_mgd.so
```



Important: You may receive an error message similar to:

```
cp: /system/lib/egl/libGLES_mgd.so: No such file or directory
```

This means that there is not enough space in the /system partition. See the information in [Prerequisites](#) on page 17 for more information.

6. Grant appropriate permissions to the interceptor:

```
chmod 777 /system/lib/egl/libGLES_mgd.so
```

7. To make Android load the MGD interceptor library before the system library, the interceptor library must be named /system/lib/egl/libGLES.so.



Note: Due to a bug in the Android O driver loader, the interceptor library must be named /system/lib/egl/libGLES (with no file extension) on Android O devices.

Usually, there will not be a library with either of these names on the system, so it is sufficient to run:

```
ln -s /system/lib/egl/libGLES_mgd.so /system/lib/egl/libGLES.so
ln -s /system/lib/egl/libGLES_mgd.so /system/lib/egl/libGLES
```

8. If your system uses a non-monolithic driver (i.e. there are files called libGLESv2_*.so, libGLESv1_CM_*.so, and libEGL_*.so in /system/lib/egl/ or /vendor/lib/egl/), you will need to replicate that driver structure with MGD. To do this, run:

```
ln -s /system/lib/egl/libGLES_mgd.so /system/lib/egl/libEGL_mgd.so
ln -s /system/lib/egl/libGLES_mgd.so /system/lib/egl/
libGLESv1_CM_mgd.so
ln -s /system/lib/egl/libGLES_mgd.so /system/lib/egl/libGLESv2_mgd.so
```

Finally, reboot the device. Once rebooted, all OpenGL ES, EGL, and OpenCL function calls in any opened applications will be intercepted by `libGL ES_mgd.so`.

Installing the OpenGL ES, EGL, and OpenCL Interceptor on 64-bit devices

On 64-bit Android devices, you will need to install two copies of the interceptor library; one for tracing 32-bit applications, and one for tracing 64-bit applications. To install the 32-bit interceptor, follow the instructions specified in [Installing the OpenGL ES, EGL, and OpenCL Interceptor on 32-bit devices](#) on page 18.



Attention:

The same warning about Project Treble devices applies to the 64-bit interceptor. If you try to install the interceptor onto the `/system` partition on a device with Treble enabled, you may render your device unable to boot. The device drivers will have been moved to the `/vendor` partition, but otherwise the structure is the same.

If possible, we recommend that you use the Android Device Manager to automate the installation (see [Android Devices](#) on page 32). Otherwise, you can follow the instructions below, but you **must** install the interceptor onto the `/vendor` partition instead of the `/system` partition.

Next, you will need to install the 64-bit interceptor in the same manner as the 32-bit interceptor, except it must be copied into `/system/lib64/egl/` instead of `/system/lib/egl/`. This can be done by following these steps:

1. Open a terminal on your host device, and navigate to the `[MGD installation directory]/target/android/` directory.
2. Copy the 64-bit interceptor on to your target device. If you are using an Arm based Mali Android device, run:

```
adb push arm/rooted/arm64-v8a/libGL ES_mgd.so /sdcard/
```

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

```
adb push intel/rooted/x86-64/libGL ES_mgd.so /sdcard/
```

3. Finally, run the following commands to install the 64-bit interceptor:

```
adb shell
su
mount -o rw,remount /system
cp /sdcard/libGL ES_mgd.so /system/lib64/egl/libGL ES_mgd.so
chmod 777 /system/lib64/egl/libGL ES_mgd.so
ln -s /system/lib64/egl/libGL ES_mgd.so /system/lib64/egl/libGL ES.so
ln -s /system/lib64/egl/libGL ES_mgd.so /system/lib64/egl/libGL ES
```

You should end up with two copies of the interceptor:

- A 32-bit version in `/system/lib/egl`
- A 64-bit version in `/system/lib64/egl`



Note:

If you have a Huawei Mate 9, Mate 10 or Mate 10 Pro, please follow these [additional steps](#).

Finally, reboot the device. Once rebooted, all OpenGL ES, EGL, and OpenCL function calls in any opened applications will be intercepted by `libGL ES_mgd.so`.

Installing the Vulkan Layer on 32-bit Devices



Note: The instructions here will install the Vulkan layer onto your device, which will enable you to trace all Vulkan applications. Alternatively, you can package the Vulkan layers into just your Android application. (See [Using the Vulkan Layer in a Target Application](#) on page 25.)



Note: We do not recommend that the Vulkan Validation Layers are used at the same time as the MGD Vulkan Layer. For more information, see [Target Installation](#) on page 15.

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/ directory.
2. Copy the 32-bit interceptor on to your target device, and rename it to libVkLayerMGD.so. If you are using an Arm based Mali Android device, run:

```
adb push arm/rooted/armeabi-v7a/libGLLES_mgd.so /sdcard/libVkLayerMGD.so
```

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

```
adb push intel/rooted/x86/libGLLES_mgd.so /sdcard/libVkLayerMGD.so
```

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

```
adb shell
su
```

4. To make sure your system partition is writable, run:

```
mount -o remount /system
```

5. Starting in Android 7, Vulkan will try to load layers from the /system/fake-libs/ folder. You can install the layer on the device by running:

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

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

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

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

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

On older versions of Android, you can run:

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

Installing the Vulkan Layer on 64-bit Devices

On 64-bit Android devices, you will need to install two copies of the Vulkan layer; one for tracing 32-bit applications and one for tracing 64-bit application. To install the 32-bit Vulkan layer, follow the instructions specified in [Installing the Vulkan Layer on 32-bit Devices](#) on page 20.



Note: For Android, the names of the Vulkan layers don't actually matter. The Vulkan loader will enumerate all libraries it finds in /system/fake-libs/, /system/fake-libs64/ or /data/local/debug/vulkan/, regardless of the file name. Any libraries which can't be loaded (such as those with a different bitness to the Vulkan application) will be ignored.

Android 7 Onwards

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/ directory.
2. Copy the 64-bit interceptor on to your target device, and rename it to libVkLayerMGD.so. If you are using an Arm based Mali Android device, run:

```
adb push arm/rooted/arm64-v8a/libGLLES_mgd.so /sdcard/libVkLayerMGD.so
```

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

```
adb push intel/rooted/x86-64/libGLLES_mgd.so /sdcard/libVkLayerMGD.so
```

3. Finally, you will need to execute some commands to install the layer:

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

Older Versions of Android

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/ directory.
2. Copy the 64-bit interceptor on to your target device, and rename it to libVkLayerMGD64.so. If you are using an Arm based Mali Android device, run:

```
adb push arm/rooted/arm64-v8a/libGLLES_mgd.so /sdcard/libVkLayerMGD64.so
```

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

```
adb push intel/rooted/x86-64/libGLLES_mgd.so /sdcard/libVkLayerMGD64.so
```

3. Finally, you will need to execute some commands to install the layer:

```
adb shell
su
mount -o remount /system
mkdir -p /data/local/debug/vulkan/
cp /sdcard/libVkLayerMGD64.so /data/local/debug/vulkan/
chmod 777 /data/local/debug
chmod 777 /data/local/debug/vulkan
chmod 777 /data/local/debug/vulkan/libVkLayerMGD64.so
```

You should end up with two copies of the Vulkan layer:

- The 32-bit libVkLayerMGD.so
- The 64-bit libVkLayerMGD64.so

Choosing which Applications/Processes to Trace

By default MGD will trace all running OpenGL ES, EGL, OpenCL, and Vulkan applications. It can be preferable to limit MGD to only trace applications you're interested in.

To do this, the name of the application that should be traced should have its name written in /system/lib/egl/processlist.cfg. The processlist.cfg file may contain more than one process name, separated by a newline character.



Note: The processlist.cfg should have appropriate permissions. You can achieve this by running:

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


on the target device.

For convenience, the interceptor will print the name of any processes which link against OpenGL ES, EGL, OpenCL or Vulkan to logcat, so the name of the process can be found in this way. For example the line in logcat will read:

```
I/mgd_interceptor(11844): (11844): Interceptor: Interceptor loaded for
11844: com.arm.mali.Timbuktu
```

In which case your processlist.cfg file should contain a single line:

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

 **Note:** If /system/lib/egl/processlist.cfg exists but is **empty** then no applications will be traced.

Uninstalling

To uninstall MGD, we need to reverse the installation process.

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

```
adb shell
su
```

2. To make sure your system partition is writable, run:

```
mount -o remount /system
```

3. Remove the daemon by running:

```
rm /system/bin/mgddaemon
```

4. If you installed the 32-bit OpenGL ES, EGL, and OpenCL interceptor:

- a. Remove the interceptor by running:

```
rm /system/lib/egl/libGLES_mgd.so
```

- b. If you created the /system/lib/egl/libGLES.so symlink, remove it by running:

```
rm /system/lib/egl/libGLES.so
```

- c. If you created symlinks for a non-monolithic system, remove them by running:

```
rm /system/lib/egl/libEGL_mgd.so
rm /system/lib/egl/libGLESv1_CM_mgd.so
rm /system/lib/egl/libGLESv2_mgd.so
```

5. If you installed the 64-bit OpenGL ES, EGL, and OpenCL interceptor:

- a. Remove the interceptor by running:

```
rm /system/lib64/egl/libGLES_mgd.so
```

- b. If you created the /system/lib64/egl/libGLES.so symlink, remove it by running:

```
rm /system/lib64/egl/libGLES.so
```

6. If you installed the 32-bit Vulkan layer, remove it by running:

```
rm /data/local/debug/vulkan/libVkLayerMGD.so
```

7. If you installed the 64-bit Vulkan layer, remove it by running:

```
rm /data/local/debug/vulkan/libVkLayerMGD64.so
```

8. Finally, reboot the target device to allow the original libraries to be reloaded.

Unrooted Android

Prerequisites

- Android Software Development Kit (SDK) installed on the host machine.
- PATH should include the path to the adb binary.

- A valid ADB connection to the target device (i.e. `adb devices` returns the ID of your device with no permission errors and you can run `adb shell` without issues). See [the Android device documentation](#) for more help and information.
- The target device should permit TCP/IP communication on port 5002.
- The source code for the application to be traced should be accessible.
- The Android device must be running Android 5.0 or above.

Limitations

There are several drawbacks to installing MGD inside the target application rather than in the /system partition of the device. These are outlined below:

- The method requires source code access
- Tracing extensions may work but is unsupported
- Application start times may increase compared to the standard installation approach

Due to these limitations, Arm recommends that the interceptor library is installed in the /system partition whenever possible.

Installation

Installation on an unrooted device involves modifying the source code and build script of the application that needs to be traced, installing the MGD Android Application on the target device, and running the MGD Android Application prior to launching the application to be traced.

We recommend using the Device Manager (see [Android Devices](#) on page 32) to automatically install the MGD Android Application. However, instructions for installing the application manually are provided below.

Installing the MGD Android Application

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/arm/ directory.
2. Install the MGD Android Application by running:

```
adb install -r MGD.apk
```



Note: The MGD Android Application fulfils the same role as the console `mgddaemon` application used when tracing on a rooted Android device.

Using the OpenGL ES, EGL, and OpenCL Interceptor in a Target Application

Using the interceptor in an Android application requires two steps:

1. Packaging the interceptor into the application
2. Enabling the interceptor

Step 1 is the same for all applications, but step 2 depends on whether your application uses Java, C++, or both.

Packaging the Interceptor

We need to add the MGD unrooted library folder to the list of locations the Android build system looks for native libraries. This will ensure MGD is included in the resulting APK.

To do this, add the following code to your modules `build.gradle` file:

```
android {
    sourceSets {
        main {
            jniLibs.srcDirs += ['[MGD installation directory]/target/
android/arm/unrooted/']
        }
    }
}
```


Replacing [MGD installation directory] with the real path.

Enabling the Interceptor with a C/C++ Only Application

Native applications on Android are unable to load a library that depends on non-system libraries. Therefore, linking your native library against MGD will cause the application to fail at runtime as it won't be able to load MGD even if it's included in the APK.

The solution to this is to add a very small Java component to your application. Create a new Activity class which extends `android.app.NativeActivity`.

Make sure this new Activity is referenced in your `AndroidManifest.xml` as the `android:name` attribute of the `activity` element. Also check that the `android:hasCode` attribute of the `application` element is set to `true`, otherwise the Java files will not be included in the APK.

Now you can follow the instructions below for mixed Java and C/C++ applications.



Note: There is no need to link your native library to MGD.

Enabling the Interceptor with a Java Only or a Mixed Java and C/C++ Application

Now we need to make sure your application will actually load MGD. Add the following code to the beginning of your project's main Activity class:

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

Recompile the application and install it on your Android device.

ABIs

The interceptor for unrooted devices is only available for the `armeabi-v7a` and `arm64-v8a` ABIs.

To ensure your build works as expected you must make sure you specify a appropriate value for `abiFilters` in your modules `build.gradle`. To target both of the Armv7 and Armv8 targets, your modules `build.gradle` should contain:

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

See the [ABI Management](#) documentation for more information about Android ABIs.

Using the Vulkan Layer in a Target Application

The same Vulkan layer used for rooted Android devices can also be used for unrooted Android devices. However, as the system directory where debug layers need to be placed is inaccessible to unrooted devices, the MGD Vulkan layer must be packaged with the target application.

1. Open a terminal on your host device, and navigate to the [MGD installation directory]/target/android/arm/rooted/ directory.
2. Create the 32-bit and 64-bit Vulkan layers by running:

```
cp armeabi-v7a/libGLLES_mgd.so libVkLayerMGD.so
cp arm64-v8a/libGLLES_mgd.so libVkLayerMGD64.so
```

3. Package the created MGD Vulkan layers into your Android application in the same manner as the Vulkan Validation Layers. The details of how to do this can be found in the [Vulkan Validation Layers on Android](#) documentation.

During initialization of the target Vulkan application, the VK_LAYER_ARM_MGD layer should be enabled as both an instance and as a device layer.



Note: We do not recommend that the Validation Layers are used at the same time as the MGD Layer. For more information, see [Target Installation](#) on page 15.

Installing for an OpenGL ES Unity® Application

You can use the Mali Graphics Debugger with Unity® on an unrooted Android device. Below is an outline of how to do this using recent versions of MGD. Alternatively, a more detailed set of instructions are provided in [the Arm Guide for Unity Developers](#).

1. Launch Unity on your desktop.
2. Open Unity Build Settings and check the Development Build option.
3. You will need to copy [MGD installation directory]/target/android/arm/unrooted/armeabi-v7a/libMGD.so into a subfolder in your Unity project.
 - To package the OpenGL ES interceptor, copy into a folder called Assets/Plugins/Android/.
 - To package the Vulkan interceptor, copy into a folder called Assets/Plugins/Android/libs/armeabi-v7a/ and rename libMGD.so to libVkLayerMGD32.so.

You may need to create these subfolders yourself if they do not already exist. You can package one or both of the interceptor libraries depending on your application's requirements.

4. Build the APK and install it using `adb install -r YourApplication.apk`.
5. Follow the instructions in [Tracing an Android Application](#) on page 27 to trace the Unity application.

Installing for an OpenGL ES Unreal® Engine Application

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

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

A summary of those instructions are provided here:

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

Once the Unreal® Engine application is installed, follow the instructions found in [Tracing an Android Application](#) on page 27 to connect to the device and trace the application.

Uninstalling

Uninstalling the MGD Android Application follows the same principle as any other Android application. Simply open the menu, press and hold on the application's icon, then drag it to the uninstall (trash bin) icon on-screen.

To exclude libMGD.so from the target application's build, simply comment out or delete the code you added in the installation instructions (see [Installation](#) on page 24).

Tracing an Android Application

Once your device has been correctly set up with MGD, you will need to connect to the device in order to trace your application.

If your device is both visible over ADB and the MGD Android Application has been installed, you can use the Device Manager to connect to your device by following the instructions in [Connecting to Your Device](#) on page 33.

Alternatively, you can manually connect to an Android device by IP:

1. If you want to connect to a device using ADB, open a terminal on your host device and run:

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

In this case, the target IP will be 127.0.0.1:5002.

2. If you are using the MGD Android Application, start it, and switch the **Enable MGD Daemon** option in the **App List** tab to the *ON* state.

If you are using the mgddaemon application on a rooted device, you should:

- a. Open a terminal on your host device.
- b. Open a shell as a super user on your target device by running:

```
adb shell
su
```

- c. If you are going to be tracing a Vulkan application, run:

```
setprop debug.vulkan.layers VK_LAYER_ARM_MGD:
```

This command tells the Vulkan loader to load the layer with the name VK_LAYER_ARM_MGD, which should have been installed in /data/local/debug/vulkan.

- d. Start the daemon by running:

```
mgddaemon
```

3. Follow the instructions in [Connect to an IP](#) on page 34 to connect to the target IP.

Once a connection has been established, you can start the Android application that you want to trace. If you have correctly packaged the interceptor or Vulkan layer with your application, or if your device is rooted and the interceptor or Vulkan layer have been installed on your device, the application will show up in the MGD Android Application's list of all traceable applications.



Note:

If an application is started before the daemon is started, only the function calls made from the time when the daemon was started will be traced. In this case, the state and assets of any traced applications will be incomplete when viewed in MGD.

If an OpenCL or Vulkan application is started before tracing begins, the state and assets of any traced applications will be incomplete when viewed in MGD. OpenGL ES applications started before tracing begins can have their state recovered. For more information, see [Tracing an Application That Is Already Running](#) on page 49.


Chrome OS

Prerequisites

MGD can be used to debug graphics APIs running on Chrome OS. There are three types of tracing that MGD supports for Chrome OS:

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


To debug your Chrome OS device, you will need to enter [Developer Mode](#). From there, you can follow the instructions below depending on your application.

 **Note:** You will want to enable debugging features when you boot into Developer Mode if you want to debug Chrome or Linux applications.

Tracing an Android Application on Chrome OS

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

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

 **Note:** There is no equivalent to the the rooted Android installation on Chrome OS. This means you will need to package the interceptor into your Android application before you can trace it. The instructions to do this can be found in [Using the OpenGL ES, EGL, and OpenCL Interceptor in a Target Application](#) on page 24 and in [Using the Vulkan Layer in a Target Application](#) on page 25.

1. Connect to your device over a network with ADB. The IP address of the target device can be found in the Chrome OS WiFi menu, and the default port to use is 22:

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

For more information about connecting to a ChromeOS device over ADB, see the [Optimizing Apps for Chromebooks](#) guide.

2. Follow the instructions in [Tracing an Android Application](#) on page 27 to connect to the device and trace the application.

Tracing a Linux Application on Chrome OS

The Linux interceptor and MGD Daemon can be used to trace Linux applications running on Chrome OS devices.

 **Note:** You will need to [set up SSH access to your Chrome OS device](#) before you can trace native Linux applications with MGD.

1. SSH into the Chrome OS device as root with `ssh root@(IP address)`
2. Create a directory to store the MGD Daemon and interceptor library, e.g. `mkdir /usr/bin/mgd`
3. Run the following command to allow connections on port 5002:

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

4. You will need to copy the MGD Daemon and interceptor library onto your device. Depending on your device, you will need to use a version of the Linux MGD components appropriate to your architecture (either hard float, soft float, or 64-bit). You may need root access to copy files onto the Chrome OS file system. For example:

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

5. From your root user SSH session, launch `mgddaemon`.
6. Using the Device Manager, connect to the running daemon by following the instructions in [Linux Devices](#) on page 33 or in [Connect to an IP](#) on page 34.
7. Launch a new SSH session.
8. Run your Linux application while preloading the interceptor library. Instructions on doing this can be found in [Tracing an OpenGL ES, EGL, or OpenCL Application](#) on page 17.

You should start to get trace data appearing in the desktop MGD client.

Tracing Chrome on Chrome OS

MGD supports tracing the Chrome application in Chrome OS. This is useful for debugging websites, web apps, and Chrome applications. The methodology for this is similar to [Tracing a Linux Application on Chrome OS](#) on page 28, with a few differences.



Note:

You will need to [set up SSH access to your Chrome OS device](#) before you can trace Chrome with MGD.



Note:

Chrome OS will try to reboot when you attempt to stop the UI. To prevent this, you will need to 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
```

Once you have SSH access and the ability to stop the UI, you will need to install MGD onto your device.

1. Follow the MGD Daemon and interceptor installation instructions in [Tracing a Linux Application on Chrome OS](#) on page 28.
2. You will need to 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 `mgddaemon`. You may need to restart the root session after starting an SSH session as chronos.
5. Connect to your Chrome OS device from MGD.
6. From your chronos SSH session, suspend the Chrome OS UI using the command `sudo stop ui`.
7. You will need to preload the interceptor library and launch Chrome from the chronos user SSH session. For example:

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

More information about preloading the interceptor library can be found in [Tracing an OpenGL ES, EGL, or OpenCL Application](#) on page 17.

You should start to get trace data appearing in the desktop MGD client.



Note: On some devices, Chrome may not launch correctly while the desktop MGD client is connected, and will launch numerous sub-processes. If this happens, you can disconnect the MGD client and Chrome should launch correctly. You should be able to connect MGD and trace Chrome after it has launched.

Troubleshooting

Target Devices With No cp Support

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

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

No Trace Visible

The interceptor component on the target reports through logcat on Android. If no trace is found then it is advisable to review the logcat trace. Generally you should:

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

Frame Capture Doesn't Work on Android 4.4 and Above

If you find that the frame capture feature of MGD is not working on an Android 4.4 (or above) device, it may be because the application is manually loading the graphics libraries based on the information in `/system/lib/egl/egl.cfg`. To check if this is the case and to resolve the problem, try to move `/system/lib/egl/egl.cfg` by running the following commands:

```
adb shell
# The following commands run on the target
su
mv /system/lib/egl/egl.cfg /system/lib/egl/egl.cfg.bak
```

Socket Permission Errors

The MGD interceptor uses an abstract namespace Unix domain socket (AF_UNIX) to connect to the MGD daemon. This doesn't require any special permissions for your application. However, certain platforms with additional security measures may limit access to AF_UNIX sockets.

If you've followed the instructions in the installation guide but are getting no trace output, have a look at the log output on your device (e.g. using `adb logcat` on Android). If you see something like:

```
06-26 20:57:18.430: I/mgd_interceptor(9622): Trying to connect to the
daemon...
06-26 20:57:18.430: E/mgd_interceptor(9622): error socket connect:
Permission denied
```

There is a permission issue on abstract namespace AF_UNIX sockets, which is stopping the interceptor talking to the daemon. In this case it's worth talking to your platform vendor to determine how to enable access to AF_UNIX sockets.

If your device uses SELinux then running `"setenforce 0"` on the device can enable access to AF_UNIX sockets.

Chapter

5

Device Manager

Topics:

- [Android Devices](#)
- [Linux Devices](#)
- [Connect to an IP](#)

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

Android devices connected to the host using ADB are automatically detected, and the Device Manager can automatically install the MGD Android app, and the interceptor and Vulkan layer for rooted devices.

Linux devices that have the daemon running, and are on the same local network and the same subnet as the host are automatically detected by the Device Manager.

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

The Device Manager can be launched by clicking the  button.




Note:


The Device Manager does not currently work with Huawei Mate 9, Mate 10 or Mate 10 Pro. Please perform a [manual installation](#).

Android Devices

Android devices connected to the host using ADB are displayed in the Device Manager. You can install MGD components to your Android device automatically using the Device Manager, and can then connect to an Android device which has had the MGD Android App installed.

 **Note:** The Device Manager is the preferred method of installing MGD onto your Android device, although instructions are still provided for installing manually. (See [Target Installation](#) on page 15.)

On the first launch of the Device Manager, you will be prompted to provide the location of the Android Debug Bridge (ADB) binary. The Device Manager can try to auto-detect your ADB binary if it is on the system PATH, or you can manually provide a location.

 **Note:** ADB is available for free as part of the Android SDK. For more information, you can consult the [Android Debug Bridge user guide](#).

You can change the ADB path at any time in **Edit > Preferences > MGD**.









If you choose to use the Device Manager to automatically install the MGD components that require root access to your device, you should be aware that there are risks involved. For example, if the connection between the host and the target is disconnected, then MGD may only partially install its components. This can make it impossible to boot into Android.

The Device Manager is conservative about writing to your system partition. It will run checks to make sure that common failure conditions (like insufficient storage) do not exist. It will try to clean up if it fails unexpectedly. It will warn you before it attempts anything that might be dangerous.

Although we recommend the Device Manager over manual installation, Arm cannot guarantee that your device will not get corrupted. If you have the source code of your application and do not want to install the rooted components, you can install the unrooted MGD Android Application using the Device Manager, then package the interceptor as described in [Unrooted Android](#) on page 23.



Installing MGD Components

After the Device Manager has found the ADB binary, it will search for devices that are visible to ADB. This includes devices that you connected over a network using `adb connect` as well as those plugged in to your host machine.

ID	Name	MGD Android App	Root Interceptor	Root Vulkan Layer
 ce091609346f430602	 samsung SM-G930F	 Installed 	 Not installed 	 Not installed 

▼ ADB Tasks

Device	Description	Status
No ADB tasks in progress		



 Copy Command Logs
  Clear Completed Tasks

The Device Manager will present a list of all connected devices and will attempt to detect each of the three MGD target-side components: the **MGD Android App**, the **Root Interceptor**, and the **Root Vulkan Layer**.

- The **MGD Android Application** is responsible for managing the MGD Daemon service on your device. The Device Manager needs the Android Application to be installed before it can connect to the device. You can also open the application on your device for a list of applications you can trace, or to manually enable/disable the MGD Daemon. Installing this component does not require root access.
- The **Root Interceptor** is responsible for tracing your application. The Device Manager will install it into your `/system/` partition, which will allow you to trace any application that uses the OpenGL ES, EGL, or OpenCL APIs. Installing this component requires root access.
- The **Root Vulkan Layer** is responsible for tracing Vulkan applications. The Device Manager will install it into your `/system/` partition (for Android 7) or your `/data/` partition (for earlier versions of Android), which will allow you to trace any application that uses the Vulkan API. Installing this component requires root access.



Note: We do not recommend that the Vulkan Validation Layers are used at the same time as the MGD Vulkan Layer. For more information, see [Target Installation](#) on page 15.

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

Uninstalling MGD Components


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

Connecting to Your Device

After you have installed the **MGD Android Application**, you can connect to your device from the Device Manager. The Device Manager will handle setting up your device for tracing, including port forwarding, starting the MGD daemon service, and enabling the MGD Vulkan layer.



Note: In order to trace your application, you will need to have installed either the interceptor or the Vulkan layer. Alternatively, you can package them into your application; this process is described in [Unrooted Android](#) on page 23.

Click on the  connect button, or double click on a disconnected device to launch the MGD Daemon on your device and connect it to the host client. Once a connection has been established, the Device Manager will automatically close and the new live trace will be shown.

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

Using the ADB Tasks Log

The ADB Tasks table contains a list of tasks that the Device Manager has attempted. You can see their status (success, failure, running, or pending) and the description of the task that was issued. Each ADB task roughly corresponds to the instructions from [Target Installation](#) on page 15.


The Device Manager maintains a record of every ADB command that it has issued. Double clicking on the task will show the command log for a specific task.


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


The Device Manager will try to provide a useful error message when an ADB task fails. But you can use the command logs to diagnose more complex issues with installing onto your device.

Linux Devices


If a Linux device is on the same local network, is on the same subnet as the host, and has the `mgddaemon` application running on it, it can be automatically be detected by the Device Manager.

	Identifier	Protocol	GL Vendor	GL Renderer	GL Version
	10.2.200.200	D015	ARM	OpenGL ES 3.2	Mali-T760

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


Click on the  connect button, or double click on a disconnected device to connect to the device. Once a connection has been established, the Device Manager will automatically close and the new live trace will be shown.


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

 **Note:** The device's GL Vendor, GL Renderer, and GL Version strings are obtained by dynamically loading the `libGLESv2` and `libEGL` libraries, and may show up as "Unknown" if the libraries could not be found or loaded.

Connect to an IP

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

IP Address	<input type="text" value="10.2.200.200"/>	Port	<input type="text" value="5002"/>	 Connect
------------	---	------	-----------------------------------	---

Enter the IP address of the target device and the port that the daemon is running on (default 5002) and click on the  connect button to connect to the device. Once a connection has been established, the Device Manager will automatically close and the new live trace will be shown.

Chapter

6

Getting Started with MGD

Topics:

- [*Running the Host GUI*](#)
- [*Using the Mali Graphics Debugger on Your Application*](#)

Running the Host GUI

Run the application:


- On Windows by double-clicking the MGD icon.
- On Linux by running

```
/path/to/installation/gui/mgd &
```

- On Mac OS X by double-clicking the MGD icon inside the gui directory.

The main window should open.

Load one of the supplied sample traces from `/path/to/installation/samples/traces/`

using **File > Open**, or using the toolbar  button. The various application windows should fill with information from the loaded trace that you may examine at will.


Using the Mali Graphics Debugger on Your Application


Tracing an Application

The instructions on how to install MGD on a target device, connect to a target device, and to start an application to be traced on a target device can be found in [Target Installation](#) on page 15.

After a connection to a target device has been successfully established, a new live trace will be shown in the GUI. At this point, the Process Configuration (see [Process Configuration](#) on page 36) can be changed to control which assets are sent from the target to the host during the trace.

After the desired configuration has been set, the application to be traced can then be started. If MGD has been correctly installed on the target, the live trace will begin to receive trace data.

To disconnect from the target, use the disconnect  button.

The connect  button will attempt to connect to the last connected to device, or open the Device Manager if there was no prior connection.

Process Configuration

The Process Configuration is used to control which contents of which types of API assets are sent from the target device back to the host when API function calls are traced.

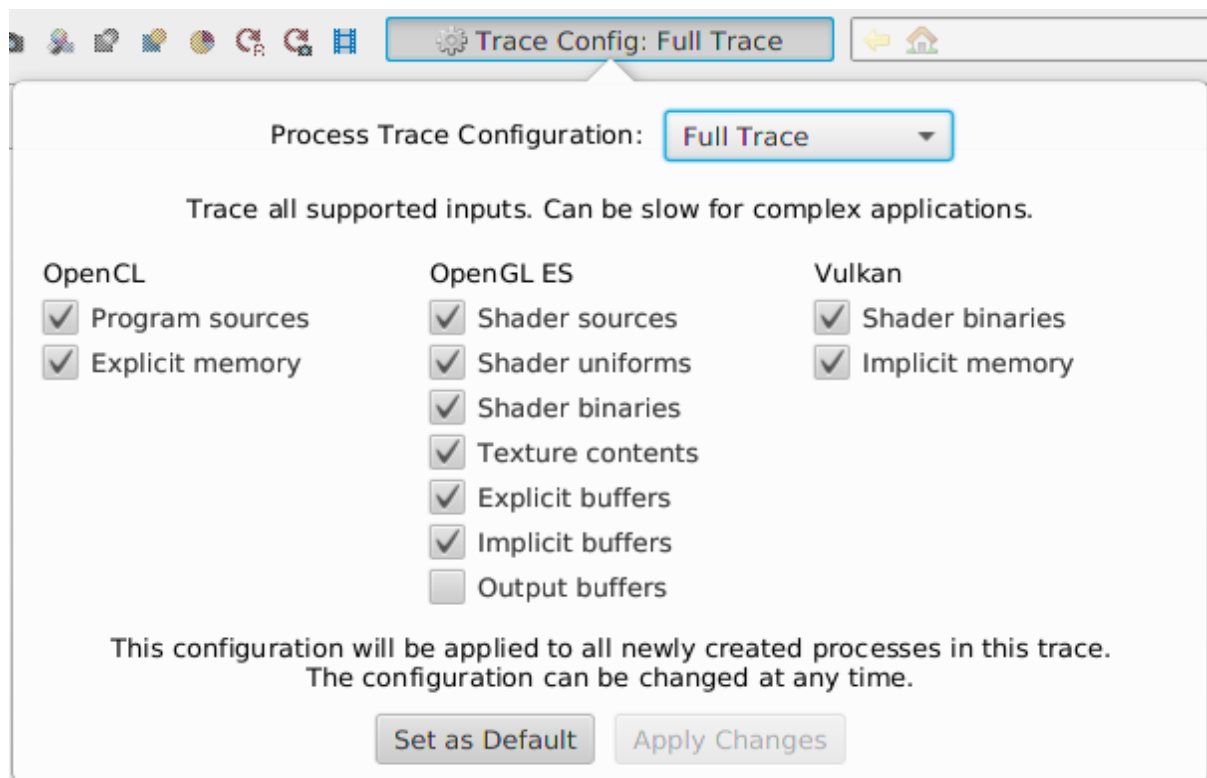
For example, if you don't care about the contents of the textures and the buffers being used in your OpenGL ES application, you can use the Process Configuration to disable these asset types, and speed up tracing of your application. In this case, MGD will still track the existence and the life cycle of these assets, but their contents can not be examined.

The more types of assets that are sent, the more information is available to and visible in MGD. However, the application being traced will run more slowly, the trace will be larger, and MGD will use more memory.



Note: A distinction between explicit and implicit memory is made in the Process Configuration. Explicit memory is memory uploaded explicitly by a function call, e.g. 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, e.g. in a call to `glMapBuffer`, `vkMapMemory`, or any other function call that returns a pointer to memory that can then be modified by the application.


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




The configuration can be changed at any time, even after an application has been started on the target device. Any function calls traced in the application process after the new configuration has been set will use the newly set configuration.

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


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

 **Note:** By default, the **Legacy** configuration is used. The types of assets sent in this configuration are equivalent to the types sent by default in versions of MGD prior to the introduction of the Process Configuration.


 **Note:** If you are creating a trace that you wish to replay using the Full Trace Replay feature, the **Full Trace** configuration (or greater) must be used. See [Full Trace Replay](#) on page 69 for more information.


Dealing With Multiple Processes


It is possible to trace multiple processes concurrently with MGD. Any intercepted application launched after the host has connected to the daemon will be received into the trace file on the host. Individual processes will be displayed in the [Trace Outline View](#).


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


Pausing, Stepping Frames, and Resume

The currently selected process can be paused by pressing the  button; the process will be halted at the next `eglSwapBuffers()` call to allow you to examine the result. Pressing this button again before the process has paused will force the application to pause on the next function call (regardless if it is `eglSwapBuffers()` or not).


All connected processes can be paused by pressing the  button; the processes will be halted at the next `eglSwapBuffers()` call to allow you to examine the result.



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


The selected process may be resumed by pressing the  button.

 **Note:** Only the threads that are calling the graphics API functions will be paused.

Capturing Framebuffer Content


Whilst executing, it is possible to perform an in-depth capture of the output from the graphics system on a draw-by-draw basis. Press the  button to take a snapshot of framebuffer content following each draw call in the next full frame of the application. Note that this involves considerable work in composing and transferring data that will slow the application down considerably.

In order to capture subsequent frames you will need to pause the application with the  button. Once paused, proceed by pressing the  button for each frame that needs capturing.

In the Trace Outline View any regular frame will now have a  icon to show that it is a captured frame.

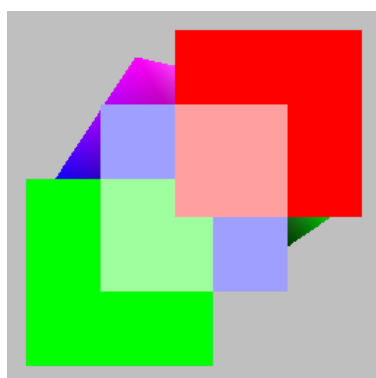
There are some additional limitations on frame capture for Vulkan applications. For more information, see the [Vulkan Frame Capture View](#) section.

Capturing All Framebuffer Attachments

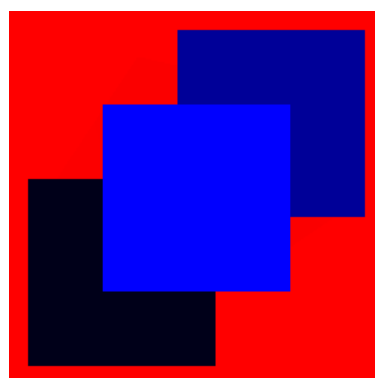
It is possible to capture most framebuffer attachments including all color attachments, and the depth and stencil attachments. Whilst capturing a live trace, press the  button. When a frame is captured with this mode enabled, all the available attachments will be captured for each draw call. This information is visible for each framebuffer in the Framebuffers View.

Example

In this example 3 squares are drawn on screen with varying depths moving from -1.0 towards 0.0, with a colored cube rendered behind them. All 4 draw calls (the 3 squares + 1 cube) have different values set for the stencil buffer write mask, with the stencil pass operation set to `GL_REPLACE`.



Color Attachment



Depth Attachment



Stencil Attachment

Depth attachment values range from -1.0 ... 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.


Capturing All Framebuffer Attachments Limitations

There are some limitations to which attachments can be captured by this mode:

- This mode is disabled in the interceptor for Mali-2/3/4xx devices
- It is not possible to capture the depth or stencil attachments for FBO 0 on a OpenGL ES 2.0 only configuration
- For any configuration where depth texture sampling is not supported, or where the device only has OpenGL ES 2.0 available and the depth attachment is a renderbuffer, only a low resolution capture of the depth buffer is possible.

It is also worth bearing in mind that capturing all attachments increases the per-draw-call capture time and the amount of data transmitted by the device to the host.

Analyzing Overdraw

MGD has the ability to show overdraw in a given scene. Whilst capturing a live trace press the  button. MGD will now replace the fragment shader in the target application with an almost transparent white colored fragment shader. Each time a pixel is rendered to the framebuffer the alpha value is increased via an additive blend, meaning the whiter the final image appears the more overdraw happens in that area.



Original Image

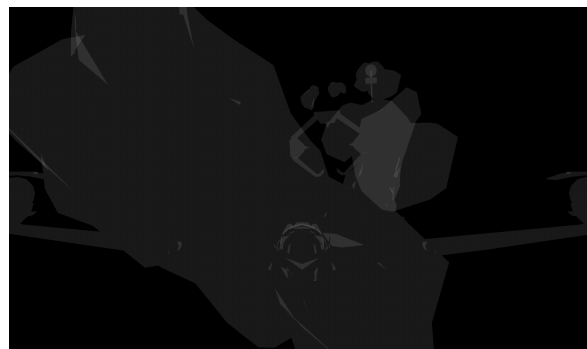




Image with overdraw feature turned on

An application with low levels of overdraw should appear to be a uniform dull gray.

Any frame with overdraw mode turned on will have this  icon in the Trace Outline View.

Analyzing the Shadermap

MGD has the ability to give each shader program in use in a given scene a different solid color. Whilst capturing a live trace, press the  button. This allows the shader in use by each object in the scene to be identified, and allows detection of any bugs that may be caused by incorrect shader assignment. An example of this feature is displayed below:



Original Image

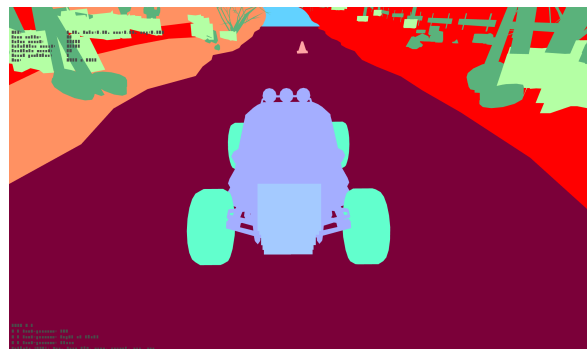



Image with shader map feature turned on

There are 100 unique colors that can be assigned to shader programs, at which point programs will have duplicate colors. The program that corresponds to a color can be identified by matching the color in the scene with the color shown in the Shaders View, as shown below. Alternatively you may position the cursor over a captured framebuffer image (captured in shadermap mode) and the active shader will be identified above the image.

Target State	Buffers	Uniforms	Vertex Attributes	Vertex Shaders							
Program	Name	A	L/S	T	Total	Uniform Registers	Work Registers	Vertices	Total cycles	% cycles	
132	Shader 130	45	18	0	63	13	8	11,121	467,082	19.2%	
384	Shader 382	21	18	0	39	16	5	14,196	397,488	16.4%	
342	Shader 340	21	18	0	39	16	5	13,323	373,044	15.3%	
258	Shader 256	31	25	0	56	8	13	8,445	346,245	14.2%	
456	Shader 454	39	22	1	62	8	10	7,878	338,754	13.9%	
174	Shader 172	14	8	0	22	6	3	10,629	159,435	6.6%	
66	Shader 64	39	15	0	54	13	8	3,912	140,832	5.8%	
300	Shader 298	31	25	0	56	8	13	2,373	97,293	4.0%	
24	Shader 22	14	8	0	22	6	3	3,912	58,680	2.4%	
426	Shader 424	29	19	0	48	16	5	1,200	39,600	1.6%	
531	Shader 529	23	13	0	36	12	6	240	6,000	0.2%	
543	Shader 541	23	13	0	36	12	6	180	4,500	0.2%	
Total Cycles: 2430931 (cumulative over frame so far)											

Assets	Fragment Shaders	Compute Shaders	Textures	Frame Overrides		
<div> <div>Assets</div> <div> <div>GLSL</div> <div> <div>Buffers</div> <div> <div>Framebuffers</div> <div> <div>Framebuffer 0</div> <div>Framebuffer 1</div> <div>Framebuffer 2</div> <div>Framebuffer 3</div> <div>Framebuffer 4</div> <div>Framebuffer 5</div> <div>Framebuffer 6</div> <div>Framebuffer 7</div> <div>Framebuffer 8</div> <div>Framebuffer 9</div> <div>Framebuffer 10</div> </div> </div> </div> </div> </div>						<div>Framebuffer 3</div> <div> <input type="checkbox"/> Alpha </div> 


Any frame with shadermap mode turned on will have this  icon in the Trace Outline View.

Overdraw and Shadermap Limitations

Applying any full screen post processing effects, for example rendering to a texture, will prevent the overdraw map or shadermap from displaying correctly on the device. However, the information can be seen by switching to the correct framebuffer in the UI after capturing a frame while either feature is active.

Analyzing the Fragment Count

MGD has the ability to count the number of fragments processed by a shader per draw call. If depth testing is enabled and a fragment would be excluded as a result, then that fragment will not be included in the count.


To toggle this feature, click the  button. When enabled, each draw call will increment 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. An example of this feature is shown below.

Assets Vertex Shaders Fragment Shaders Compute Shaders Textures Frame Overrides											
Program	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%	
Total Cycles: 125060512 (cumulative over frame so far)											

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



Note: The two columns *Fragments* and *Total cycles* will only be available for those frames where the fragment count analysis has been requested. These columns will indicate ‘N/A’ (not available) for other frames.



Any frame with fragment count mode turned on will have this  icon in the Trace Outline View.


Fragment Count Limitations


To maintain compatibility with older OpenGL ES 2.0 hardware and software, this feature uses a software method to count the number of fragments. As a result, a single draw call can take several seconds to complete. In addition, the target device screen will show only the final draw call in a frame, and the frame capture feature will not show any usable information.

Replaying a Frame

MGD has the ability to replay certain frames on the target device, depending on what calls were in that frame. To see if a frame can be replayed you must pause your application in MGD. Once paused, if

a frame can be replayed, the  (frame replay) and  (frame replay with capture) buttons will be enabled.

Clicking  will cause MGD to reset the OpenGL ES state of your application back to how it was before the previous frame had been drawn. It will then play back all the function calls in that frame on

the target device. The  button operates in the same way except it will also enable *frame capture* at the same time.

This feature can be combined with the *Capture All Attachments*, *Overdraw*, *Shademap*, and *Fragment Count* features. You can, for example, pause your application in an interesting position, activate the Overdraw mode and then replay the frame. The previous frame will be replayed exactly as before but with the overdraw mode enabled. You can repeat this process with the other modes enabled to get a complete picture for a single frame.

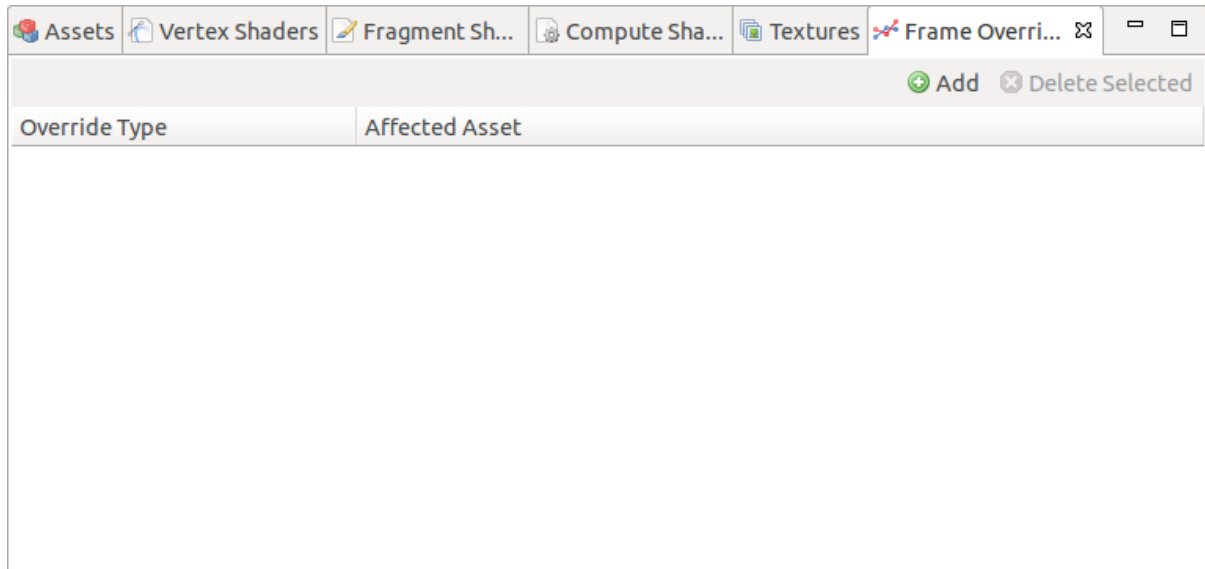
Frame Replay Limitations

Frame replay will be disabled if the frame you want to replay:

- creates or deletes any OpenGL ES/EGL objects,
- changes state that had not previously been changed,
- is not complete (i.e. no `eglSwapBuffers` call),
- has calls from multiple threads/contexts,
- calls various unsupported functions.

More information about exactly why frame replay is disabled for a particular frame can be found in the Console View in MGD.

Frame Overrides



MGD has the ability to change a Frame before replaying it again on a target device. These modifications are made on the Frame Overrides View, which is shown in the image above. At present it is possible to apply the following overrides:

- [Replace Texture](#)
Replace a selected texture with a 256 x 256 pixel texture of different colors in a grid like pattern.
- [Force Precision](#)
Replace the shaders of a program with a version that forces a specific precision for all types.
- [Modify Shaders](#)
Replace both the fragment and vertex shaders of a program with custom versions.

Replace Texture



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

Force Precision

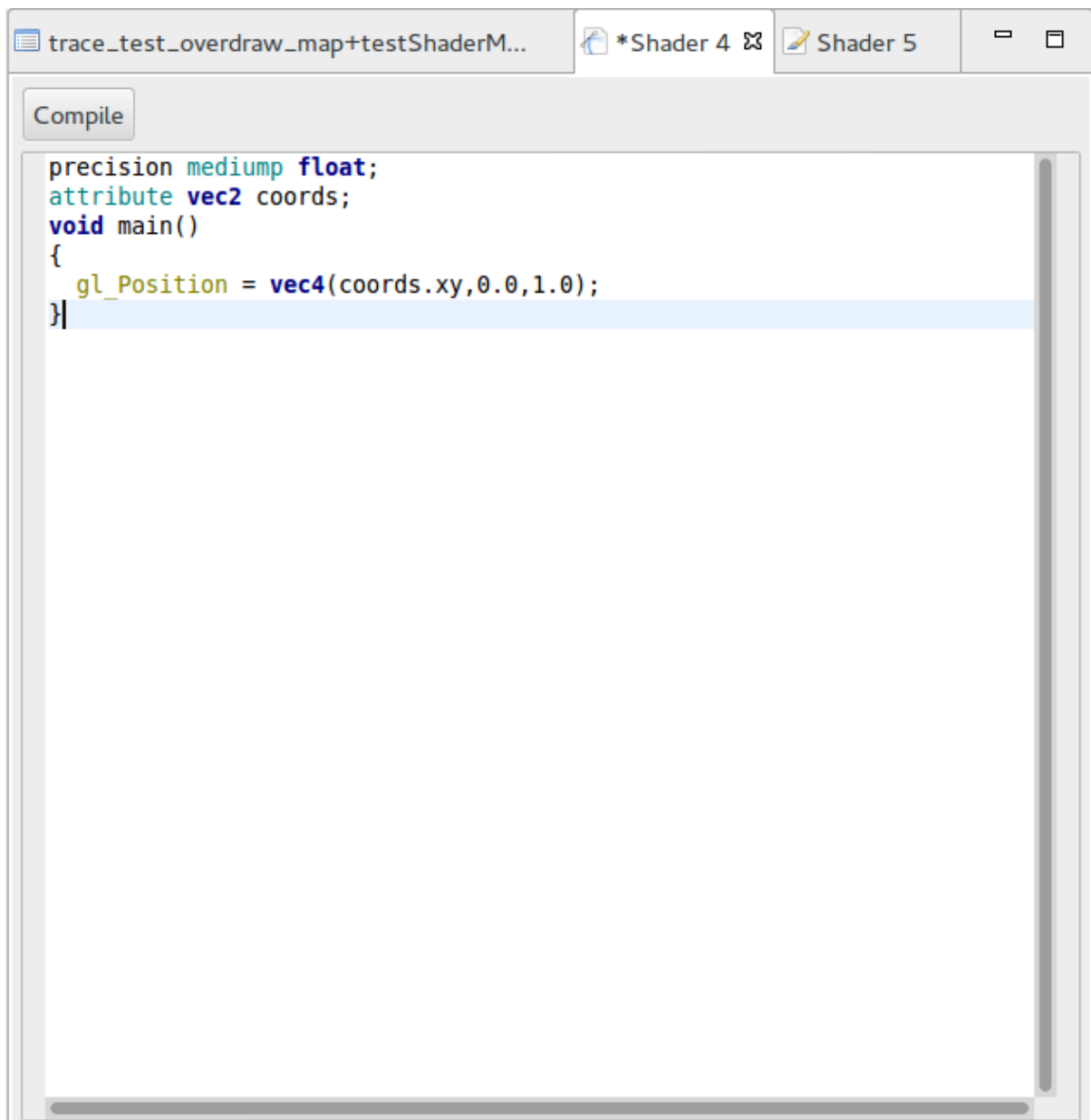
For a given program, the fragment and vertex shader are modified so that any precision specifiers (`highp`, `mediump` and `lowp`) are replaced with the precision specified by the user, and default precision modifiers are applied to the source for the following types:

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

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

Modify Shaders

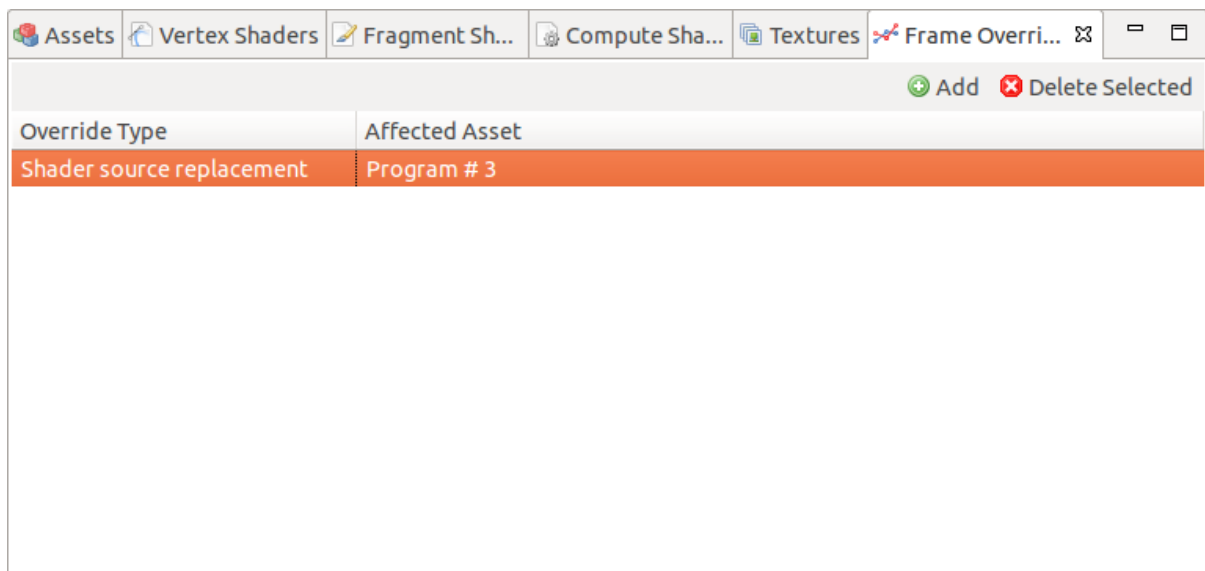
This override allows you to change both the fragment shader and vertex shader of a particular program. When you select a program and select finish on the wizard, two text editors will be displayed. One of the editors corresponds to the vertex shader of the program and the other corresponds to the fragment shader of the program.




Each editor will allow you to edit that particular shader's source. Once you save your changes these new source files will be stored and be used in the next replay frame of your application. This will happen until the override itself is deleted.



Note: If you do not save your modifications then they won't be used when replaying the frame.



There is also an option to compile the source to see if there are any errors that exist in your modifications.

 **Note:** The results of the compilation can be found in the **Console View**.

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


If the changes that are made to the shader are invalid. Then the frame override will not be used and the original program will be used instead. Invalid shaders are shaders that either don't compile, or the inputs and outputs don't match the original shader.

Debugging an OpenCL Application

MGD supports tracing OpenCL applications as well as 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 MGD.

When you start or open an OpenCL trace, you will be prompted to launch the OpenCL perspective. This will adjust the visible views to only those that are supported for OpenCL. (For more information, see [Perspectives](#) on page 52.)

The Assets View will track and display contexts, kernels, memory objects, and programs. MGD tracks the relationship between memory objects and sub-buffers, which are displayed in the Assets View. MGD will warn you about dangerous overlapping sub-buffers.

 **Note:** MGD will track memory that is initialised into any memory object (using the `CL_MEM_USE_HOST_PTR` flag). It will also track calls to `clEnqueueCopyBuffer` and `clEnqueueWriteBuffer`. However, MGD does not support sending changes to mapped OpenCL memory, or changes to memory caused by kernel invocations. This means that the memory reported in MGD may not accurately reflect your application.

Because there are no conceptual "frames" in OpenCL, the Trace Outline will attribute all function calls to Frame 0 and Render Pass 0. The Trace Outline View will display the following function calls:

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

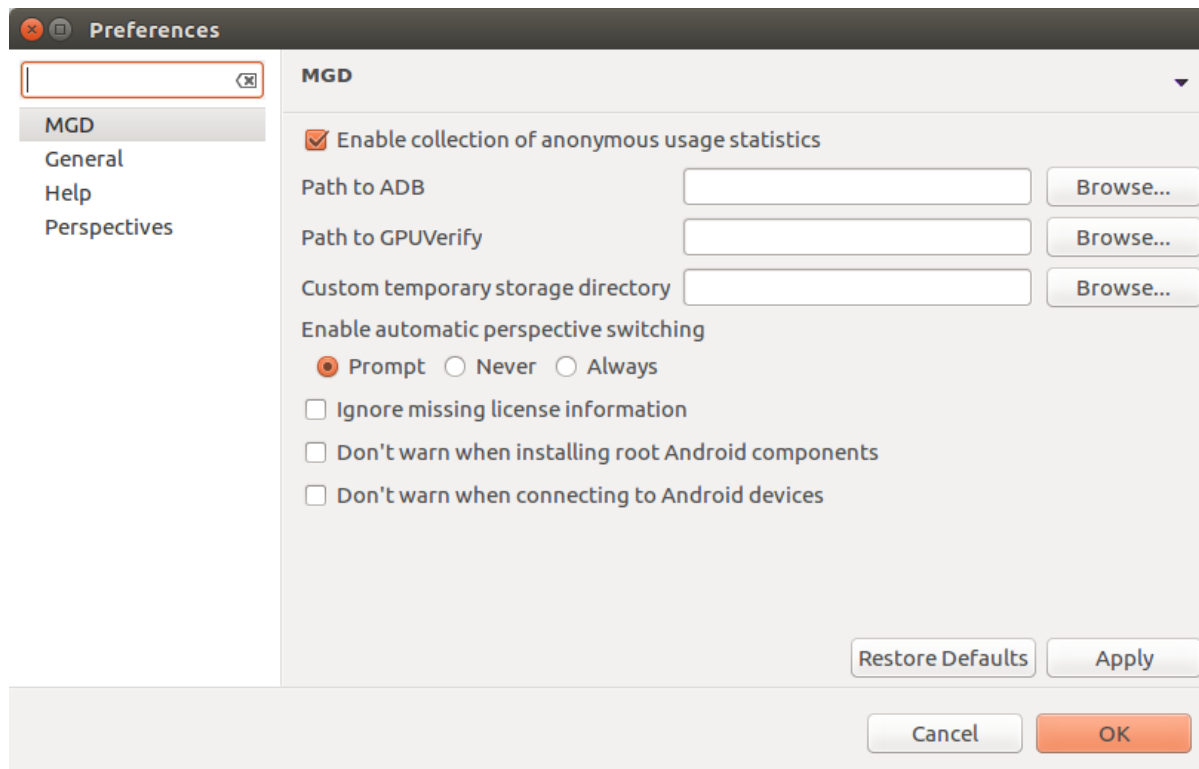
Blocking calls will tell you how long they blocked for, and also the size of the wait list passed into the function call, if applicable.

Using GPUVerify to Validate OpenCL Kernels

GPUVerify is a tool that has been developed by Imperial College London. It can be used to test whether an OpenCL kernel is free from various issues including intra-group data races, inter-group data races, and barrier divergence.

GPUVerify is a stand alone application and is not provided by MGD. It needs to be downloaded separately from Imperial College's [website](#). Before using this tool with MGD, it is recommended that the user spends time getting GPUVerify working in a stand alone context following its own documentation.

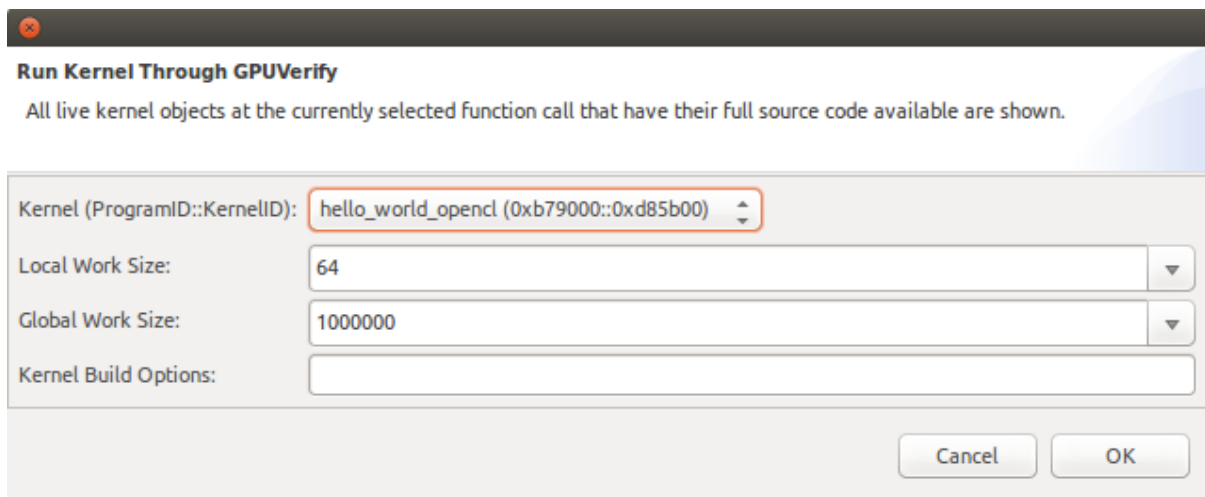
To use this tool with MGD you first need to point MGD to the binary in your GPUVerify directory. To do this click on **Edit > Preferences** and fill in the **Path to GPUVerify** field.



The **Apply** and **OK** fields will be grayed out if you enter an invalid path in this location.

When tracing a OpenCL application, MGD will capture calls to all OpenCL 1.2 functions, including `clCreateProgramWithSource`, `clCreateKernel`, `clCreateKernelsInProgram`, and `clEnqueueNDRangeKernel`. It will then use the data from these functions to get the names for all the OpenCL kernels and the source code associated with them, as well as any runtime parameters that are known at this point, such as the local and global workgroup sizes.

If you want to run any of the kernels found in a trace through GPUVerify you need to select a function call in the trace with live kernel objects, then click on **Debug > Launch GPUVerify**. You will be presented with a dialog that looks like the one below.



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 MGD for kernels that have been created from a program that was created using `clCreateProgramWithSource` followed by `clBuildProgram`. If the kernel you expect to see isn't 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.

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

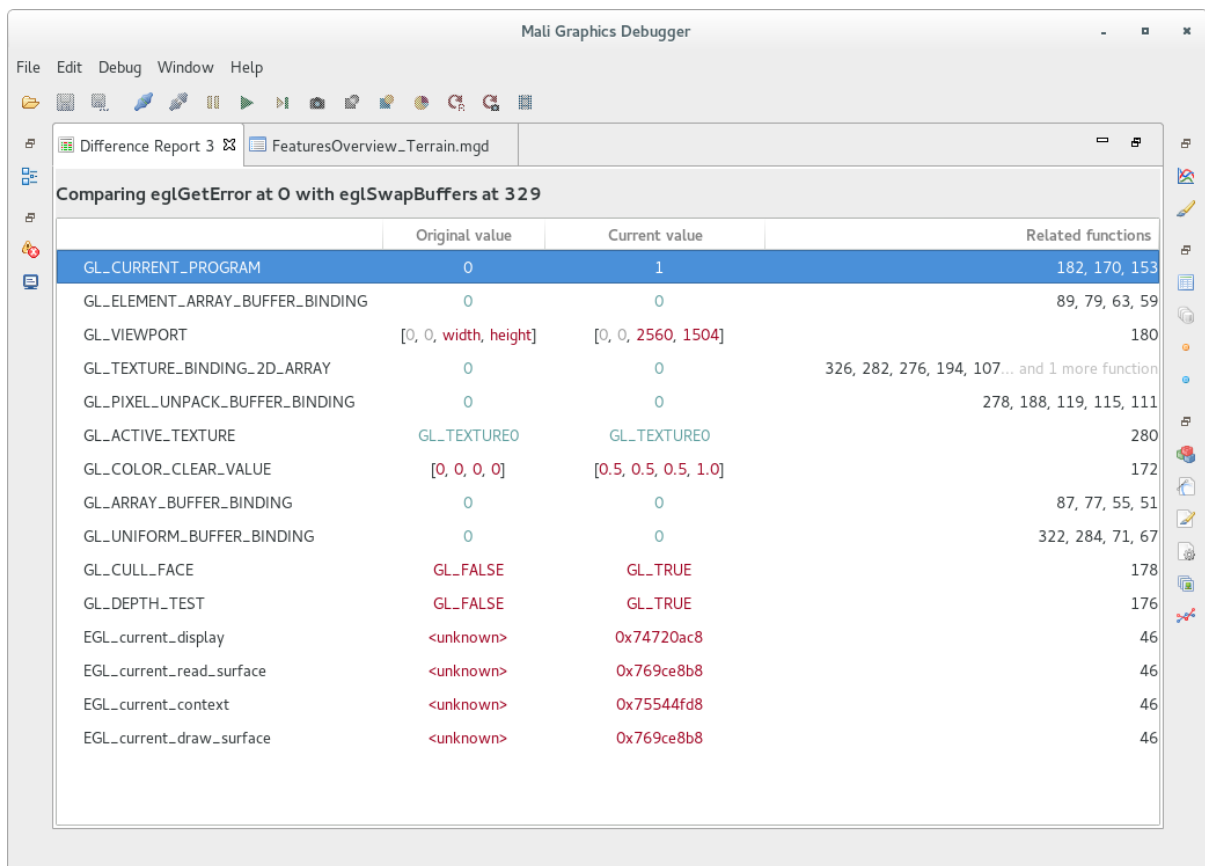
The output for GPUVerify is shown in the MGD **Console View**.

Comparing State Between Function Calls

During the course of investigation of an application trace, the user may wish to compare the API state between two function calls to examine what has changed. Users may do this using the **Generate Diff** command.

To compare changes in state between two functions in the trace either select two function calls from the trace using control+click on Windows / Linux hosts, or command+click on OS X hosts, or select two draw calls from the Trace Outline View, then select **Generate Diff** from the popup menu.

Difference Report View



The difference between state for the two function calls is shown in a new view which is created as a result of running the **Generate Diff** command. This view shows a table of the items that are different, or have changed at some point between the two functions. Values that have changed, but have since reverted to the original value are highlighted in light blue, whereas values that are different between the two functions are highlighted in red. Where state values are made up of multiple components (for example GL_VIEWPORT), the subcomponents are highlighted individually. In this case, subcomponents that have not changed are shown with gray text.

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

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

Bookmarks


MGD contains a Bookmarks feature to allow the user to bookmark particular function calls and optionally add notes to the bookmark. These bookmarks can be saved and loaded with the trace. This can be used, for example, to make notes on a function call that looks like it could be a candidate for optimization. The user can add a bookmark with a note to remind them in the future.

Bookmarks can be viewed and manipulated in the [Bookmarks View](#) and the [Trace View](#).

Dealing With VR Applications


Virtual Reality (VR) applications have a peculiar pattern in terms of OpenGL ES calls. They usually have multiple threads and/or contexts to handle different steps: scene rendering, barrel distortion, chromatic aberration, etc. MGD shows every context using a different color to make it clear what is part of which context. This applies to the [Trace View](#), the [Trace Outline View](#) and the [Timeline View](#).

When you pause a VR application or a generic multi-threaded application, pausing will be delayed over the frame end until all render passes (including those from other threads) have finished. The function calls and render passes traced after the frame end will be shown as part of the next frame in the [Trace](#)


Outline View. That frame is considered incomplete and is marked with the  icon until the application is resumed and the end of the frame is reached.


Tracing an Application That Is Already Running


Previous versions of the Mali Graphics Debugger could be attached to a running process but would be missing any data (assets/state, etc.) that would have been traced prior to starting intercepting. Starting with version 3.5.0 it is possible to attach to a process that is already running and recover almost all of the state of the OpenGL ES and EGL APIs, despite having not intercepted all of the calls from the application. Furthermore it is now also possible to attach and detach from the same process multiple times within a single trace. This allows skipping tracing parts of the application that are not interesting.

 **Note:** The Frame Replay and Automated Trace features are not available if MGD is attached to an already running process or reattached to a detached process.

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

To detach from an application that is being traced, select the  toggle button from the toolbar. When the application is detached from the debugger, the button will be depressed. To reattach, simply toggle the same button. Function calls that contain information about the target API's state following an attach

will be marked with the icon  as will frames in the outline view. A **bookmark** will also be generated for the reattach function to aid navigation.

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

A non-exhaustive list of things that may not be recovered are:

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



Warning: OpenCL and Vulkan applications are not supported.

Chapter

7




Exploring Your Application

Topics:

- [*Perspectives*](#)
- [*Trace View*](#)
- [*Trace Outline View*](#)
- [*Timeline View*](#)
- [*Statistics View*](#)
- [*Function Call View*](#)
- [*Trace Analysis View*](#)
- [*Target State View*](#)
- [*Buffers View*](#)
- [*OpenGL ES Framebuffers View*](#)
- [*Vulkan Frame Capture View*](#)
- [*Assets View*](#)
- [*Shaders View*](#)
- [*Textures View*](#)
- [*Images View*](#)
- [*Vertices View*](#)
- [*Uniforms View*](#)
- [*Automated Trace View*](#)
- [*Render Pass Dependencies View*](#)
- [*Bookmarks View*](#)
- [*Console View*](#)
- [*Full Trace Replay*](#)
- [*Scripting View*](#)
- [*Filtering and Searching in MGD*](#)

Perspectives

MGD includes a *perspectives* feature which allows related windows to be grouped together for ease of use. Out of the box, MGD comes with three perspectives:

-  OpenGL ES + EGL (default)
-  Vulkan
-  OpenCL

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

You open new perspectives by selecting the  button. You can switch between perspectives at any time using the perspective switcher located in the top right.

By default, MGD will prompt 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 > MGD** to either never prompt you, or to always automatically switch perspectives when a different API is detected.

You open create new perspectives by **Right clicking** on an existing perspective and selecting **Save As....** Custom perspectives can be removed in **Edit > Preferences > Perspectives**.


You can customize perspectives by moving, resizing, opening (**Windows > Show View**), and closing views. Customizations will be saved when MGD is closed.

Trace View


The main window in MGD shows a table of function calls made by your application as it is running. Use this to examine exactly what your application was requesting of the graphics system and what that 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 is the time spent in the driver for the function call and not how much time the GPU spends doing the work requested by the function call.

- the list of arguments sent to the call

 **Note:** This list will be truncated to save space. For a complete list of the arguments, see the [Function Call View](#) on page 56.

- 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 the user has added to the function call (see [Bookmarks](#) on page 53).

 **Note:** Some columns in this table may be initially hidden, click the  button to enable or disable columns.



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

The Trace Outline shows a frame-oriented view of the trace. Each frame is delimited by a call to `eglSwapBuffers()`. Draw calls in a frame are grouped by the framebuffer bound at the time they were called. Selecting an item in the overview will highlight 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 selection **Open Documentation**.

It is possible to select two function calls from the trace using control+click on Windows / Linux hosts, or command+click on OS X hosts. Menu-clicking on one of the two selected functions will bring up a popup menu showing various options including the ability to [generate a state difference report](#). Alternatively the user may 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), MGD 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. Simply type your search string in the search box and MGD will highlight matching calls in the trace. You can use the  and  buttons to jump between the search results.

Pressing the  button or pressing the **Esc** key will close the search and hide the results.

By default the search only looks at the function call name. If you want to search the function call parameters as well you can select the **Include parameters** option. With this selected, MGD will search the functions exactly as they appear in the *Function Call* column of this view.

See [Searching and Filtering in MGD](#) for more information.




Bookmarks

For more general information on bookmarks please see [Bookmarks](#).

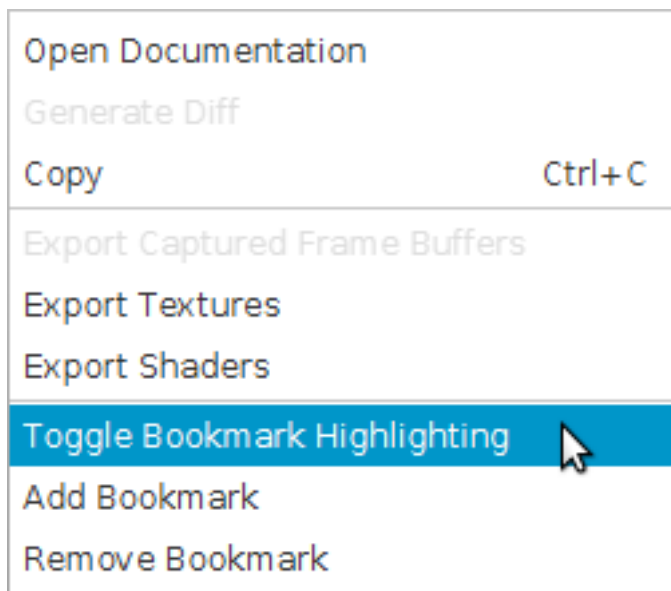
In the Trace View window there is a column labeled **Bookmark Notes** which lets you view and edit Bookmarks for each function call.

To add a bookmark with a note **Double Click** on the notes area for that function call and enter you note. To add an empty bookmark **Right Click** on the function call and select **Add Bookmark**. To add remove a bookmark **Right Click** on the function call and select **Remove Bookmark**.

Every marked function will then appear green in the trace and a new marker will be placed on the ruler on the right hand side of the trace. This allows quick navigation to areas with notes.

#	Error	Return	Function Call	Notes
92	GL_NO_ERROR		glUniform1i(location=2, v0=4)	
 93	GL_NO_ERROR		glDrawElements(mode=GL_TRIANGLES, count=6, type=GL_UNSIGNED_SHORT, ind	
94	GL_NO_ERROR		glUniform1i(location=2, v0=5)	
 95	GL_NO_ERROR		glDrawElements(mode=GL_TRIANGLES, count=6, type=GL_UNSIGNED_SHORT, ind	
 96	EGL_SUCCESS	EGL_TRUE	eglSwapBuffers(dpy=0x1, surface=0x84a5d20)	End of Frame

This highlighting option can be disabled by right clicking anywhere in the Trace View window and unchecking the option that says **Toggle Bookmark Highlighting**.



Bookmarks can also be viewed and manipulated in the [Bookmarks View](#)

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 to easily navigate through the trace.


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

Function calls are further grouped depending on the API in use. For example, OpenGL ES calls are grouped into frames and render passes. For each item (including the groupings) in the tree we show the name, index, and additional interesting information for that item. For certain items, even more information can be found in the item's tooltip.

Selecting an item in the tree will cause other views to be updated to provide information about that item. Selections made in other navigation views (the Trace View, Breadcrumb Bar, or the Statistics View Charts Tab) will 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 will be shown to indicate where the item would be in relation to other items.

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

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

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

Right clicking on items in the tree allows you to generate a diff report between two items and export framebuffers. See [Comparing State Between Function Calls](#) on page 47 and [Asset Exporting](#) on page 61 for more information.

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

Timeline View

The Timeline View shows a graphical representation of the function calls in your application. There are three types of timeline view:

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

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

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

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 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 will show you statistics about all the processes in the current trace. The currently selected item will be highlighted in the chart and the item's parent is shown as the chart's title. The following actions are available for the charts:

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



Note: Some items do not support all the available statistics. For example, Render Passes do not support the *Number of render passes* statistic.

Memory

This tab shows information on memory usage for each frame. This data can be produced on Mali T600 or later based devices, but the vendor may choose not to enable this feature. To check if this feature is supported, please check if your device contains one of the following files: `/sys/kernel/debug/mali0/ctx/*/mem_profile` or `/sys/kernel/debug/mali/mem/*`. If the files are present and non-empty then your device is supported.



Attention: To allow this data to be processed by MGD, please turn off the SELinux permissions by running `'setenforce 0'` on your device.



Attention: You may need to mount the Linux "debugfs" mount point for this feature to work:
`mount -t debugfs /sys/kernel/debug /sys/kernel/debug`

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

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

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

Function Call View

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

Arguments

This tab shows the unabridged arguments for the selected function call alongside their values. This can be useful since, due to the limited size of the [Trace View](#), the values of arguments will be 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 will have additional information.

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

Documentation

This tab shows the Khronos documentation page for the selected function call if it's available.

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 (passing illegal arguments, for example) 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 will highlight the offending issue(s) within the trace view. Hover your cursor over the problem report to gain a more detailed view of the problem (if available).

Target State View



Attention: This view is only available in the  **OpenGL ES + EGL** [perspective](#) by default.

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

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

If a state item has been changed anywhere in the trace then it will be 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 are 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.

Filtering

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 will only show you states which have 'texture' in the name and/or those which have 'texture' in the value. See [Searching and Filtering in MGD](#) for more information.

In addition to the **Filter** box, there are six 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 will be shown.
- **States that are not currently their defaults** - Only states which, at the currently selected function call, are not at their default values will be shown.
- **States changed by this function** - Only states changed by the currently selected function call will be shown.
- **Read-only states** - Only read-only constant states such as `GL_MAX_DRAW_BUFFERS` will be shown.

Buffers View

The view shows information about the currently allocated buffer objects. The list of buffer objects can be filtered by usage, or by the last bound target column (for GLES only). 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 buffers objects is shown.


Filtering

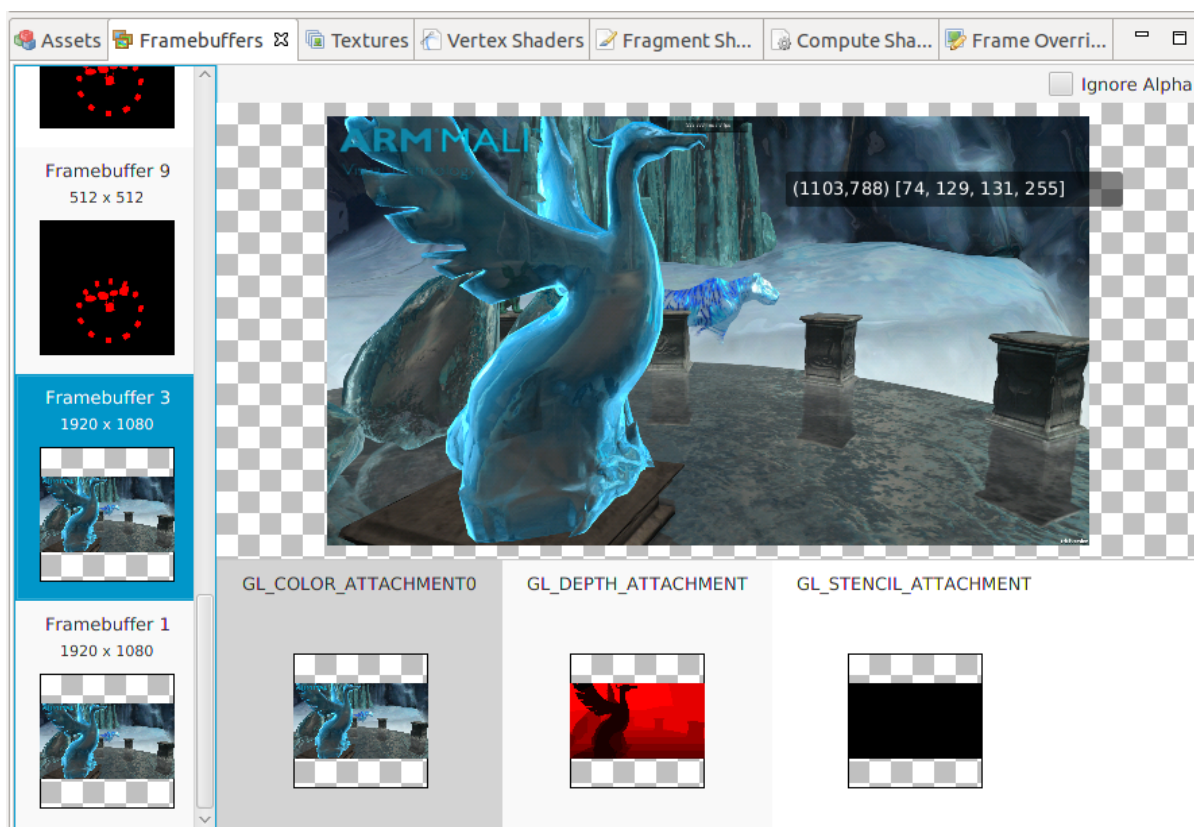
To filter the view to show only the buffer objects with a given usage or last bound target, use the **Filter** box. For example, type *transform* into the box and the view will only show you the buffer objects which have 'transform' in the usage field. See [Searching and Filtering in MGD](#) for more information.

OpenGL ES Framebuffers View



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

Framebuffers can be captured using the  *Capture Frame* button (see [Capturing Framebuffer Content](#) on page 38). Once you have a frame with a successful frame capture, the results can be found in the Framebuffers View. In the view, framebuffers in use at the currently selected point in the trace. Use this to gain an insight into what the graphics system has produced as a result of your application calls.



Selecting a framebuffer in the left-hand list will bring up a list of the attachments used in that framebuffer in the lower list as well as histograms displaying the overdraw or shader map data, if relevant. Selecting an attachment will bring up a larger view of that attachment. In case of multiview rendering, the views will be displayed as separate selectable elements in the attachments list. You can then mouse over the attachment for additional information. Double clicking the main image will open up the attachment in an external editor.


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


- **Use Alpha** does normal alpha blending using the alpha values in the framebuffer.
- **Ignore Alpha** ignores the alpha values in the framebuffer and sets the alpha value for each pixel to its maximum (i.e. fully opaque).
- **Visualize Alpha** ignores the color information in the framebuffer and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

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

Vulkan Frame Capture View






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

Just as for OpenGL ES applications, framebuffers for Vulkan applications can be captured using the  *Capture Frame* button (see [Capturing Framebuffer Content](#) on page 38). For all calls to `vkQueueSubmit` within the captured frame, MGD will capture each color, resolve, and depth/stencil framebuffer attachment that has been modified by each draw call inside each submitted command buffer.

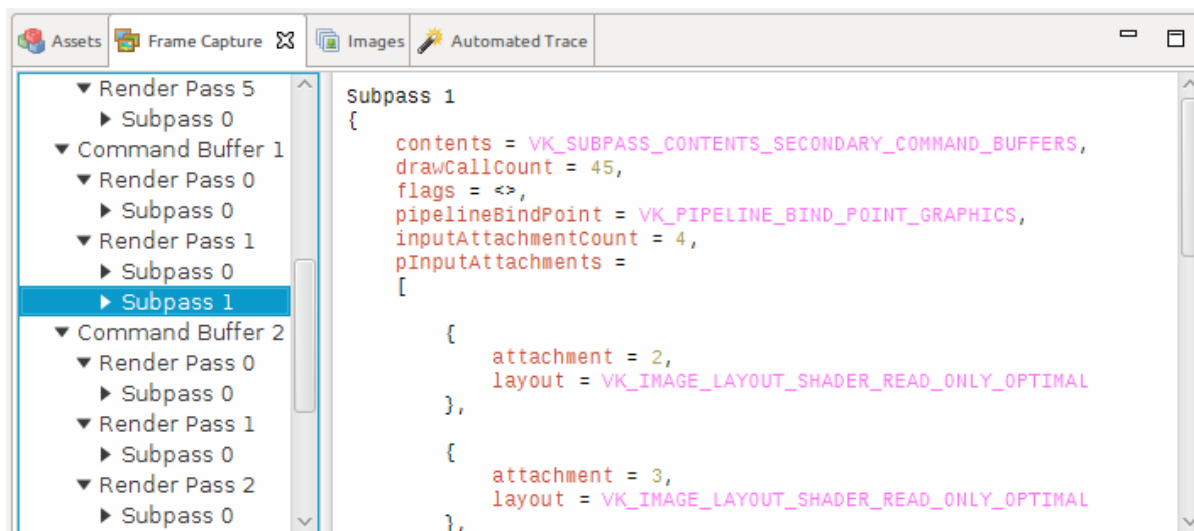
Framebuffer attachment images that were created with anything other than the `VK_SAMPLE_COUNT_1_BIT` as the `samples` parameter will not be captured. However, if the

multisampled image has a corresponding resolve attachment in the render pass, the resolve attachment will be captured after each draw call.

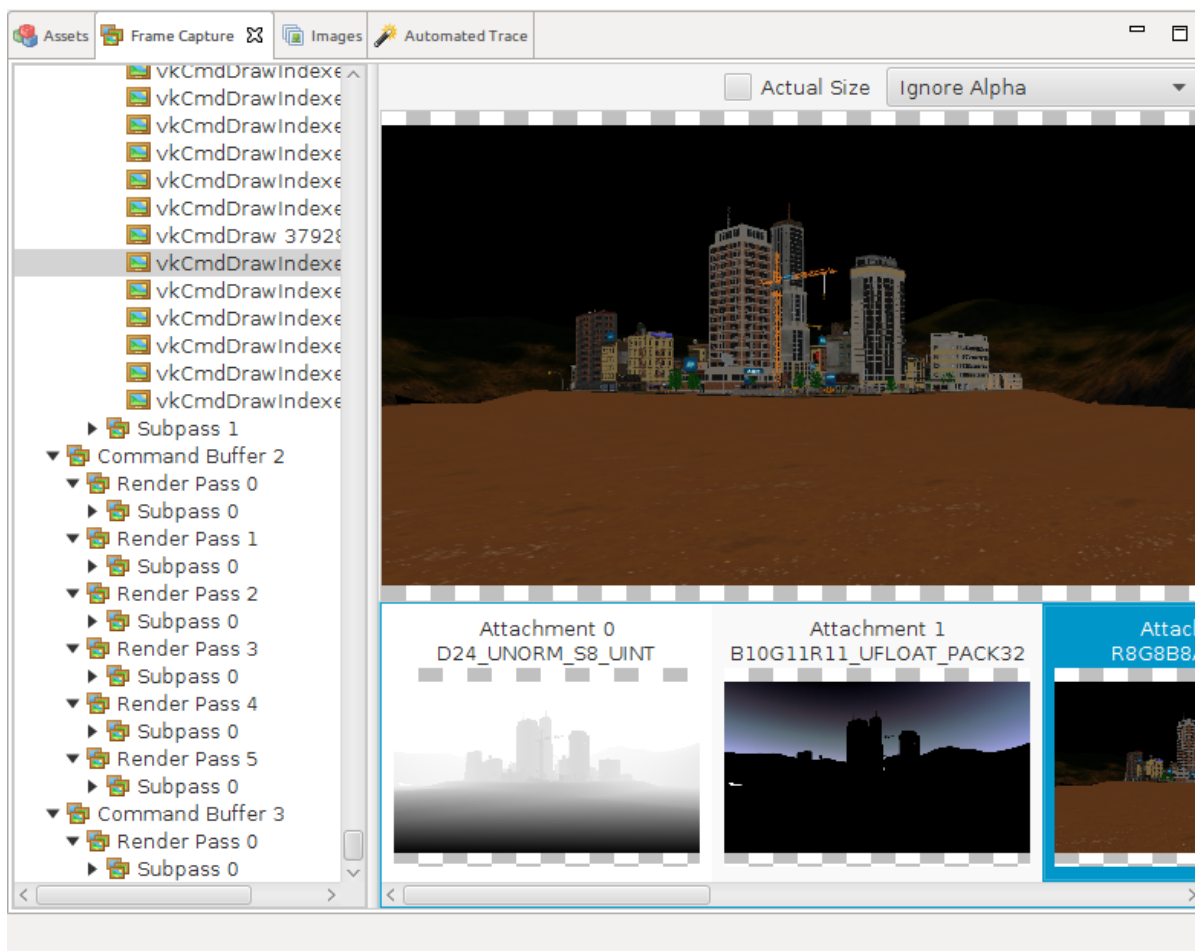
Currently, the  *Overdraw*,  *Shadermap*, and  *Fragment Count* capture modes are unsupported for Vulkan applications, and the buttons will be disabled when tracing an application that contains only Vulkan function calls.


Additionally, the  *Capture All Framebuffer Attachments* button is disabled, as by default all framebuffer 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 will show information about that tree item.



If the currently selected call to `vkQueueSubmit` is within a captured frame, selecting a draw call will display the contents of all captured framebuffer attachments after that draw call was executed.



The  icon will be displayed next to tree items when framebuffer attachment data has been received. Captured framebuffer attachments can be viewed as soon as the data has been received by the host, so it is not necessary to wait until the **Capturing frame** dialog has completed to start examining the data.

In certain situations you might want to view the framebuffer 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 (i.e. fully opaque).
- **Use Alpha** does normal alpha blending using the alpha values in the attachment.
- **Visualize Alpha Channel** ignores the color information in the attachment and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

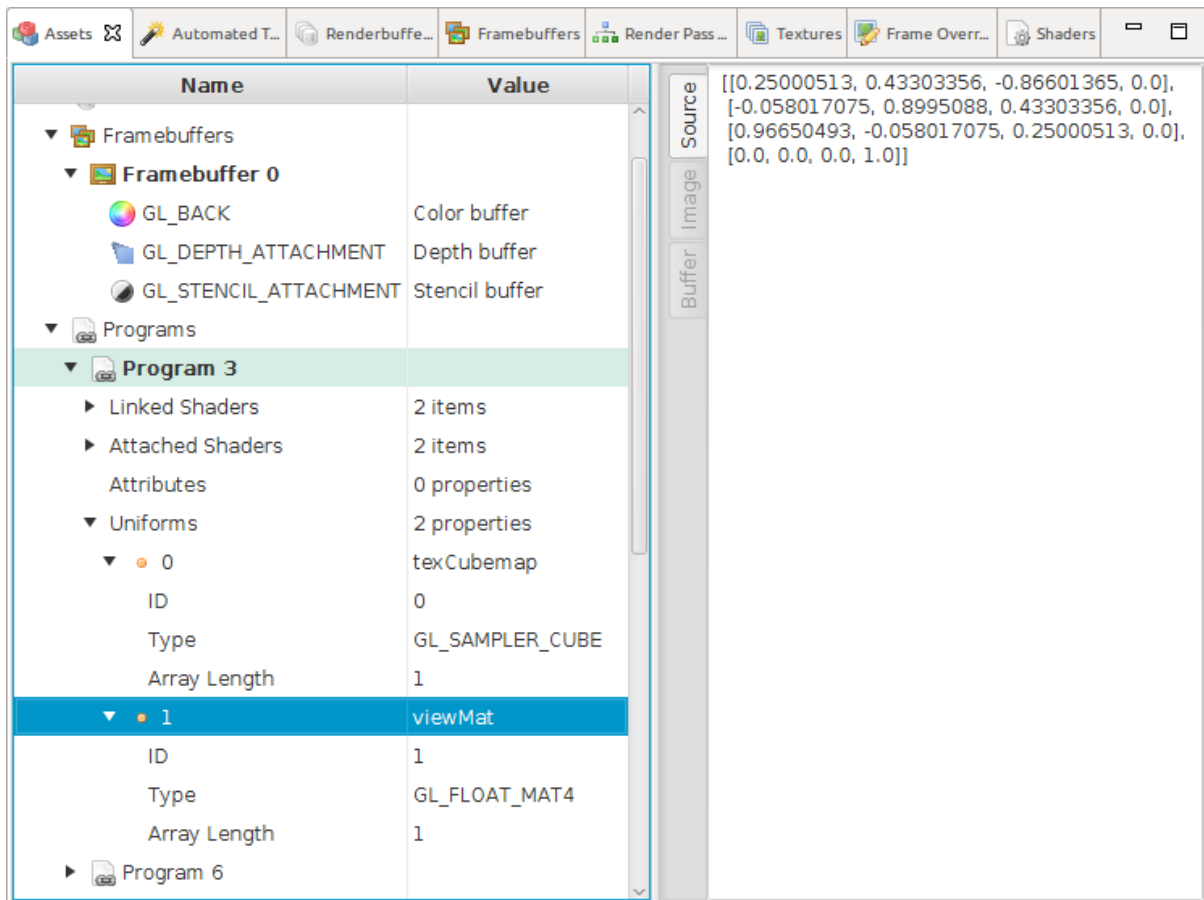
The stencil component of combined depth/stencil attachments will be visualized in the alpha channel of the displayed image.

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

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 to explore the API objects that your application has created and is currently 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 will become selectable and will display the asset data.



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

- Assets highlighted in green were created in the currently selected function call.
- Assets highlighted in light green were modified in the currently selected function call.
- Assets that are considered "active" (such as the currently bound OpenGL ES framebuffer) 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.

Asset Exporting

Assets can be exported from your application trace to disk.

Framebuffers can be exported by:

- Selecting **File > Export All Captured Framebuffers** (only enabled if OpenGL ES function calls are present in the application trace).
- Selecting the frames, render targets, or draw calls containing the captured framebuffers you want export from the Trace Outline View, right clicking, and selecting **Export Selected Captured Framebuffers**. This option will be grayed out if your selection does not contain any captured framebuffers.

Textures can be exported by:

- Right clicking on a single function call and selecting **Export Textures**. This will export all textures that existed at the time of that function call.
- Selecting the textures you want to export from the Textures View, right clicking, and selecting **Export Textures**.

Shaders can be exported by:

- Right clicking on a single function call and selecting **Export Shaders**. This will export all shaders that existed at the time of that function call.
- Selecting the shaders you want to export from the Shaders View, right clicking, and selecting **Export Shaders**.

Shaders View



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

This view is an alternative tabular view of all currently loaded shaders. For each shader various cycle counts are shown allowing you to identify which shaders are most costly. These cycle counts are calculated using the Mali Offline Shader Compiler for the Mali T760 GPU.

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

$$Cycles = \max(a, ls, t)$$

$$a = (\text{arithmetic_shortest_path} + \text{arithmetic_longest_path}) / 2$$

$$ls = (\text{load_store_shortest_path} + \text{load_store_longest_path}) / 2$$

$$t = (\text{texture_shortest_path} + \text{texture_longest_path}) / 2$$

For fragment shaders, the *Fragments* column (and other dependent columns) will be empty unless the *Fragment Count* feature was active for the selected frame.

Active shaders are shown in bold type.

Textures View



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

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

Loading textures is done using an external program: note that for larger traces the application can take a short time to convert and display all textures. The following texture formats are currently 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 it's maximum (i.e. fully opaque).
- **Visualize Alpha** ignores the color information in the texture and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

Images View



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

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

Loading images is done using an external program: note that for larger traces the application can take a short time to convert and display all images. The following image formats are currently 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

Also, the view has the following limitations:

- only one layer is displayed for multiple layers 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/to buffer subregions and from/to specific image subresource 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 it's maximum (i.e. fully opaque).
- **Visualize Alpha** ignores the color information in the image and shows the alpha values for each pixel. The alpha values are shown in a range from black (minimum alpha/fully transparent) to white (maximum alpha/fully opaque).

Vertices View



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

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.

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 will be used to provide the values. The vertex indices used in this view have been sorted and duplicates removed.

Indices Tab

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

Geometry Tab

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

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

In order to render the correct geometry, MGD must know which one of the shader attributes corresponds to the geometry position data. You can select this from the **Position Attribute** choice box. MGD will use 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: Currently the Geometry viewer (and export function) only works with the GL_TRIANGLES, GL_TRIANGLE_STRIP, and GL_POINTS draw modes. When using GL_POINTS, each point will be rendered as a small tetrahedron. Primitive restart is also not supported, if you are using this feature then you may 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 you can use the mouse scroll wheel or the W and S keys.

To move the camera you can click and drag with the secondary mouse button or use the A and D keys (left and right) and the Q and E keys (up and down).

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

Exporting

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



Note: Wavefront .obj files do not support triangle strips so MGD will convert any triangle strip data to a series of individual triangles when exporting.



Note: Wavefront .obj files do not support points so MGD will convert any point data into a series of small tetrahedrons when exporting.

Uniforms View




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

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.

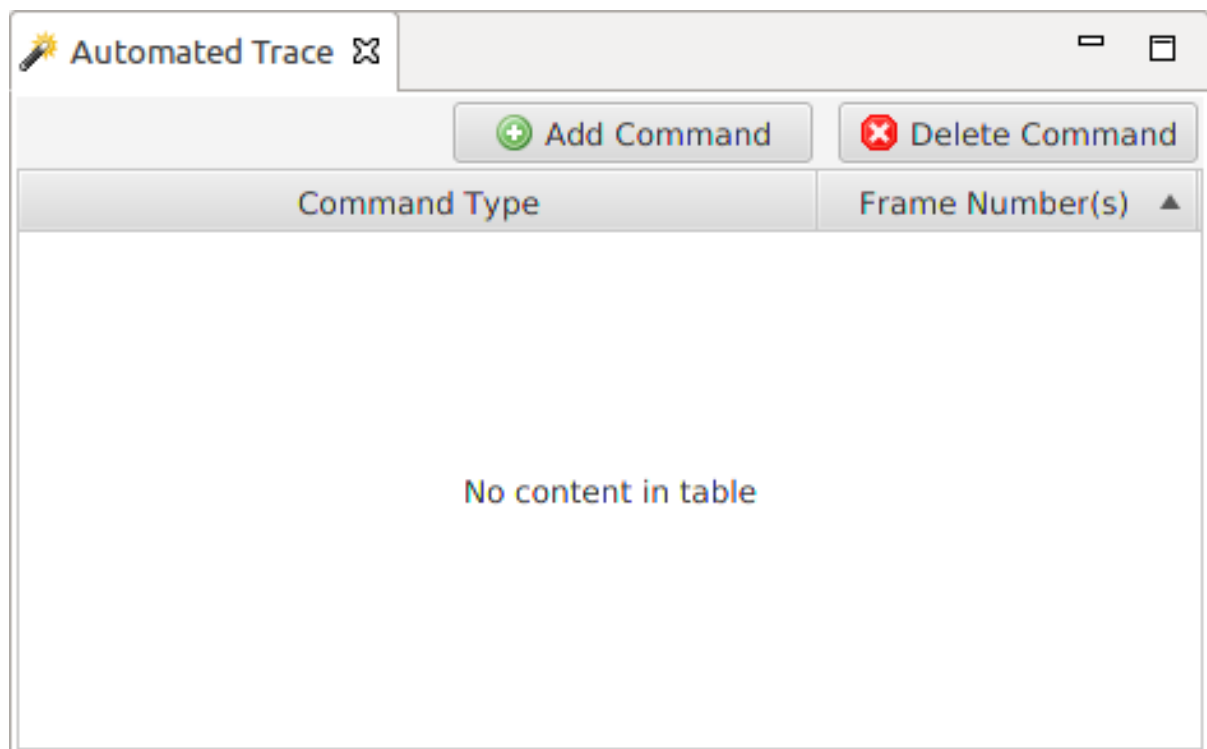
For each active uniform, we shown its index, location, type, and value. If the uniform is a block, the block name and the block buffer binding will be shown.

Automated Trace View

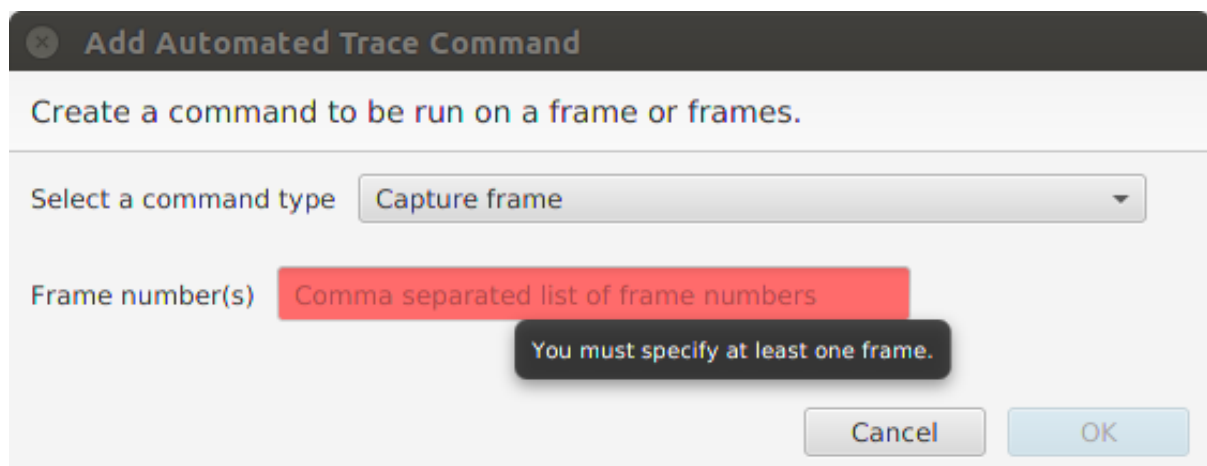
The Automated Trace View allows you to run a range of standard MGD commands automatically when a certain frame is encountered. For example, you could run your application and automatically take a frame capture on frame 10, do a frame capture with overdraw mode switched on at frame 20 and do a frame capture with fragment count mode enabled at frame 30.

 **Note:** Currently you can only add automated trace commands after an application has started.

To add an automated trace command, first select and pause the process you want to add commands to and then open up the **Automated Trace View**.




Select **Add Command** and the *Add Automated Trace Command* dialog will be opened.



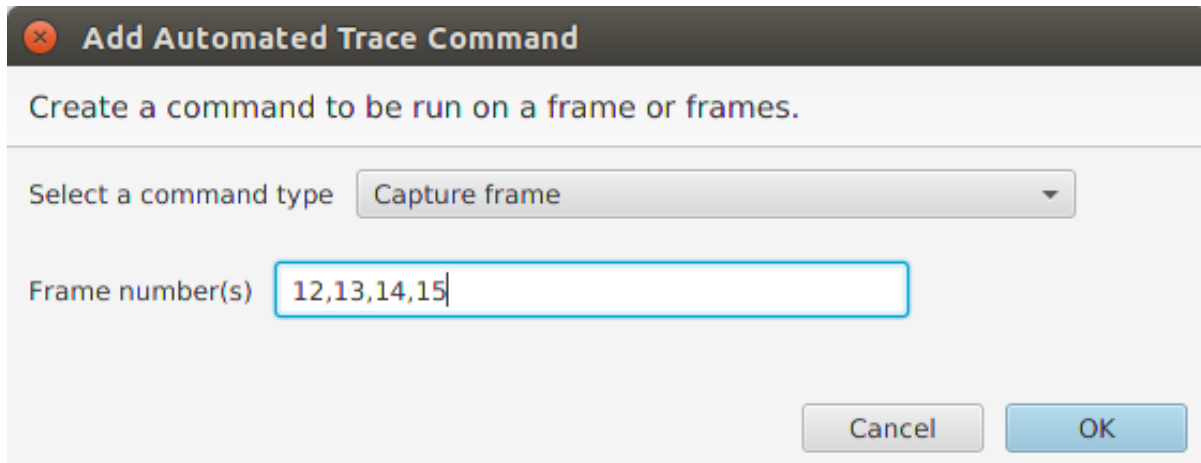
Here you can select the type of command you want and which frames it should apply to. You must specify at least one frame number to add a command. For a frame number to be considered valid, it must:

- 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 will be highlighted in red. A tooltip on text box will give you the reason why.

 **Note:** 'Empty' frame numbers (a series of commas with no number between them) and duplicate frame numbers are ignored.

When you have a valid list of frames, select the **Ok** button.



Add Automated Trace Command

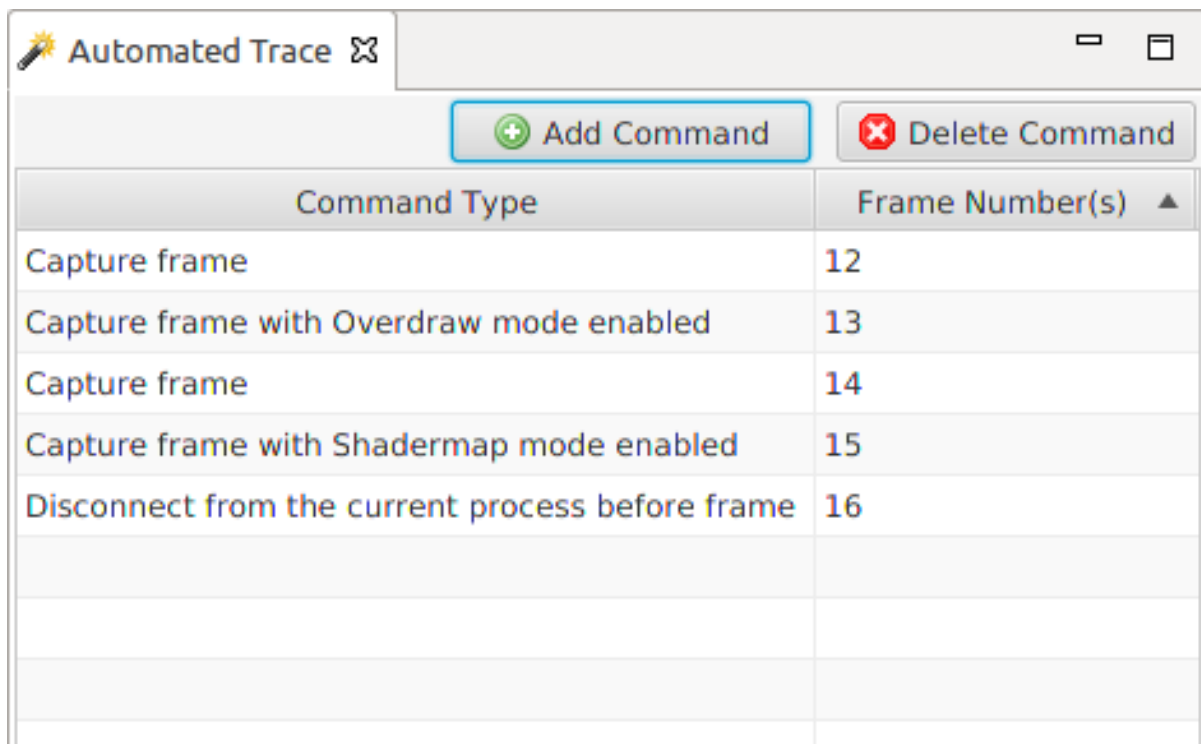
Create a command to be run on a frame or frames.

Select a command type: Capture frame

Frame number(s): 12,13,14,15

Cancel OK

You can then add more commands (and remove existing ones).



Command Type	Frame Number(s)
Capture frame	12
Capture frame with Overdraw mode enabled	13
Capture frame	14
Capture frame with Shadermap mode enabled	15
Disconnect from the current process before frame	16

When you are happy with the list, press the  button. When the trace reaches frames which you have added commands to, those commands will be executed.

 **Note:** Automated trace commands for a frame will be ignored if:

- you send a play, step, or capture command in that frame
- you send a play, step, or capture command in the frame before that frame

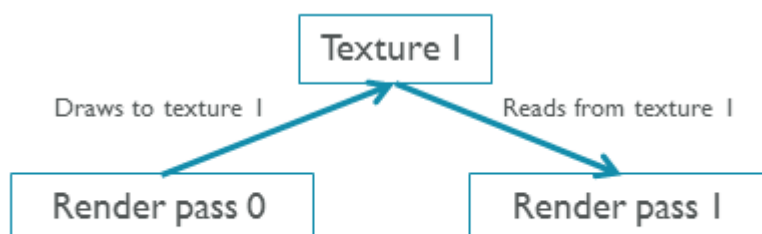
Render Pass Dependencies View



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

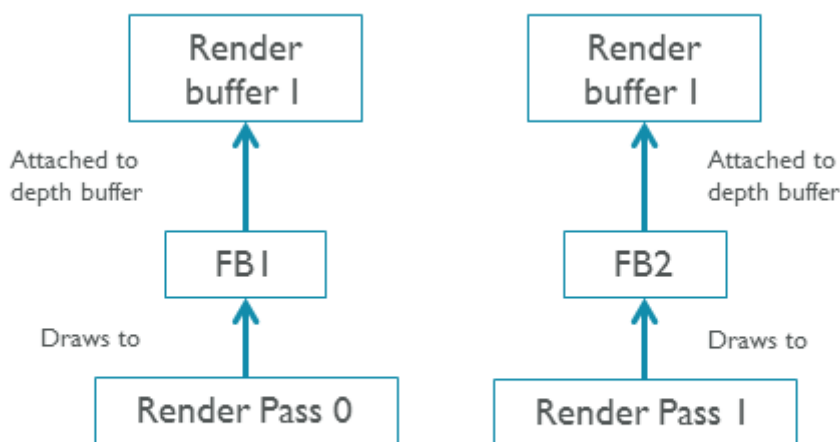
MGD can work out what dependencies there are between different render passes in a selected frame. The different types of dependencies it can detect are as follows:

1. If a render pass reads from a texture that is wrote to in a different render pass without being cleared. E.g. render pass 0 draws to texture 1 and render pass 1 then reads from texture 1.



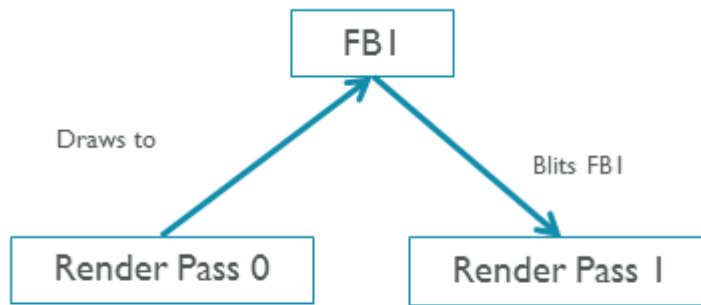
Render pass 1 has a dependency on render pass 0 due to texture 1

2. 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. E.g. render pass 0 has render buffer 1 attached to its depth target and render pass 1 also has render buffer 1 attached to its depth target.



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

3. If a render pass does a glBlitFramebuffer call on a different framebuffer. E.g. render pass 0 draws to framebuffer 1 and render pass 1 blits framebuffer 1 into framebuffer 2.



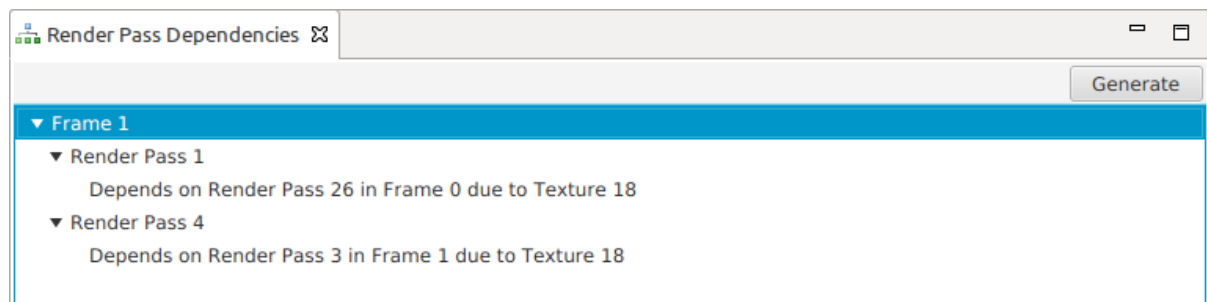
Render pass 1 has a dependency on render pass 0 due to a blit of FBI

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 any render pass in the selected frame that is dependent on another render pass will be shown in the tree. Expanding the render pass will tell you:

- Which render pass it's dependent on
- Which frame that render pass is in
- Why MGD 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.

As a example, in the following screen shot, we can see that two of the render passes in Frame 1, Render Pass 1 and Render Pass 4, have dependencies on previous render passes. Render Pass 1 is dependent on Render Pass 26 in the *previous* frame (Frame 0). Render Pass 4 is dependent on Render Pass 3 in the *current* frame (Frame 1). In both cases there are dependencies because Texture 18 is attached to the active framebuffer for the render passes.





Bookmarks View

This topic describes how to view and manipulate Bookmarks in the Bookmarks View, for more general information on bookmarks please see [Bookmarks](#).

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

You can jump to the function call associated with a specific function call by clicking the **Go to Function** button next to the bookmark.

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 will add an empty bookmark to the currently selected function call in the Trace View. Pressing the  **Remove Bookmark** button will

remove the selected bookmark in the Bookmarks View. You can edit a bookmark by **Double Clicking** on the notes area next to the bookmark.

Bookmarks can also be viewed and manipulated in the [Trace View](#)

Console View

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

Full Trace Replay



Attention: This feature requires a [DS-5 license](#) to use.

Full Trace Replay is a feature of the MGD Android application. It allows you to replay a full MGD trace on a target device.

Usage

To use Full Trace Replay:

1. Trace the OpenGL ES 3.2 application you want to replay using MGD. The **Full Trace** Process Configuration, or a configuration that sends more types of assets than the **Full Trace** configuration (e.g. the **Everything** configuration) must be used for the entirety of the trace. See [Process Configuration](#) on page 36 for more information.
2. [Install the MGD Android application](#)
3. Push the trace file you want to replay to `/sdcard/mgd/replay/` on the target device. For example:

```
adb push trace.mgd /sdcard/mgd/replay/trace.mgd
```



Note: On some Android versions you may need to restart the MGD Android application for the trace file to appear.

4. Launch the MGD Android Application.
5. Navigate to the **Replay** tab.
6. Select the trace you want to replay.



Note: The first time you use Full Trace Replay you will have to [authorize the device](#).

Device Authorization

Any device you want to use with Full Trace Replay must first be authorized to do so. To authorize a device:

1. Ensure your MGD host application is correctly set up to use a license (see [here](#) for more information).
2. Connect to the MGD Android application from the host (see [Tracing an Android Application](#) on page 27 for more information).
3. Disconnect the MGD host.

You should see valid license information on the **Replay** tab. The MGD Android application will store the authorization for 30 days. Full Trace Replay will continue to work without reconnecting the host for those 30 days. When the 30 days has expired simply follow these instructions again.

Limitations

Full Trace Replay is only guaranteed to work in the following circumstances:

- All the function calls are part of the OpenGL ES 3.2 core spec (some common extensions are supported)
- The trace only contains calls from a single process
- The **Full Trace** process configuration (or greater) was used for the entirety of the trace
- The trace was taken on the target device you're attempting to replay on
- Windows surfaces may be redirected off-screen in case of traces creating multiple window surfaces

Scripting View



Attention: This feature requires a [DS-5 license](#) to use.

While using MGD, you may find that you would like to extract or analyze information retrieved from the target device, but the user interface doesn't have the tools that you need. For this purpose, MGD has a Python scripting environment that allows you to directly interface with MGD's trace model. You can use this to either perform an analysis natively in Python, or output the data you need to an external file.

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 you can also import modules from the Python standard library.

There is one interpreter per trace file. These interpreters cannot share data. Closing the **Scripting View** closes all interpreters.



Note: The `import` keyword is only valid for the Python standard library. The interpreter only supports loading external code in the form of scripts that must be run from the **Scripting View**. After a script completes, all top-level declarations (variables, function definitions, class definitions, etc) will persist in the scripting environment.

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

- `trace`, a representation of the MGD internal model.
- `monitor`, an interface to the progress bar underneath the interpreter input text area. You can use this to track progress inside your scripts.

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


```
help(trace)
```


MGD also comes with some sample scripts inside `/path/to/installation/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.

Pressing the  **Interrupt** button will cause any running script and any created threads to stop.


The  **Clear** button allows you to clear the output text area.


To reset the interpreter back to its original state, press the  **Reset** button.

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. MGD only loads scripting files with the `.py` extension.

The scripts you load are stored in your workspace and are persistent across runs of MGD.

To load a single script press the  **Add Script** button.

To load a directory press the  **Add Directory** button. File-system changes to this directory will be reflected in MGD.

To remove any top level item press the  **Remove** button. This does not affect the file or directory on the file-system.

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 potentially very powerful, but also potentially very memory intensive. Especially if your trace is very large, the following tips may help improve the performance of your scripts:

- Holding global references to objects that you no longer need will waste memory. You can delete an individual reference with the `del` keyword or use the Reset button to re-initialize the scripting environment, deleting all references.
- You can keep memory usage low by only touching the parts of the model that you are specifically interested in.
- Traversing the model forwards will be faster than traversing backwards.
- Printing excessively large strings to the interpreter console can slow down the MGD user interface. If you need to write large strings, you may want to write to an external file rather than the scripting console.

Filtering and Searching in MGD

Filtering and searching in MGD uses Java regular expressions. For example, type `gl_program|gl_texture` into a filter or search box and that will match both entries that contain the 'gl_texture' or 'gl_program'. Filtering and searching in MGD is case insensitive.

By default the matches are performed on substrings, e.g. `program` will match `GL_PROGRAM`. If you wish to anchor your expressions you can use the standard regular expression boundary matchers such as `^` (the beginning of a line) and `$` (the end of a line). For example, typing `program$` 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 will go red and the specific error will be shown as a tooltip.

To learn more about Java regular expressions see:

- [The documentation for the Pattern class](#)
- [The Oracle® regular expression tutorial](#)

Chapter

8

Integration with DS-5 Streamline

Topics:

- [Installation](#)
- [Using Streamline Annotations](#)

The Mali Graphics Debugger interceptor library generates [Arm® DS-5 Development Studio Streamline](#) version 5.22 or later 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:

- Active EGL context for each intercepted processes' thread is shown in the Heat Map and Core Map views.
- Per process activity view is now available which shows active contexts, frames within each context, render passes within each frame and important calls per render pass including draw calls, frame end calls and flushing calls.
- Charts showing:
 - Frames per second
 - Direct and indirect draw calls per frame
 - Vertices and instanced vertices passed to a draw call
 - Vertices per frame

Installation

The user should install and have working DS-5 Streamline version 5.22 or later. Instructions for how to do this are included with DS-5 Streamline.

Mali Graphics Debugger integration only requires the interceptor library from Mali Graphics Debugger to be installed on the target. Neither the host application nor `mgddaemon` are required if the user is only interested in DS-5 Streamline annotations.

The user should follow the instructions in [Target Installation](#) on page 15 to install the interceptor on their target device.

Once all the various setup instructions have been followed you should end up with:

- A working installation of DS-5 Streamline.
- `gator` installed and working on the target device.
- The MGD interceptor installed and working on the target device.
- Mali Graphics Debugger installed on host. *(Optional if only interested in Streamline annotations)*
- `mgddaemon` installed and working on the target device. *(Optional if only interested in Streamline annotations)*

Using Streamline Annotations

The user should launch the DS-5 Streamline application and launch `gator` on the target device, then should start a new profiling capture within DS-5 Streamline as per the Streamline documentation.

Once the trace has started to be received, the user should start the OpenGL ES application they are interested in profiling. The MGD interceptor must be loaded by the application.



Note:

For Android targets it should be sufficient to just to launch the appropriate application from the launcher.

For Linux targets, the user may need to use the [LD_PRELOAD](#) or [LD_LIBRARY_PATH](#) methods.



Attention: The `mgddaemon` daemon application should not be running when using the Streamline tracing functionality.

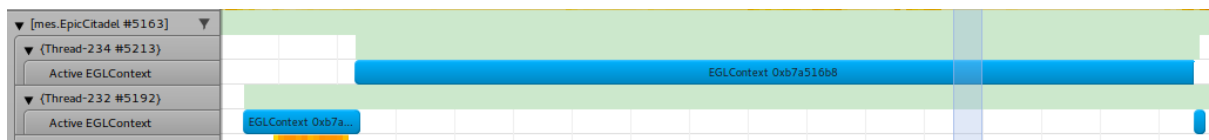


Attention: To allow MGD to collect the required data, please turn off the SELinux permissions by running `'setenforce 0'` on your device.

Once the user has finished tracing the application they are interested in, they should terminate the application and then disconnect the trace in DS-5 Streamline.

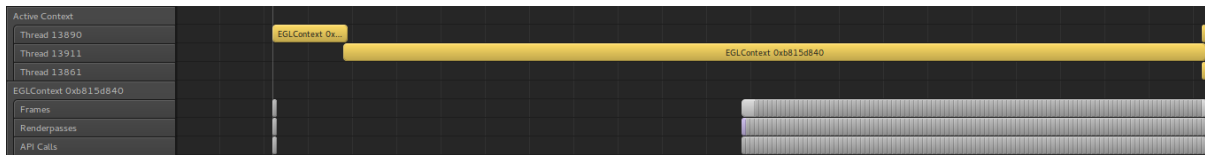
Heat Map / Core Map Annotations

The Heat Map and Core Map views now show active EGL contexts for each rendering thread. The length of each bar indicates the duration that that context was active (between `eglMakeCurrent` calls)



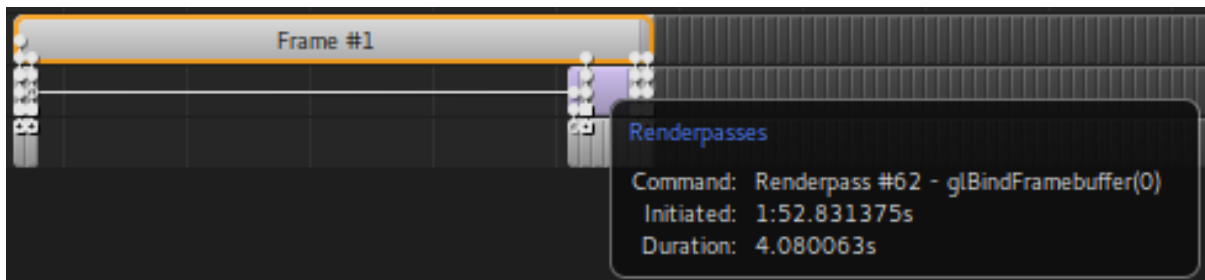
Mali Graphics Debugger Activity View

A new view is available from the same menu as the Heat Map / Core Map / Processes menu for each process that was traced using the MGD interceptor. This view shows the active contexts on each thread, each frame within a context, each render pass within a frame, and interesting API calls within a render pass.

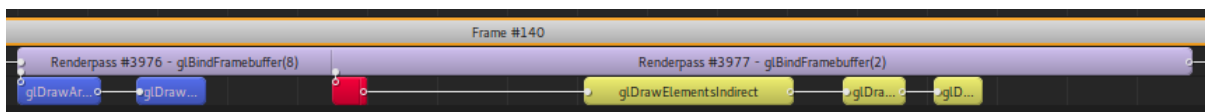


It is possible to select a frame, render pass or call item and see its relationship with other items. Selecting a frame will highlight all render passes within that frame and all calls associated with each render pass. Selecting a render pass will highlight the chain of render passes and calls for a given frame so far. Selecting a call will highlight 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. Additionally render passes will give an indication of the reason for the render pass, such as `eglSwapBuffers` for the end of frame, or `glBindFramebuffer(fboID)` indicating that the user changed the bound draw FBO.



For more detailed information the user may zoom in to a level where it is possible to see individual API calls. Calls are color coded to indicate the type of call they are.

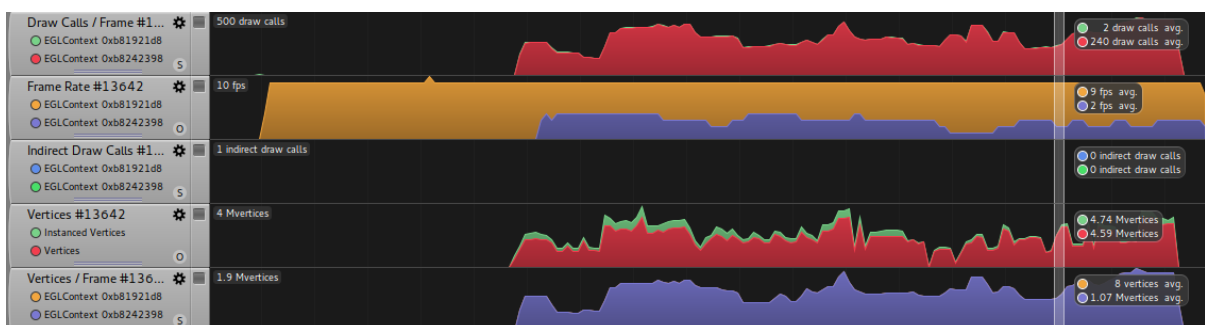


API Call Marker Colors

Red	Flushing calls such as <code>glFlush</code>
Green	End of frame calls such as <code>eglSwapBuffers</code>
Blue	Direct draw commands such as <code>glDrawArrays</code>
Yellow	Indirect draw commands such as <code>glDrawArraysIndirect</code>

Charts

Five charts are provided by the interceptor tracking draw calls, frame rate and vertices.



Draw Calls / Frame

This chart shows, for each EGL context, the number of draw calls per frame. Selecting a range with the caliper tool will give 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 as shown below using a simple rolling average over the last 6 frames. Selecting a range using the caliper tool will show an average value for that period.

$$r = \frac{1}{\bar{d}}$$

$$\bar{d} = \frac{\sum_{i=0}^n t_i - t_{i-1}}{n}$$

Calculation of frame rate r measured in frames per second. d is the average duration between frames (in seconds), t_i is the time of the i 'th last frame end (in seconds), and n is the size of the rolling average; which is 6.

Indirect Draw Calls

This chart shows, for each EGL context, the number of indirect draw calls. This information is provided as an indication of how much additional work may be being done by the GPU 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 will show 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 such 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 will be the same.

Vertices / Frame

This chart shows, for each EGL context, the number of vertices per frame. Selecting a range using the caliper tool will show an average value for that period. This chart is stacked so the total height indicates the total number of vertices at any given time.

Chapter

9

About the Implementation

Topics:

- [Khronos Conformance](#)

The Mali Graphics Debugger works by intercepting each OpenGL ES, EGL, OpenCL, and Vulkan calls made by the application under test. Once intercepted, call arguments are logged before they are passed transparently to the existing OpenGL ES, EGL, OpenCL, or Vulkan library. The logged data is collected and passed to the host via a daemon running on the system.

Khronos Conformance

This product is based on a published Khronos Specification. Under normal trace conditions no changes are made to the stream of calls made by the application to the existing OpenGL ES, OpenCL, EGL, or Vulkan implementation and therefore should not affect the conformance status of that implementation.

Chapter

10

Known Issues

Topics:

- [*OpenGL ES Extensions*](#)
- [*Shading Language Version*](#)
- [*Shader Compiler*](#)
- [*Performance*](#)
- [*API Asset Coverage*](#)
- [*Memory*](#)
- [*Partial Support for Earlier Trace Versions*](#)
- [*GTK Warnings when closing MGD \(Ubuntu only\)*](#)
- [*Issues With Texture and Image Output on 64-bit Linux Hosts*](#)
- [*Issues Viewing Khronos Reference Pages*](#)
- [*Intercepting Without Using LD_PRELOAD*](#)
- [*Multiple Drivers Installed on the System*](#)
- [*Daydream VR compositor skips right eye execution on capturing a frame*](#)
- [*Device Manager does not work correctly on Huawei Mate 9, Mate 10 and Mate 10 Pro*](#)

OpenGL ES Extensions

Generally, known OpenGL ES extensions will appear in the trace, but will not have an effect on the trace or asset state.

Shading Language Version

The host application supports syntax-highlighting of shaders from OpenGL ES 3.1 and before. There is support for Open CL kernel source but this is currently limited to a plain text view.

Shader Compiler

The system uses a built-in version of the Mali Offline Shader Compiler to determine OpenGL ES shader cycle counts. These cycle counts will only be correct for Mali-T760 devices running r7p0 drivers, however, other Mali-T600 based devices are likely to offer similar counts.

Performance

The Mali Graphics Debugger aims to be as unobtrusive as possible. However, there is a performance impact imposed by the gathering and sending of trace data to the host. This is particularly pronounced when you request a snapshot of a frame as this causes pixel read-back calls to be inserted by the interceptor after each draw call.

The interceptor library has negligible impact on application performance when not tracing.

API Asset Coverage

- There is currently limited support for most API extension enumerations
- Not all asset types are covered at present
- Not all texture types/formats are currently supported

Memory

Capturing for long periods of time or attempting to capture many framebuffer and/or other images is very expensive on memory. If a memory shortage is detected an active trace may be terminated to protect the system. See [Increasing Available Memory](#) on page 10 for advice.

Partial Support for Earlier Trace Versions

As new features are being added to the Mali Graphics Debugger, the data format of the saved traces is updated. We aim to support opening and visualizing traces captured with an earlier version of the tool but we cannot guarantee full functionality with older traces. The best solution is to update the Mali Graphics Debugger software on both target and host and re-trace the application.

GTK Warnings when closing MGD (Ubuntu only)

Occasionally on closing the application the system will show errors from the GIMP Toolkit (GTK) similar to the following:

```
LIBDBUSMENU-GTK-CRITICAL **: watch_submenu: assertion
`GTK_IS_MENU_SHELL(menu)' failed
(Mali Graphics Debugger:3989): LIBDBUSMENU-GLIB-WARNING **: Trying to
remove a child that
```



```
doesn't believe we're it's parent.
```

These are believed to relate to issues within GTK itself: they are widely reported and are believed to be harmless.

Issues With Texture and Image Output on 64-bit Linux Hosts

Mali Graphics Debugger requires that the 32-bit version of libjpeg62 be available on your system. On certain distributions, this is not there by default. On Ubuntu, this can be installed using the following command:

```
sudo apt-get install libjpeg62:i386
```

Issues Viewing Khronos Reference Pages

Depending on your particular platform, you may or may not be able to correctly view the Khronos Reference Pages when double clicking functions. This is particularly an issue for Windows which uses Internet Explorer. This is, unfortunately, outside of our control, as it is an incompatibility between the Khronos reference page format and the platform default browser.

Intercepting Without Using LD_PRELOAD

In some cases it may not be possible to use LD_PRELOAD (for example, LD_PRELOAD is already being used for another purpose). In this case you need to 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 case, the /path/to/intercept/dir/ should be the directory on the target where the installation files were copied. This must contain libinterceptor.so, and should include symlinks to libinterceptor.so named libEGL.so, libGLv2.so and libGLv1_CM.so.

It is possible to setup 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/
libGLv1_CM.so
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libGLv2.so
ln -s /path/to/intercept/libinterceptor.so /path/to/intercept/libOpenCL.so
```

The /path/to/original/drivers/dir/ should contain the pre-existing libGLv2.so, libEGL.so files from the graphics driver installation.

LD_PRELOAD need not be defined when using this method.

When a graphics application runs, the MGD 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.



Note: Sometimes you will find that the original Mali drivers pointed to by MGD_LIBRARY_PATH are tiny shim libraries that do not export any entry points, but instead depend on libmali.so; in this situation the interceptor will fail to correctly load the driver libraries unless MGD_LIBRARY_PATH also contains libmali.so. When this is not the case you may either point MGD_LIBRARY_PATH to the location of libmali.so, regardless of whether that location also contains libEGL/libGL libraries, or you may point MGD_LIBRARY_PATH to a location that contains symlinks to libmali.so instead.

Multiple Drivers Installed on the System

In some cases you may have more than one version of Mali driver installed on your device. An example would be if you aim to use both X11 and FBDEV on the same Linux platform. In these circumstances it may not be possible to use the standard [LD_PRELOAD approach](#) on its own.

You must instead use that approach as normal whilst defining the MGD_LIBRARY_PATH environment variable as follows:

```
MGD_LIBRARY_PATH=/path/to/original/drivers/dir/
```

The `/path/to/original/drivers/dir/` should contain the pre-existing libGLESv2.so, libEGL.so files from the graphics driver installation.

When a graphics application runs, the MGD 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 [Intercepting without using LD_PRELOAD](#) for more information.

Daydream VR compositor skips right eye execution on capturing a frame

When you capture a frame of a Daydream application the VR compositor may skip the right eye execution. This is because capturing the framebuffer content takes longer than the expected operation duration.

In this case, frames will only contain:

1. The draw calls made by the application to prepare the eye buffers.
2. The left eye draw calls to FB0 made by the VR compositor.

One way to make the VR compositor draw both eyes on capturing is to disable distortion and asynchronous reprojection. From the Daydream application, check **Settings > Developer options > Force Undistorted Rendering**

You can [enable "Developer options" by tapping Build Version 7 times](#) from the Daydream application settings.

Alternatively you can disable the asynchronous reprojection from your application, using [setAsyncReprojectionEnabled\(boolean\)](#)

Device Manager does not work correctly on Huawei Mate 9, Mate 10 and Mate 10 Pro

At the moment the [Device Manager](#) does not work correctly on Huawei Mate 9, Mate 10 and Mate 10 Pro devices, however you can perform a [manual installation](#).

Installing the interceptor requires a few extra steps for the Mate 9, Mate 10 and Mate 10 Pro. After following the instructions your device now has the files `/vendor/lib/egl/libGLES_mgd.so` and `/vendor/lib64/egl/libGLES_mgd.so`.

From an ADB shell, run the the following commands:

```
rm /vendor/lib/egl/libGLES.so /vendor/lib/egl/libGLES
cp /vendor/lib/egl/libGLES_mgd.so /vendor/lib/egl/libGLES.so
cp /vendor/lib/egl/libGLES_mgd.so /vendor/lib/egl/libGLES
rm /vendor/lib64/egl/libGLES.so /vendor/lib64/egl/libGLES
cp /vendor/lib64/egl/libGLES_mgd.so /vendor/lib64/egl/libGLES.so
cp /vendor/lib64/egl/libGLES_mgd.so /vendor/lib64/egl/libGLES
```

Chapter 11

Changes from Previous Versions

Topics:

- [*Changes between version 4.9.0 and 4.9.2*](#)
- [*Changes between version 4.8.1 and 4.9.0*](#)
- [*Changes between version 4.8.0 and 4.8.1*](#)
- [*Changes between version 4.7.0 and 4.8.0*](#)
- [*Changes between version 4.6.0 and 4.7.0*](#)
- [*Changes between version 4.5.0 and 4.6.0*](#)
- [*Changes between version 4.4.1 and 4.5.0*](#)
- [*Changes between version 4.4.0 and 4.4.1*](#)
- [*Changes between version 4.3.0 and 4.4.0*](#)
- [*Changes between version 4.2.0 and 4.3.0*](#)
- [*Changes between version 4.1.0 and 4.2.0*](#)
- [*Changes between version 4.0.0 and 4.1.0*](#)
- [*Changes between version 3.5.1 and 4.0.0*](#)
- [*Changes between version 3.5.0 and 3.5.1*](#)
- [*Changes between version 3.4.0 and 3.5.0*](#)
- [*Changes between version 3.3.0 and 3.4.0*](#)
- [*Changes between version 3.2.0 and 3.3.0*](#)
- [*Changes between version 3.1.0 and 3.2.0*](#)
- [*Changes between version 3.0.0 and 3.1.0*](#)
- [*Changes between version 2.1.0 and 3.0.0*](#)
- [*Changes between version 2.0.2 and 2.1.0*](#)

- *Changes between version 2.0.1 and 2.0.2*
- *Changes between version 2.0.0 and 2.0.1*
- *Changes between version 1.3.2 and 2.0.0*
- *Changes between version 1.3.0 and 1.3.2*
- *Changes between version 1.2.2 and 1.3.0*
- *Changes between version 1.2.1 and 1.2.2*
- *Changes between version 1.2.0 and 1.2.1*
- *Changes between version 1.1.0 and 1.2.0*
- *Changes between version 1.0.2 and 1.1.0*
- *Changes between version 1.0.1 and 1.0.2*
- *Changes between version 1.0.0 and 1.0.1*
- *Changes in version 1.0.0*

Changes between version 4.9.0 and 4.9.2

The following summarizes the changes made in this issue.

Issue	Summary
MGD-3049	Filter non-active shaders in the Shaders View
MGD-2997	Mark up <code>glDrawElementsInstancedBaseVertex</code> , <code>glDrawRangeElements</code> , and <code>glDrawRangeElementsBaseVertex</code> as draw calls
MGD-2983	Extract and Remarshall Trace to target for Full Trace Replay
MGD-3069	Fixed: Unrooted interceptor fails to traces functions calls having <code>R_ARM_GLOB_DAT</code> relocations
MGD-3068	Fixed: Unrooted interceptor fails to trace libraries not loaded using the full path
MGD-3051	Fixed: <code>NullPointerException</code> when you "find previous affecting function" when selected function call is in "No active context" renderpass
MGD-3039	Fixed: Interceptor does not work on Mate 10 Pro with UE4 applications
MGD-2844	Fixed: Device Manager doesn't work on Mate 9

Changes between version 4.8.1 and 4.9.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2255	Extend full trace replay to OpenGL ES 3.2
MGD-2907	Multi-context support for full trace replay
MGD-2975	Use off-screen surface for previous EGL window surfaces in full trace replay
MGD-2966	Map EGL configurations more intelligently in full trace replay
MGD-2960	Fixed: Exporting captured framebuffer reverses the image
MGD-2744	Fixed: FFTOcean not correctly replayed
MGD-3000	Fixed: Resuming the full trace replay Android app after pausing does not work

Changes between version 4.8.0 and 4.8.1

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2971	Fixed: MGD views fail to update correctly

Changes between version 4.7.0 and 4.8.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2905	Add Android O device support
MGD-2936	Add Android Project Treble device support
MGD-2913	Move from Eclipse Mars to Eclipse Neon
MGD-2912	Improve target device tracing performance

MGD-2920	Improve host trace processing performance
MGD-2608	Reduce host memory consumption
MGD-2609	Improve host UI speed and fluidity
MGD-2949	Delete per-trace temporary files on trace close
MGD-2950	Reduce total temporary disk space used
MGD-2951	Reduce total number of temporary files
MGD-2179	Improve performance of OpenGL ES Framebuffers view
MGD-2846	Add the ability to hide the attachments in the OpenGL ES Framebuffers and Vulkan Frame Capture view
MGD-2916	Show "Professional" in the MGD window name if a valid DS-5 license is detected
MGD-2922	Fixed: mgddaemon crashes on startup on some linux platforms
MGD-2802	Fixed: Timeline render pass labels are wrong
MGD-2780	Fixed: MGD renders empty windows on the first start on MacOS
MGD-2557	Fixed: Out of memory issue when there are large amounts of mapped memory changes in Vulkan apps

Changes between version 4.6.0 and 4.7.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2670	Detect devices and install MGD
MGD-2675	Support OVR_multiview
MGD-2792	Implement selective asset tracing
MGD-2757	Unified handling of mgddaemon, interceptor and MGD.app version mismatch
MGD-2758	User would like to easily find the version of interceptor, mgddaemon and MGD.app
MGD-2777	Improve error message in Android Device Manager when something goes wrong
MGD-2790	Implement and document ChromeOS support
MGD-2798	Document Unity support
MGD-2761	User would like to easily access the Connection Wizard command logs
MGD-2786	Add browse button for ADB and GPUVerify in Preferences
MGD-2787	Add the amount of required space to error message in Android Device Manager
MGD-1670	Add full support for KHR_Debug
MGD-2712	Support EGL_KHR_lock_surface3 extension
MGD-2804	Fixed: Automated trace commands are ignored
MGD-2749	Fixed: Ice Cave frame capture is incomplete
MGD-2745	Fixed: Capturing framebuffer at end of trace can't be cancelled
MGD-2769	Fixed: Interceptor looks for wrong libc version on Arndale-5250
MGD-2778	Fixed: Installing Vulkan layer on Nexus 10 fails
MGD-2781	Fixed: Android Device Manager failed to install root interceptor on rooted Galaxy S7

Changes between version 4.5.0 and 4.6.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2754	Update documentation explaining how to better capture frames in Daydream applications
MGD-2752	Update documentation for using MGD with Unreal Engine 4 applications
MGD-2751	Update documentation for using the root Vulkan layer on Android N
MGD-2707	Add support for Daydream, with correct frame end detection
MGD-2703	Support frame capture modes with VAOs/VBs
MGD-2433	Add Android Device Manager for one-click connection to devices and installation of target side components (e.g. MGD daemon app, interceptor, Vulkan layer)
MGD-2424	Support OVR_multiview_multisampled_render_to_texture extension
MGD-1724	Improve error messages when processlist.cfg is installed with incorrect permissions
MGD-232	Improve error messages when mgddaemon and interceptor library versions are different
MGD-2763	Fixed: mgddaemon fails to launch on certain soft-float platforms
MGD-2727	Fixed: Full Trace Replay doesn't work on Android 7
MGD-2725	Fixed: Missing MS Visual C++ 2010 redistributable
MGD-2656	Fixed: No dynamic help for renderbuffers view
MGD-2650	Fixed: Exception printed when launching second instance of MGD

Changes between version 4.4.1 and 4.5.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2667	Unrooted support for Android 7
MGD-2581	MGD.apk should gracefully handle the situation when mgddaemon is already running
MGD-2565	Support Non-Mali devices
MGD-2195	Update documentation to show how to use libMGD.so with Android Studio
MGD-2030	mgddaemon should exit gracefully when port 5002 is busy
MGD-1963	Improve error message when invalid trace version is detected
MGD-2663	Increase reliability of non-root interceptor
MGD-2683	Fixed: Extra incorrect render passes added when live tracing
MGD-2680	Fixed: Interceptor crashes when application indexes into deleted buffers
MGD-2661/ MGD-2681	Fixed: Interceptor crashes when used on non-Mali devices

Changes between version 4.4.0 and 4.4.1

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2677	Add Mali Offline Compiler that supports OpenGL ESSL 320

Changes between version 4.3.0 and 4.4.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1093	User Scripting: User would like to filter/highlight draw calls by criteria
MGD-471	User Scripting: Show call frequency per frame, average non-draw calls per draw call
MGD-190	User Scripting: User would like to see the draw calls that use a specific program to understand scope for optimization of that program
MGD-182	User Scripting: User would like to identify assets (textures, buffers, etc.) which have duplicate contents
MGD-2460	User Scripting: Scripts can read and write to the MGD model
MGD-2623	User Scripting: Scripts can interact with frames from the MGD model
MGD-2598	User Scripting: Create set of examples
MGD-2586	User Scripting: Scripts can get assets and their properties at a given function call
MGD-2585	User Scripting: Scripts can get the parameters and return value of a function call
MGD-2584	User Scripting: Scripts can get the OpenGL ES state at a given function
MGD-2494	Function call view should show return value
MGD-2493	Image/Texture View could allow smaller image column
MGD-2469	OpenGL ES framebuffers should be displayed in assets view
MGD-2561	Fixed: When using an active texture of more than 31 invalid constants are displayed
MGD-2533	Fixed: Depth/stencil buffer contents not sent on glClear* when depth/stencil test disabled
MGD-2500	Fixed: "ASTC data size not as expected for given dimensions and block size" has empty tooltip
MGD-2489	Fixed: No tooltip for "Copy of not tightly packed regions is not supported" problem on vkCmdCopyBufferToImage
MGD-2316	Fixed: GL_TRANSFORM_FEEDBACK_ACTIVE target state not updating and always saying GL_FALSE even when active
MGD-74	Fixed: Shader view sorts names alphabetically rather than numerically

Changes between version 4.2.0 and 4.3.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2542	Add documentation for using MGD with Unreal Engine
MGD-2435	Add support for tracing Vulkan on Linux
MGD-2364	Vulkan Capture should support multiple sub passes
MGD-2362	Vulkan Capture should support transient attachments
MGD-2361	Vulkan Capture should capture depth/stencil formats
MGD-2252	OpenGL ES 2.0 Full-Trace Replay on Android

MGD-2543	Fixed: Incorrect sending of VkPipelineViewportStateCreateInfo could cause a segfault
MGD-2541	Fixed: Sending VkWriteDescriptorSet crashes the interceptor
MGD-2534	Fixed: ASTC images using non-zero buffer row length are not displayed in ProtoStar
MGD-2525	Fixed: Inconsistent active/inactive status of the depth attachment
MGD-2522	Fixed: Image/texture opening/exporting is available even when image content not available
MGD-2510	Fixed: Saving an overridden shader source code with wrong syntax triggers an exception
MGD-2504	Fixed: OpenGL ES only commands are enabled even in Vulkan or OpenCL traces
MGD-2498	Fixed: MGD disconnects when tracing ProtoStar
MGD-2496	Fixed: When editing a bookmark in the bookmarks view losing focus doesn't save it
MGD-2366	Fixed: MGD does not work with conformant Vulkan driver
MGD-2537	Fixed: OpenGL ES shader map framebuffer capture fails with some ESSL versions
MGD-2532	Fixed: Closing a trace while generating a Render Pass Dependency report throws exceptions

Changes between version 4.1.0 and 4.2.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2495	Buffers View should allow filtering by handle/id
MGD-2414/ MGD-1686	Show state set by glTexParameter or glSamplerParameter
MGD-2407	Show Vulkan Command Buffers state
MGD-2345	Show Vulkan vertex buffers bindings
MGD-2386	Support all pointer data for OpenGL ES 2.0 and EGL
MGD-2262	Implement Vulkan Image View
MGD-2028	Better support for Open CL buffers
MGD-1679/ MGD-1069	List information on FBO texture and renderbuffer attachments
MGD-198	User would like to see texture units and their bound textures
MGD-2507	Fixed: JVM crashes when saving certain traces
MGD-2502	Fixed: Depth capture with texture array attachments is broken
MGD-2501	Fixed: Framebuffer previews broken for depth only framebuffers
MGD-2492	Fixed: Buffer View copy issues
MGD-2491	Fixed: Texture View copy issues
MGD-2487	Fixed: Message "Using GL_PIXEL_UNPACK_BUFFER" is wrongly displayed for textures loaded using glCompressedTexImage2D with no GL_PIXEL_UNPACK_BUFFER buffer bound
MGD-2482	Fixed: Trace view not cleared when changing trace
MGD-2459	Fixed: Deleted and detached shaders are missing/wrong
MGD-2458	Fixed: Breadcrumb bar missing on Windows

MGD-2127	Fixed: Automated trace doesn't work when the application is paused before starting
----------	--

Changes between version 4.0.0 and 4.1.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2326	Send Vulkan user data
MGD-2409	Improve MGD VR detection
MGD-2419	Support some floating point texture formats when using Frame Capture
MGD-2359	Fixed: Generate Render Pass Dependencies tab is incorrect if glClearBufferX is used
MGD-2358	Fixed: Target state for GL_DRAW_BUFFER doesn't work (this information is now in the Framebuffers View)
MGD-2356	Fixed: glClearBufferX is not treated as a draw during Frame Capture
MGD-2216	Fixed: Large/broken font issue on Mac OS X

Changes between version 3.5.1 and 4.0.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2320	Implement render pass capture for Vulkan
MGD-2248	Extend Vulkan render pass capture to draw call capture
MGD-2281	Track Vulkan vkPipeline, vkShaderModule, and vkPipelineLayout assets
MGD-2280	Track Vulkan vkDescriptorPool, vkDescriptorSet, and vkDescriptorSetLayout assets
MGD-2278	Track Vulkan vkImage and vkImageView assets
MGD-2277	Track Vulkan vkDeviceMemory, vkBuffer, and vkBufferView assets
MGD-2263	Implement Buffers View for Vulkan (with no data)
MGD-2224	Show Vulkan handle in a consistent way through all the views
MGD-2199	Go to function/frame dialog focuses text box on open
MGD-2377	Fixed: Buffers view filter doesn't filter by usage
MGD-2354	Fixed: Features such as filmstrip can no longer be enabled before application start up
MGD-2353	Fixed: Opening old traces loses information about frame capture modes
MGD-2346	Fixed: glMapBuffer(OES) 'access' parameter is incorrectly interpreted
MGD-2338	Fixed: Exceptions in traces with glTexImage3D and glTexSubImage3D using multiple layers textures
MGD-2299	Fixed: InvalidNavigationURIException thrown from the console window
MGD-2292	Fixed: MissingAssetException when connecting to a target with a running app
MGD-2275	Fixed: Double clicking columns in the Trace View causes NullPointerException
MGD-2241	Fixed: Replay frame progress bar doesn't work as expected
MGD-2170	Fixed: Capture/overdraw/etc. commands are not disabled for Vulkan and OpenCL
MGD-2167	Fixed: Framebuffers view can throw CancellationException, NullPointerException

MGD-2165	Fixed: Negative progress percentage when capturing frame
----------	--

Changes between version 3.5.0 and 3.5.1

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2229	The version of the MGD interceptor for unrooted devices now works on Android 6.0
MGD-898	Texture view now shows information about 3D textures
MGD-1898	Nodes no longer disappear from Trace Outline View
MGD-2131	glDrawBuffers now decodes the "bufs" parameter as a constant
MGD-2162	MGD now longer raises many dialog boxes in a out-of-disk-space error situation
MGD-2166	Newly captured frames are now visible in Trace Outline View when filtering is on
MGD-2213	glClearBufferX now decodes the "value" parameter correctly as an array
MGD-2214	glTransformFeedbackVaryings now decodes the "varyings" parameter correctly as strings
MGD-2226	The Geometry View now has mouse control for camera position
MGD-2230	Frame icons in the Trace Outline View now shows specific modes if only one mode was used

Changes between version 3.4.0 and 3.5.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2183	Filmstrip mode now works with Vulkan traces
MGD-2128	Added "Go to frame/function Call" option
MGD-2206	EGL attribute values are now shown in their text form (rather than their numeric values)
MGD-2019	Histograms support copy-to-clipboard as a CSV (context menu option)
MGD-2081	Support for tracking cl_mem assets (images and buffers) added
MGD-2200 + MGD-2171	Support for tracking Vulkan assets (render passes, command buffers, images, and image views) added
MGD-1977	Improved tracking of OpenGL ES assets and properties (up to OpenGL ES 3.2 now supported)
MGD-2120	Sorting fragment shaders by fragment count now works as expected
MGD-1915	Trace Analysis View no longer contains duplicate entries
MGD-2181	Fixed bug where the Trace Analysis View shows: "Infinity% of the draw calls are using GL_TRIANGLES"
MGD-2093	You can now start tracing OpenGL ES applications after they have started and their state will be recovered.
MGD-2192	Vulkan calls in the Trace Outline View now show time spent blocking
MGD-2219	Replay frame no longer fails when using force precision or modify shader overrides

Changes between version 3.3.0 and 3.4.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2075	Multisampled framebuffer attachments can now be captured
MGD-1937	The "Add Command" dialog in Automated Trace Control can no longer be hidden behind the main MGD window
MGD-2070	The Trace View and Trace Outline View selections are no longer reset when switching between traces
MGD-1932	The Render Pass Dependency View can now track renderbuffer dependencies
MGD-2062	MGD now color codes EGL contexts in GUI where appropriate.
MGD-1007	MGD now has more state filtering modes, see Target State View on page 56 for more information.
MGD-1923	MGD can now trace Vulkan api calls
MGD-2051	For VR applications, MGD now shows proper frame delimitation in the Trace Outline View
MGD-2133	Fixed MGD_LIBRARY_PATH sometimes not working correctly on a Firefly board with fbdev
MGD-2117	Fixed filmstrip images being 2-3 frames behind the actual frame
MGD-1809	The Console View has been simplified

Changes between version 3.2.0 and 3.3.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-2084	Add full cl_kernel and cl_program support
MGD-2072	Add support for OpenCL errors
MGD-2025	Add support for tracing OpenCL 1.2 functions
MGD-1859	Detect blocking calls in OpenCL
MGD-480	Add wireframe view on host to assess the complexity of a draw call
MGD-729	Add exporting geometry from draw calls into external graphics applications
MGD-1926	Show render pass dependencies (for textures only)
MGD-1877	Clicking on a framebuffer in framebuffers view should show color attachment 0
MGD-2067	Fix missing Streamline integration documentation

Changes between version 3.1.0 and 3.2.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1862	An internal error occurred during: "Closing 1> Capturing Target@/127.0.0.1:5002"
MGD-1885	Send streamline annotations from MGD interceptor
MGD-1801	Convert shaders view into one view

MGD-1867	Frame replayed with capture enabled is marked with an incorrect icon
MGD-1906	MGD overdraw map layer count tooltip is wrong
MGD-2002	Fragment count is incorrect
MGD-1757	Improved performance of Textures View
MGD-1917	Current note could be saved when user changes focus to a different API call
MGD-1483	User would like to end the search by pressing Escape
MGD-1452	MGD API trace search could allow an option to do a full text search
MGD-2007	Cannot use MGD 3.0 interceptor on arm-linux-sf target
MGD-226	Display indices for a selected glDrawElements call
MGD-1945	Add support for driver supplied memory reports
MGD-1726	Improve installation instructions for MGD Android App
MGD-1959	Can't trace app on unrooted Android device
MGD-2018	Overdraw histogram doesn't show 0x overdraw
MGD-1759	Improved performance of Vertex Attributes View
MGD-1758	Improved performance of Buffers View
MGD-1949	Track current binding for each buffer to allow to group buffers by binding type
MGD-1951	Allow user to select items in the Buffers View; show total memory consumption for selected items
MGD-1960	Update MGD.mk file to reflect current directory structure
MGD-2031	Histogram in Framebuffers View should refresh when switching between the trace files
MGD-2001	10x overdraw is reported for two different shades of gray
MGD-1950	Allow user to filter / sort Buffers View by binding type
MGD-2034	Using the "Ignore Alpha" option in the Framebuffers View sometimes sets the color data to black

Changes between version 3.0.0 and 3.1.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1901	Added Automated trace control
MGD-1742	Target state filtering now accepts regular expressions
MGD-1889	An update popup has been added
MGD-1887	There are now graphs of existing statistics
MGD-1871	Support for Android 6.0 (Marshmallow)
MGD-1770	Histograms for overdraw and shader map
MGD-1231	Currently bound EGL objects in Assets View are now shown in bold.
MGD-1800	Support for Tessellation and Geometry shaders
MGD-1900	Splash screen now shown on Windows

Changes between version 2.1.0 and 3.0.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-714	Show API calls for a single process from a system running multiple graphics processes
MGD-1225 + MGD-1788	Improved tracking of Shader assets when they are attached and linked to multiple programs
MGD-1697	Processes started after the pause command is sent should start in a paused state
MGD-1699	Replay commands should only be sent to the process from which they originated
MGD-1729	GUI now has the concept of multiple processes
MGD-1798	Fragment Count doesn't work on anything other than Framebuffer 0
MGD-181	User would like to see uniform data for OpenGL ES 3.0 entry points and types
MGD-150	Support multiple color attachments in the Framebuffers View (MRT)
MGD-1657	Improve handling of low disk/memory
MGD-1847	X86 Android support in Interceptor
MGD-167	Display a context-aware trace
MGD-1074	Add ability in Assets View to see what is bound to each framebuffer
MGD-1156	Windows installer could offer an option to launch MGD after successful installation
MGD-1268	MGD Assets View needs an export option
MGD-1562	Release User Guide as PDF
MGD-1571	Support Uniform Storage Blocks
MGD-1572	Support Vertex Array Objects
MGD-1582	Frame Capture doesn't work on framebuffers that are not RGBA
MGD-1655	MGD reports ASTC as unrecognized texture format
MGD-1656	MGD reports KHR_DEBUG as an unrecognized state
MGD-1666	MGD interceptor prints too many message if unsupported extensions are used
MGD-1677	Bring back the Framebuffers View
MGD-1680	glFramebufferRenderbuffer doesn't understand attachment code 0x821a
MGD-1682	Give user the option to not send buffers during glDispatchCompute
MGD-1687	MGD Assets View should provide an export shaders option
MGD-1714	Problem with texture files when using custom temporary storage directory
MGD-1725	Wrong kernel given to GPUVerify
MGD-1733	Create breadcrumb bar
MGD-1734	Separate different target processes into different models
MGD-1741	GPUVerify - String index out of range: -1
MGD-1742	PATH corrupted during installation
MGD-1749 + MGD-1804 + MGD-1805	Create a Timeline View
MGD-1356	Esc key could close Find API search

MGD-1362	No tooltip for Binary Format in shader views
MGD-1712	MGD should check if it can write to the temporary directory
MGD-1786	Add tooltip to explain deleted shaders

Changes between version 2.0.2 and 2.1.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1436	<i>Added state comparison support</i>
MGD-1448	<i>Added GPUVerify support</i>
MGD-1546	Upgraded base Eclipse version to Eclipse 4.4
MGD-1559	Remove fabricated shaders and textures from selection in Frame Overrides View
MGD-1558	Double clicking on item in Frame Overrides View should take you to that item
MGD-1561	User should be able to delete a frame override from the Frame Overrides View by pressing the delete key
MGD-1554	"Dirty" shader files are marked with a '*' in the Frame Overrides View
MGD-1553	Remove the compile button in Shader Override Editor (shaders are now compiled on save)
MGD-1551	You can now create frame overrides by right clicking on the relevant asset
MGD-1419	Convert interceptor to C++
MGD-1155	EGL parameters are now shown in the Assets View
MGD-1518	Support for 64-bit Android
MGD-1629	Fix a crash on Ubuntu when the user tries to open the Help
MGD-1570	Fix an issue where MGD could not uncompress ASTC images
MGD-1451	Add support for glBlitFramebuffer

Changes between version 2.0.1 and 2.0.2

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1580	MGD works correctly when it's not called libGLES_mgd.so
MGD-1581	MGD only looks for libEGL.so in /vendor/lib/egl/

Changes between version 2.0.0 and 2.0.1

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1565	MGD works correctly on Windows systems with non-English languages.
MGD-1539	Minor issues with images in the documentation.

Changes between version 1.3.2 and 2.0.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1146	Filmstrip feature has been re-enabled
MGD-1053	Help is now provided as an Eclipse plugin
MGD-1284	Added "Framebuffer" as a mid-level hierarchy in the Outline View
MGD-1210	Mgddaemon now provided as an Android application for rooted devices
MGD-1299	Hard float version of interceptor is now built without neon for greater compatibility
MGD-1222	Search highlighting improved
MGD-1131	Context menus in MGD now contain default actions
MGD-1137	Show and hide menus added for column context menus
MGD-71	Improved error message regarding the libjpeg62 dependency
MGD-240	Added more default items to the context menus
MGD-1011	Added alpha channel support when opening images opened from the Framebuffers View
MGD-835	User can now copy function calls from the Trace View to the clipboard
MGD-1352	Made improvements to previous/next change location in Target State View
MGD-879	New samples have been added to show features such as overdraw map, fragment count statistics, shader map and frame capture
MGD-1234	Debug menu now contains all of options from the toolbar and in the same order
MGD-1341	Columns in the Statistic View are now resizable
MGD-1246	User can now filter frames by MGD features such as overdraw
MGD-1432	MGD updated to use Java 7 and JRE is bundled with package
MGD-1086	Improved handling of large trace files
MGD-1322	Framebuffers View now correctly reports pixel coordinates no matter what the view size
MGD-1333	Progress bar now shown when doing a capture from paused
MGD-1429	MGD no longer hangs when replaying a frame that had previously been captured
MGD-1296	OpenGL ES 3.1 basic support has been added. All calls are now intercepted and displayed in the Trace View
MGD-1317	Failed calls to glTexImage* no longer update texture assets
MGD-219	Vertex attribute support added for OpenGL ES 3.0 entry points
MGD-1319	Increased the number of texture formats that MGD supports
MGD-1253	Attachment data is now captured for glDiscardFramebufferEXT and glInvalidateFramebuffer
MGD-245	Added the ability to create notes in the Trace View to highlight specific lines in a trace
MGD-290	Filmstrip feature can be toggled on or off during tracing
MGD-1460	User can now override a texture with a predefined checker board texture whilst replaying a frame
MGD-1466	User can now override the precision mode for all types in a shader whilst replaying a frame

MGD-1433	Added documentation for using the interceptor inside an APK
MGD-974	Icons added to help distinguish between regular frames, replayed frames and frames with different features
MGD-1297	Added new Compute Shaders View
MGD-1457	Added app selection screen to the new MGD daemon app
MGD-1088	Improved speed whilst calculating unique indices
MGD-1142	Ability to copy text strings in MGD added
MGD-1338	Total cycles number is now uses decimal notation instead of E notation
MGD-1170	Fixed tooltip for the index number of an API call
MGD-1339	Vertex and Fragment Shaders View counters are now aligned to the right
MGD-1230	Preferences menu has been re-enabled for all platforms
MGD-1202	Inconsistent use of N/A in the UI has been resolved
MGD-1287	Trace search now updates as the application is running
MGD-260	Context menu items are grayed out if unavailable
MGD-1395	Can now do a frame capture on a replayed frame
MGD-1282	Framebuffers now open in an image editor the same orientation as MGD displays
MGD-1458	MGD daemon application can now detect whether it is on a rooted device or not
MGD-1505	Version 4.5.0 of the Mali offline compiler integrated. This allows the compilation of compute shaders
MGD-1383	Items in the GUI are now correctly sorted
MGD-1470	User can now edit shaders in an application and then replay a frame using their edited shaders instead of the original ones
MGD-1179	EGL attribute names are now shown rather than numerical values
MGD-1134	Improved OpenGL ES 1.1 support
MGD-1445	Compiler manager no longer causes MGD to crash when opening certain traces

Changes between version 1.3.0 and 1.3.2

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1475	Support for Android 5.0

Changes between version 1.2.2 and 1.3.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-504	The results of glGetError are now show for every call
MGD-246	Added "Step" feature to allow stepping through frames on the target
MGD-896	Traces are now in a binary format to increase performance
MGD-282	Added a feature in the Vertex Attribute View to visualize sparse vertex indices. Also added a warning if the vertex indices are too sparse.
MGD-1024	Added support for replaying frames which don't create or destroy state

MGD-297	Shader cycles statistics are now calculated for ' <i>Midgard</i> ' based devices
MGD-263	Double-clicking an MGD trace file now opens MGD (Windows only)
MGD-900	Search function is faster
MGD-1000	MGD is more responsive when selecting the last frame in a big trace
MGD-1001	MGD no longer runs out of memory after opening and closing several trace files
MGD-1092	Fixed shader views sorting when using "Show create location"
MGD-241	String arguments are now shown instead of the pointer value in the Trace View
MGD-1121	The Vertex Attribute View is now correct when the minimum vertex index is >0
MGD-1138	Fixed Shaders View so it shows shaders correctly before they've been linked
MGD-1155	Added EGL parameters are in the Assets View
MGD-146	Stopped context being lost in the Outline View when selecting a call other than glDrawElements
MGD-1104	Fixed shadermaps so they use the same shader number as API calls
MGD-1096	In a multi-context trace, the trace problems are now correct
MGD-1094	Constants are now correctly displayed for glClear
MGD-65	Texture formats for T-Rex HD are now correct
MGD-1066	Stopped UI locking up while saving traces
MGD-831	Attaching a texture to a channel of an FBO now detaches whatever is currently bound
MGD-1091	Added support for BGRA textures
MGD-1079	Framebuffers View is now cleared when trace file gets closed
MGD-1065	Stopped UI locking up when connecting to a non-existing IP address
MGD-1015	"Show create location" / Show last modification now works for the shader views
MGD-997	It is no longer possible to close main application window when Open File dialog is open
MGD-1071	Assets View selection is no longer cleared when stepping backwards through the trace
MGD-1068	Stepping through draw calls no longer resets the asset table
MGD-1036	Fixed a NullPointerException which was thrown after reopening the Assets View
MGD-903	Stopped the Framebuffers View of other open traces being cleared when you close a trace
MGD-2	Canceling file read no longer results in Stream Closed error
MGD-888	Traces can be exported to a text file
MGD-210	Trace file highlights can now be removed once set
MGD-947	Function arguments now display correct enumeration values
MGD-935	Fixed java.util.ConcurrentModificationException thrown while tracing GFXBench

Changes between version 1.2.1 and 1.2.2

The following summarizes the changes made in this issue.

Issue	Summary
MGD-1012	Fixed unique indices count.

MGD-231	Support for Android KitKat 4.4.
---------	---------------------------------

Changes between version 1.2.0 and 1.2.1

The following summarizes the changes made in this issue.

Issue	Summary
MGD-935	Fixed java.util.ConcurrentModificationException.
MGD-197	Show framebuffer pixel information.
MGD-922	Eliminate memory-free error on interceptor shutdown.
MGD-921	Fix for interceptor segfault on GFXBench3 (PBO).
MGD-907	Handle Unicode characters in shader sources.

Changes between version 1.1.0 and 1.2.0

The following summarizes the changes made in this issue.

Issue	Summary
MGD-905	Allow binary shaders to be debugged. All apps will fall back to compiled shaders.
MGD-852	Added end to end tests for fragment count
MGD-837	Handle click on empty framebuffer image. Previously the app would throw a generic warning if an empty framebuffer was double-clicked.
MGD-887	Suppress multiple error dialogs. These were produced when trying to read a texture without the appropriate decoder. Only one report is now produced for each failing decoder.
MGD-463	Shaders View shows a fragment count for each shader All frames now start with eglSwapBuffers. Previously this call was included at the end of a frame. This change improves application consistency. Add an 'empty message' to the Framebuffers View.
MGA-860	Fixed exception thrown when calculating shader statistics.
MGA-876	Add armhf_fbdev in the installer
MGA-873	Fix workbench save/restore with new versions.
MGD-3	Fixed Vertex Shaders View and Fragment Shaders View being empty after shader deletion. Shaders are shown next to their bound program in the shader views. There may be duplicate shaders, as a shader can be bound to multiple programs. After a shader is deleted, the shader is shown grayed out if still attached Vertex shaders in the Vertex Shaders View also now have a color. Add framebuffer size in the tooltip
MGD-335	README_linux and README_android merged into user guide Added target installation content to the user guide. Deleted README_android and README_linux
MGD-844	Daemon now reports version during start up
MGD-852	Updated user documentation. Also improved the messages that a daemon produces when it starts up. Includes updates to user guide and code to better document the use of port 5002.
MGD-859	Adding Android ASTC support All mga references become mgd

	<p>Changed test trace file extension to mgd.</p> <p>Fix frame and draw call index numbers in Statistics View. These were one-out.</p> <p>Adding ASTC decoding to GUI.</p>
MGD-854	Changed Linux installer extension to tgz. This simplifies publishing.
MGD-160	Renaming State View 'State' column to 'State Item'
MGD-836	<p>Renaming samples to have a .mgd extension</p> <p>Show colors in the Fragment Shaders View.</p>
MGD-838	<p>Close process streams after use. These were no being closed, and so were accumulating until the system ran out of descriptors.</p> <p>Rename Trace Problems View</p>
MGA-753	Clear frame image when model unloads.
MGD-12	Show multiple texture formats/types only if they are different.
MGD-34	Overdraw map now supports OpenGL ES 3.0 shaders (Timbuktu works)
MGD-63	Add status line with progress report for background jobs.
MGA-595	Fix 'task already scheduled or canceled' when closing and re-opening the Statistics View.
MGA-801	Fix exception when canceling a file open action.
MGA-788	Inform the user when the connection to the target is broken
MGA-793	change extension to .mgd
MGA-722	<p>Better message for broken URLs in help lookup.</p> <p>Fix opening a file shows up as dirty.</p>
MGA-815	<p>Add title to vertex attribute index column.</p> <p>Hide the filmstrip panel if no screenshots are available.</p> <p>Display screenshots in the left pane in the GUI.</p>
MGA-760	<p>Saving now updates tab. And you can differentiate between save and save as.</p> <p>Adding single-step function.</p> <p>Fixing interaction between pause and framebuffer requests.</p> <p>Improve image drawing, remove unnecessary images.</p> <p>Add support for RGB and RGB565 formats for framebuffers.</p> <p>Enable O3 optimization in the Android build.</p>
MGA-794	<p>Message when texture cannot be displayed externally.</p> <p>Implemented overdraw map</p>
MGA-834	Fix bug calculating shader cycle average.
MGA-779	Vertices per draw call and frame now show Unique vertices as well
MGA-834	Add vertices count to shader stats.
MGA-284	<p>Selecting frame selects last call in frame. Previously the first call would be selected.</p> <p>Will allow us to more easily present per-frame information.</p> <p>Enable all the basic texture formats.</p> <p>Add GL_ALPHA texture format as grayscale image.</p> <p>Revised error handling. All LOGE or LOGI statements now terminate with \n (previously there was a mix). All exit() calls are now made with a non-zero value if indicating failure.</p>

	Implemented core OpenGL ES 3.0 vertex attribute types
MGA-727	<p>Updated User Guide. Updated known-issues section to contain information on the Khronos manual page issues.</p> <p>Added mgd.ini option to substitute vendor</p> <p>Renaming start/stop to connect/disconnect. This better reflects what these commands actually do, and helps towards separating the start/stop controls.</p> <p>Added LD_LIBRARY_PATH/MGD_LIBRARY_PATH interception method By naming/symlinking MGD appropriately, MGD can be placed in the LD_LIBRARY_PATH variable, and the library path to forward calls to can then be specified by passing the environment variable MGD_LIBRARY_PATH. This method is safer than LD_PRELOAD, and can be used with other tracers.</p> <p>Improved interceptor to support more drivers</p> <p>Included updated OpenGL ES 1.1 extension header.</p> <p>Updated size of bit fields in CL definition.</p> <p>Add to the trace a timestamp showing the time before/after each function call.</p> <p>Stop up-scaling the FB images if they are already smaller than the canvas.</p>
MGA-762	<p>Fixed find on a live trace</p> <p>Added armhf build to bundle</p> <p>Updated emulator to load libGLES from same directory. Now loads libGLES* from same directory as libEGL, rather than using LD_LIBRARY_PATH, which was causing some trouble previously.</p>
MGA-758	Fix null pointer exception when saving. This occurs when a constant value is unrecognized. These are now replaced with 'unrecognized' but this can only be temporary until unknown constants are properly supported.
MGA-724	Add User Guide note on GTK warnings.
MGA-720	Improving the progress bar on multiple file load. The previous bar was not correctly showing the total progress when opening multiple files, this has been fixed. There remains an issue with the blocking of the UI thread by the open editor call, but this is unavoidable at the moment.
MGA-717	Only permit existing file to be opened. Previously the File > Open dialog would allow the user to type a file name that didn't exist, causing problems later in the flow.
MGA-691	<p>Search substrings (they don't need to fully match).</p> <p>Fix Y coordinate of cube maps.</p>
MGA-684	Fix bug. MGD did not show up on Mac with Java 1.6
MGD-686	Fixing standard Mac menu items. The 'About MGD' and 'Preferences' options were missing from the Mac implementation.
MGA-697	<p>Revising socket close mechanism to eliminate errors. The socket shutdown code reported lots of warnings even in the correct case. Code has now been refactored to improve the messaging and to try and eliminate spurious warnings.</p> <p>Added EGL plugin to send context on eglMakeCurrent</p> <p>OpenGL ES 3.0 working on Android, and Nexus10 4.3 booting. Only needed to chmod to work on Android 4.3, so have changed the README to reflect this requirement. Have also removed the OpenGL ES 2.0 version of the android interceptor, and will now ship OpenGL ES 3.0</p>
MGD-698	Fix sequencing on save. Save operation no longer closes an active trace if the user cancels the file selection dialog.

Changes between version 1.0.2 and 1.1.0

The following summarizes the changes made in this issue.

Issue	Summary
MGA-353	Added Vertex Attributes View, which shows all the vertices used in a 'draw' call
MGA-353	Indices buffers are captured in the trace
	Added warning for swapping between buffer targets
MGA-653	Adding statistic for vertices per frame and show vertices count in the Outline View
	Shader statistics are now updated individually
MGA-349	Added ability to find function calls with text pattern search (Ctrl-F)
	Shaders are now sorted numerically
	Improved the vertical bar with markers to point to the correct items when it is clicked
MGA-591	"Save" now asks before overwriting
	Add reference to the buffers table in GL_ELEMENT_ARRAY_BUFFER_BINDING
MGA-620	Numbers start from 1 in views.

Changes between version 1.0.1 and 1.0.2

The following summarizes the changes made in this issue.

Issue	Summary
	Add uniforms support.
	Fix concurrent modification exception.
MGA-550	Fix bug in Windows installer.
	Moving views; adding right click menu.
	Dimension sorts by width following area.
	Adding shader attributes to the Programs View.
	Update EULA with Apache Commons library.
MGA-543	Fixing trace load progress counter.
MGA-539	Texture names now sort on ID
MGA-353	Added glVertexAttrib*f family of entry points
MGA-353	Added glVertexAttrib*f commands to the Asset Processor
	Upgrade everything to Java 1.7
	Small improvements to the UI of the Outline View
	Fix for interceptor vertex array buffer sending
MGA-514	Adding index to draw call in frame.
MGA-485	Fix warning 'timer already canceled'.
	Add detection of eglGetError errors.
	Display frames with alternate colors in the Trace View.
MGA-518	Adding table header tool tips.
MGA-517	Added index column to Trace View.

MGA-515	Attachment temporary files now deleted on exit. Small UI improvements Fix NullPointerException with buffers.
MGA-489	Fix Trace View switching. Add statistics entry to show the number of API calls per frame. Make open file trace faster.
MGA-483	Improved performance when highlighting the locations of a problem in the trace.
MGA-508	Fix double-click global action issue. Add Buffers View. Changed Assets View to sort OpenGL ES objects alphabetically
MGA-498	Added preview text and icons for Buffer assets. Add a simple binary viewer for buffers. Fix precision errors in the progress bar.
MGA-482	Show MB read in the progress bar to show activity for enormous files. Set the size of a texture even if no data is available. Display information for textures bound to framebuffers.
MGA-341	Show number of vertices and draw calls in the Outline View.
MGA-341	Add per frame and per draw call statistics in the view.
MGA-498	Added Asset Processor support for all OpenGL ES 3.0 target buffers.
MGA-498	Updated Asset Processor to handle glBufferSubData
MGA-498	Added tracking of buffer creation/deletion/modification to model.
MGA-498	Added glBufferData test app Improve file moving to work cross device

Changes between version 1.0.0 and 1.0.1

The following table details the changes made in this issue.

Issue	Summary	Notes
MGA-429	Removed multi-selection on Trace Problems View.	Since only the first selection is used. Also selection now done on a single-click rather than a double-click.
MGA-473	Fixed issue when textures with type 565 were sent.	Caused traces with RGB565 images to fail to load.
MGA-427	Personalized icons on shader editor.	Shader editor pane now shows the icon appropriate to the content.
MGA-472	Added eglGetProcAddress path for function pointer lookup in interceptor functions.	Previously, the only path available used dlsym, which will not work with extension functions which must look up the function in the underlying driver with eglGetProcAddress. The interceptor originally returned addresses to core API functions, which is counter to the wording of the EGL spec.

MGA-453	Fixing trace load cancel button.	MGD now stops cleanly when the cancel button is pressed on a large trace load.
MGA-451	Reducing argument memory usage.	Provide support for saving attachments and framebuffers to disk when the system is short of memory.
MGA-474	Fix Save dialog (before it was actually a open dialog). Add type to the image buffers and textures, and display it.	Also add a filter for txt extension on trace files.
MGA-418	Added documentation for all special case functions.	All trace functions can be double-clicked to reveal help page.
MGA-447	Modified Trace View code to handle bit fields.	Functions such as glClear which take a bit mask now show the bit mask value expanded into each of the mask values if possible.
MGA-469	Fix exception thrown when no markers are selected.	
MGA-398	Added process filter to Android target.	Now requires a /system/lib/egl/processlist.cfg to exist, and have the running process name equal one of the lines of that file, in order for a single process trace to occur. The interceptor will trace all processes by default if processlist.cfg is not present.
MGA-435	Removed shader editor right click menu	The menu items had no effect, and so have been removed.
MGA-436	Save and restore window layout	The application will now remember where windows were placed in the workbench, and will restore these at the start of the next session.
MGA-448	Retain connection IP address and port information	If the user enters information in the connection dialog then this is now saved across sessions.
MGA-431	Removing unused popup menu on Statistics View.	Removed unnecessary popup menu.

Changes in version 1.0.0

This version is the first release of the Mali Graphics Debugger.

Chapter 12

Support

For help and support from Arm and fellow developers, please visit the [Arm Mali Graphics Community](#).

It should be noted that continuing support of the product is at Arm's discretion unless explicitly included in a current contract between Arm and you.

Chapter

13

Legal

Topics:

- *Proprietary Notice*
- *Analytics Information We Track*

Proprietary Notice

Words and logos marked with ® or TM are registered trademarks or trademarks of Arm Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

OpenGL is a registered trademark and the OpenGL ES logo is a trademark of Silicon Graphics Inc. used by permission by Khronos.

EGL and the EGL logo are trademarks of the Khronos Group Inc.

Vulkan and the Vulkan logo are trademarks of the Khronos Group Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Android is a trademark of Google Inc.

Mac and OS X are trademarks of Apple Inc., registered in the U.S. and other countries.

Unreal is a trademark of Epic, registered in the U.S. and other countries.

Unity is a trademark of Unity Technologies.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Arm Limited in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Arm Limited shall not be liable for any loss for damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Analytics Information We Track

The following summarizes the analytics information we track.

When Tracked	Name	Summary
Per product version	product/build	Tracks the current product build ID
Per product version	product/version	Tracks the current product version
Per session	product/user_preference	Tracks which MGD user preferences are enabled and disabled
Per session	system_property/os_arch	Tracks the operating system bitness
Per session	system_property/os_distribution_description	(Linux only) Tracks the Linux distribution version descriptive name
Per session	system_property/os_distribution_name	(Linux only) Tracks the Linux distribution name
Per session	system_property/os_distribution_version	(Linux only) Tracks the Linux distribution version number
Per session	system_property/os_name	Tracks the operating system name

Per session	system_property/os_version	Tracks the operating system version
Per session	system_property/user_country	Tracks the users locale country
Per session	system_property/user_language	Tracks the users locale language
Per session	action/connect_to_android	Tracks the number of times an Android device was successfully connected to using the device manager
Per session	action/connect_to_ip	Tracks the number of times an IP address was successfully connected to using the device manager
Per session	action/connect_to_last_device	Tracks the number of times the last device was successfully connected to using the connect button
Per session	action/connect_to_linux	Tracks the number of times a Linux device was successfully connected to using the device manager
Per session	action/user_clicked_updated_version_block_link	Tracks if the user clicked the block link in the update available popup
Per session	action/user_clicked_updated_version_download_link	Tracks if the user clicked the download link in the update available popup
Per session	memory/amount_available	Tracks the amount of memory available to Java when the program is first run
Per session	memory/critical_event	Tracks the number of times a critical memory level is detected
Per session	memory/warning_event	Tracks the number of times a warning memory level is detected
Per session	perspective/perspective_activated	Tracks the number of times a perspective is used
Per session	product/license_checked_out	Tracks when MGD successfully checks out a license
Per session	product/user_preference	Tracks which MGD user preferences are enabled and disabled
Per session	view/tab_activated	Tracks the number of times an individual view is used
Per session	window/pop_up_window_created	Tracks the number of times a new pop-up window is created
Per trace	action/add_override_count	Tracks the number of times each kind of frame override is used
Per trace	action/enable_capture_all_framebuffer_attachments	Tracks the number of times additional framebuffer attachment capturing was enabled
Per trace	action/enable_filmstrip_count	Tracks the number of times the filmstrip mode feature was used
Per trace	action/enable_fragment_count	Tracks the number of times the fragment count feature was used
Per trace	action/enable_frame_capture	Tracks the number of times the frame capture feature was used
Per trace	action/enable_overdraw	Tracks the number of times the overdraw feature was used

Per trace	action/enable_shadermap	Tracks the number of times the shadermap feature was used
Per trace	action/open_documentation	Tracks number of times user uses 'Open Documentation' feature
Per trace	action/replay_frame	Tracks the number of times the frame replay feature was used
Per trace	action/replay_frame_with_capture	Tracks the number of times the frame replay with capture feature was used
Per trace	action/use_gpu_verify	Tracks when GPUVerify feature is used
Per trace	action/use_state_diff_feature	Tracks the number of times the state diff is used
Per trace	trace/gles_api_version	Tracks the numbers of each EGL context created, grouped by OpenGL ES version
Per trace	trace/gles_buffer_target_use_count	Tracks the numbers of uses of each OpenGL ES buffer target (GL_ARRAY_BUFFER, etc.)
Per trace	trace/gles_draw_mode_use_count	Tracks the numbers of uses of each OpenGL ES draw mode (GL_TRIANGLES, etc.)
Per trace	trace/gles_fbo_attachment_point_use_count	Tracks the numbers of uses of each OpenGL ES FBO attachment point (GL_COLOR_ATTACHMENT0, etc.)
Per trace	trace/gles_fbo_attachment_type_use_count	Tracks the numbers of uses of each OpenGL ES FBO attachment target (GL_RENDERBUFFER, GL_TEXTURE2D, etc.)
Per trace	trace/gles_fbo_dimensions_largest_area	Tracks the largest overall area of an FBO encountered
Per trace	trace/gles_fbo_dimensions_largest_height	Tracks the largest height of an FBO encountered
Per trace	trace/gles_fbo_dimensions_largest_width	Tracks the largest width of an FBO encountered
Per trace	trace/gles_texture_dimension_largest	Tracks the largest overall area of any texture encountered
Per trace	trace/gles_texture_format_use_count	Tracks the numbers of uses of each OpenGL ES texture format used (GL_RGBA8, etc.)
Per trace	trace/gles_texture_target_use_count	Tracks the numbers of uses of each OpenGL ES texture target used (GL_TEXTURE2D, etc.)
Per trace	trace/largest_call_count	Tracks the total number of all function calls
Per trace	trace/largest_drawcallperframe_count	Tracks the largest number of draw function calls in a single frame
Per trace	trace/largest_frame_count	Tracks the total number of frames
Per trace	trace/largest_instancedverticesperframe_count	Tracks the largest number of instanced vertices per frame
Per trace	trace/largest_shader_cycles	Tracks the largest total shader cycles for any shader
Per trace	trace/largest_verticesperframe_count	Tracks the largest number of vertices per frame
Per trace	trace/live_process_config	Tracks the number of function calls each process configuration was used for in a newly created trace.

Per trace	trace/live_renderer_id	Tracks the GPU renderer identifier (a hardware configuration) for a newly created trace.
Per trace	trace/live_target_type	Tracks the type of live target connected to (Android, Linux)
Per trace	trace/live_vendor_id	Tracks the GPU vendor identifier for a newly created trace.
Per trace	trace/live_version	Tracks the version number of a trace
Per trace	trace/number_of_generated	Tracks the numbers of each type of OpenGL ES resource created (Textures, Shaders, Sync Objects, etc.)
Per trace	trace/opengl_context_counter	Tracks the number of OpenGL contexts created
Per trace	trace/read_renderer_id	Tracks the GPU renderer identifier (a hardware configuration) for an opened trace.
Per trace	trace/read_vendor_id	Tracks the GPU identifier for an opened trace.
Per trace	trace/read_version	Tracks the version of the trace file loaded from a saved trace
Per trace	trace/vulkan_api_version	Tracks the numbers of each Vulkan instance created, grouped by Vulkan API version
Per user triggered ADB task (e.g. interceptor/daemon install, connect, etc.)	user_adb_task/description	Tracks the description of a user triggered ADB task
Per user triggered ADB task (e.g. interceptor/daemon install, connect, etc.)	user_adb_task/device_manufacturer	Tracks the manufacturer of the Android device that a user triggered ADB task ran on
Per user triggered ADB task (e.g. interceptor/daemon install, connect, etc.)	user_adb_task/device_model	Tracks the model of the Android device that a user triggered ADB task ran on
Per user triggered ADB task (e.g. interceptor/daemon install, connect, etc.)	user_adb_task/device_sdk_version	Tracks the SDK version of the Android device that a user triggered ADB task ran on
Per user triggered ADB task (e.g. interceptor/daemon install, connect, etc.)	user_adb_task/error_message	Tracks the error message (if there was one) of a user triggered ADB task
Per user triggered ADB task (e.g. interceptor/	user_adb_task/succeeded	Tracks whether a user triggered ADB task succeeded or not

daemon install,
connect, etc.)