



Arm[®] Frame Advisor

Version 1.8

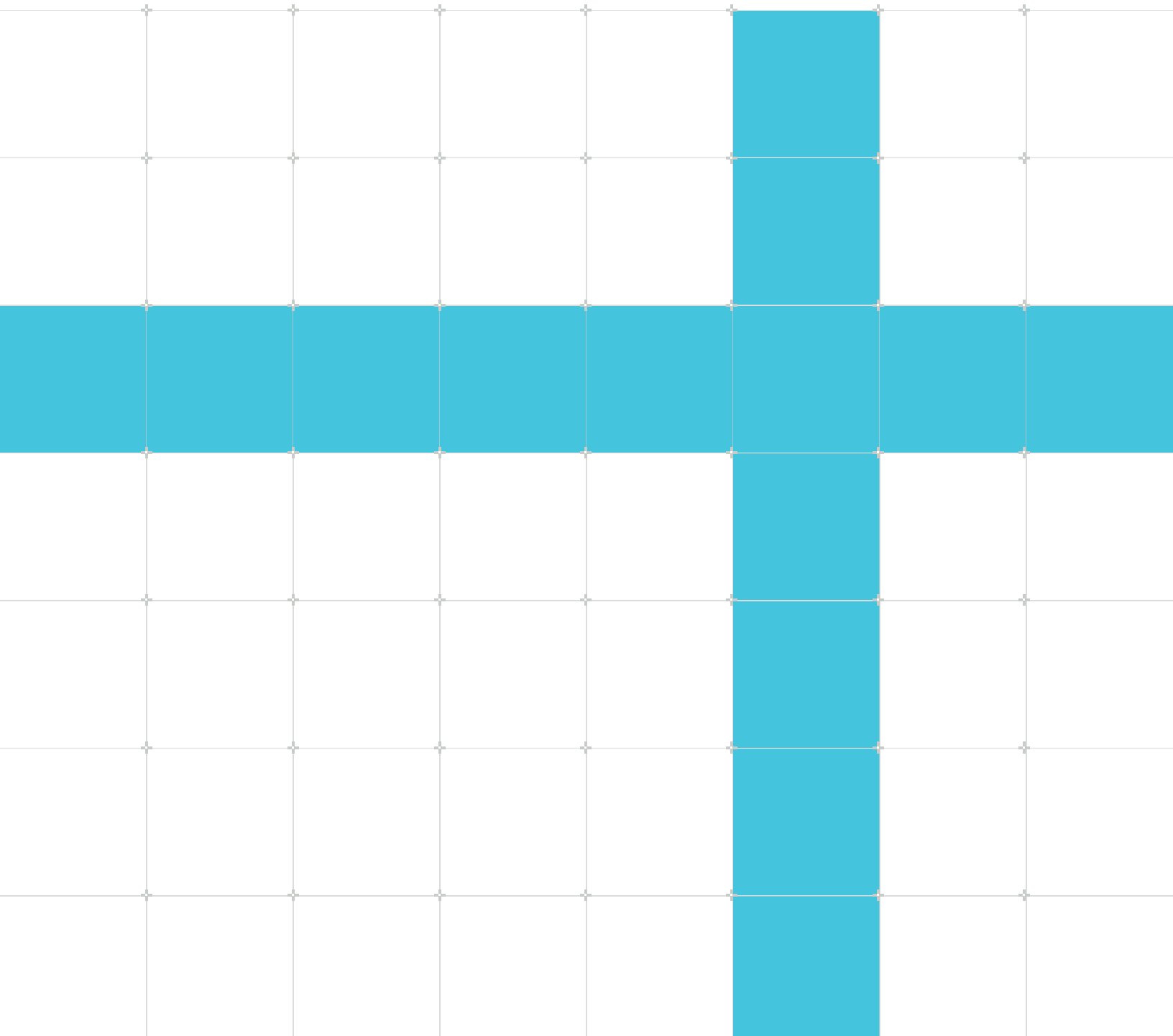
User Guide

Non-Confidential

Copyright © 2023–2025 Arm Limited (or its affiliates).
All rights reserved.

Issue 00

102693_1.8_00_en



Arm® Frame Advisor User Guide

This document is Non-Confidential.

Copyright © 2023–2025 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (102693_1.8_00_en) was issued on 2025-07-24. There might be a later issue at <https://developer.arm.com/documentation/102693>

The product version is 1.8.

See also: [Proprietary notice](#) | [Product and document information](#) | [Useful resources](#)

Start reading

If you prefer, you can skip to [the start of the content](#).

Intended audience

This document is intended for software developers who want to use Arm® Frame Advisor for frame-based graphical analysis of Android applications.

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>.

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Contents

1. Overview of Frame Advisor.....	5
1.1 Platform support.....	9
2. Get started with Frame Advisor.....	12
2.1 Setup tasks.....	12
2.2 Capture a frame burst.....	14
2.3 Analyze the results.....	20
3. Analyzing your frames.....	31
3.1 Navigating your frames.....	31
3.2 Analyze your frame using the Render Graph.....	33
3.3 Analyze object rendering.....	36
3.4 Analyze overdraw.....	41
3.5 Analyze object complexity.....	44
3.6 Analyze model geometry.....	46
3.7 Content metrics in detail.....	48
3.7.1 Mesh complexity.....	48
3.7.2 Mesh efficiency.....	49
3.8 Analyze function calls.....	52
3.9 Analyze shader programs.....	54
4. How to get help.....	58
5. Troubleshooting Frame Advisor.....	60
5.1 My device is not listed in Frame Advisor.....	60
5.2 My application is not listed in Frame Advisor.....	60
5.3 The Framebuffer view is slow to load images.....	61
5.4 A timed out error message appears when I start a capture.....	62
5.5 An unsupported image format message is displayed in the Framebuffer.....	62
5.6 A no image data message is displayed in the Framebuffer.....	63
5.7 Capturing frames from an Unreal Engine application ends unexpectedly.....	63
Proprietary notice.....	64

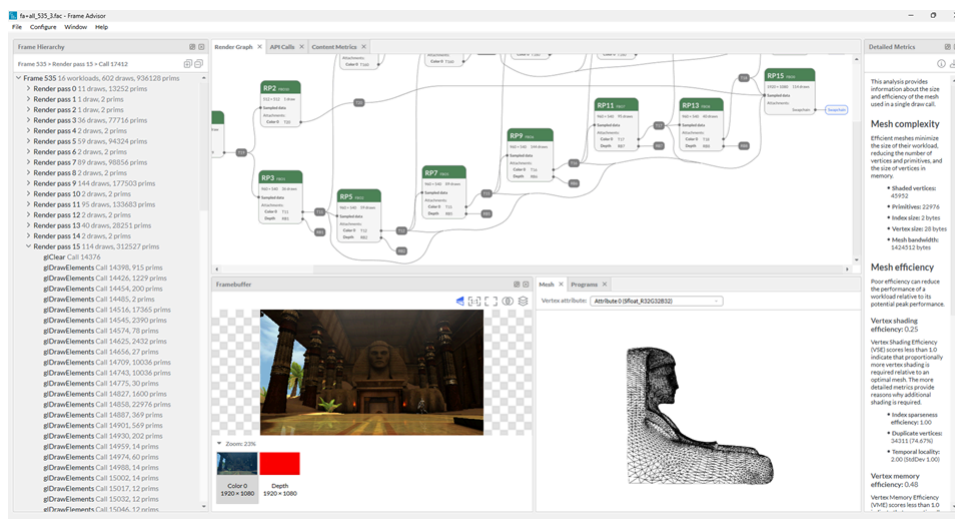
Product and document information.....66
Product status..... 66
Revision history.....66
Conventions..... 67

Useful resources.....69

1. Overview of Frame Advisor

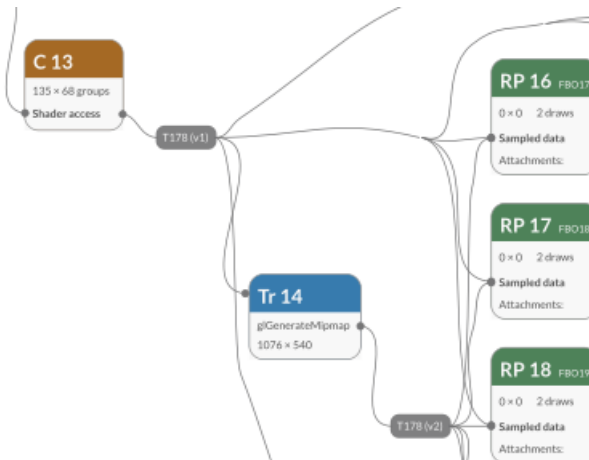
Arm® Frame Advisor captures the API calls and GPU output for frames rendered by your application as it runs on a connected Android device. The analysis views provided by Frame Advisor show how your application is performing on Arm GPUs. Evaluate the data and visual outputs to identify any frames that are causing performance problems with your device or application. Example performance problems include overheating or shortened battery life of your device, and your application running slowly resulting in low frame rates. To improve performance, optimize your application so that the GPU processes workloads efficiently, and makes best use of tile memory with minimal external DRAM accesses.

Figure 1-1: The Frame Advisor Analysis screen



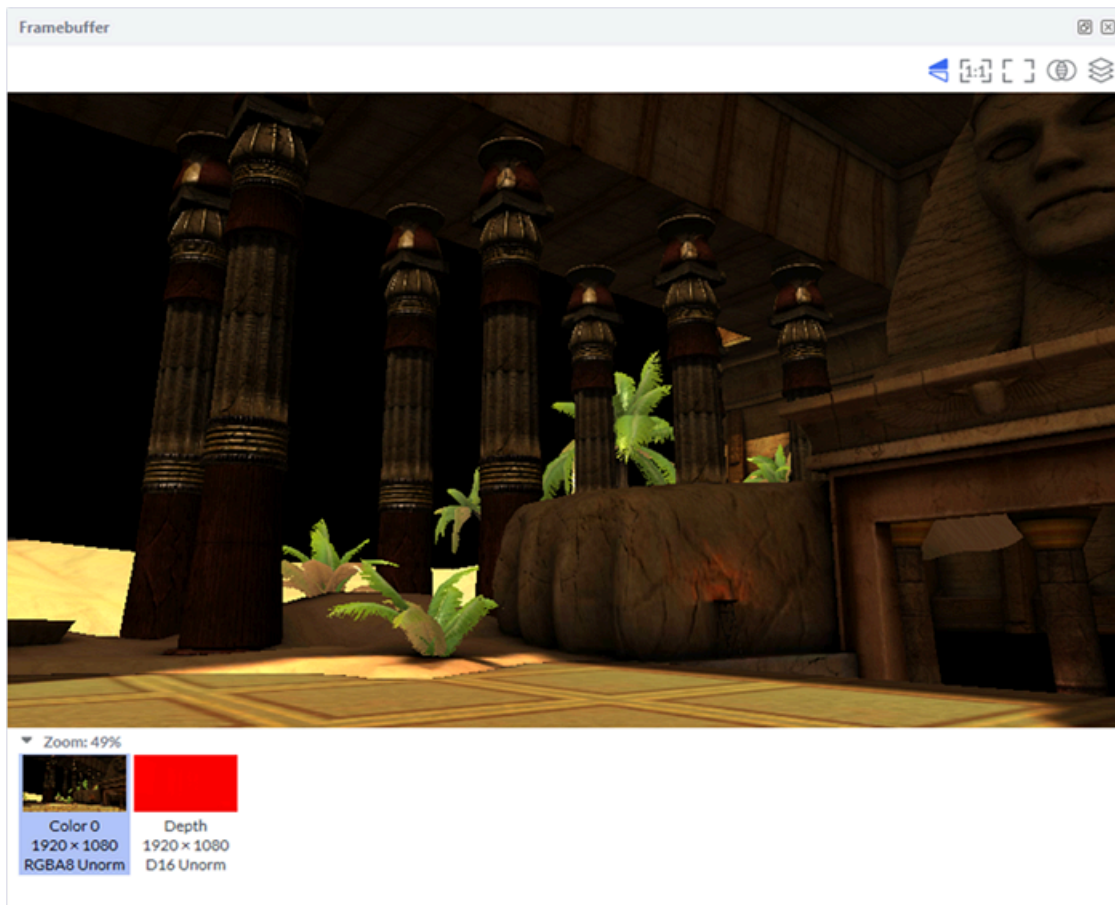
Visualize the structure of your frame

Frame Advisor shows you a graph of the workloads and resources that make up a frame. You can see how render passes, transfer workloads, and compute workloads are processed by the GPU, the commands used in each workload, and the data flows between them. This information helps you to optimize workload processing and memory bandwidth, both of which are essential for best performance using a tile-based Arm GPU.

Figure 1-2: Render Graph

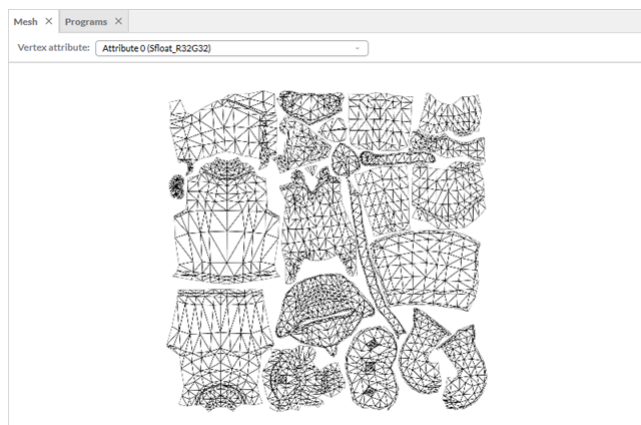
See the framebuffer output

See what the GPU rendered to the framebuffer for each draw call. Step through the draw calls and watch the visual output to see how your application constructed each frame. Identify that draws are rendering in the correct order, or see if any draws render no visual output. To see how objects are drawn, capture frames with overdraw mode and check any areas of white that indicate a pixel is being rendered multiple times.

Figure 1-3: Framebuffer view

Inspect rendered object complexity

Step through the draw calls and look at the primitives to find any objects that do not follow best practice standards. Ensure that the level of detail of each object is suitable based on its distance to the camera. Check if you can use a simpler mesh to improve performance, and reduce the processing cost of draw calls.

Figure 1-4: Mesh view

Evaluate your model geometry

Frame Advisor provides content metrics that you can use with other graphs and visual outputs to analyze your model geometry. Quickly find draw calls that are large or complex, and that use a lot of memory bandwidth to process. You can also use the visual color guidance on the built-in efficiency metrics to see which draw calls are performing badly.

Figure 1-5: Content Metrics view

Render Graph × API Calls × Content Metrics ×								
Frames Render passes Draws								
Frame 543 > Render pass 15 ×								
Frame	Render pass	Call	Prims	Unique indices	Vert size	VSE	VME	
543	15	17047	536	336	28	0.96	0.79	
543	15	17076	22976	45952	28	0.25	0.48	
543	15	17089	202	402	28	0.34	0.48	
543	15	17102	2	6	28	0.75	0.52	
543	15	17115	304	408	28	0.56	0.51	
543	15	17129	915	1180	28	0.57	0.51	
543	15	17141	1229	2308	28	0.32	0.49	
543	15	17155	2	4	36	1.00	0.37	
543	15	17169	17365	38343	28	0.23	0.48	

Perform a detailed analysis of the geometry and buffer memory layout used in a single draw call. Frame Advisor presents clear efficiency metrics, each associated with a specific optimization recommendation.

Evaluate graphics shaders

See which shader programs ran for each workload in different stages of the shader pipeline, and how efficiently the GPU in your connected device processed them. To improve GPU performance, identify which shader programs you can optimize. The [Programs view](#) shows how the shader programs use resources, as well as their cycle costs for the shortest and longest control flow paths. To investigate a shader program further, open the OpenGL ES source or SPIR-V disassembly in the **Source** view and explore the shader source code.

Figure 1-6: Programs view

Program	Shortest path				Longest path				Total emitted				Occupancy	Workre
	A	LS	T	Max	A	LS	T	Max	A	LS	T	Max		
38	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
37	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
36	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
34	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
33	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
30	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
28	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
27	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
26	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
25	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
24	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
23	0.91	3.00	0.00	3.00	0.91	3.00	0.00	3.00	0.91	3.00	0.00	3.00	100	
22	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
21	1.48	3.00	0.00	3.00	1.48	3.00	0.00	3.00	1.48	3.00	0.00	3.00	100	
20	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
19	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
16	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
13	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
5	0.25	3.00	0.00	3.00	0.25	3.00	0.00	3.00	0.25	3.00	0.00	3.00	100	

See [Analyze shader programs](#) to learn more about shader analysis.

Explore OpenGL ES or Vulkan API calls

See the functions and parameters that the application passed to the API, and their return values. API calls are shown in the same order that they are seen by the GPU, at the point that workloads are submitted to a queue. This sequence makes it easier for you to see changes over time, and helps you to locate any problem calls in Vulkan applications.

See Frame Advisor in action

To see an example of Frame Advisor in action, watch our training video [Capture and analyze a problem frame with Frame Advisor](#).

To start capturing frames with Frame Advisor, see [Get started with Frame Advisor](#).

1.1 Platform support

Arm® Frame Advisor supports capturing profiles on a compatible [Android device](#).

Host support

Frame Advisor supports the following host OS versions:

- Windows 11 or later

- Ubuntu 22.04 or later
- macOS 13 (Ventura) or later

Device support

Frame Advisor supports device OS version Android 11 or later.

API support

Frame Advisor supports the following API versions:

- OpenGL ES 2.0 and 3.0-3.2
- Vulkan 1.0-1.2

GPU support

Frame Advisor supports Arm® Mali™ and Arm® Immortalis™ GPUs implementing the Midgard, Bifrost, Valhall, and 5th Generation architectures.

Image formats

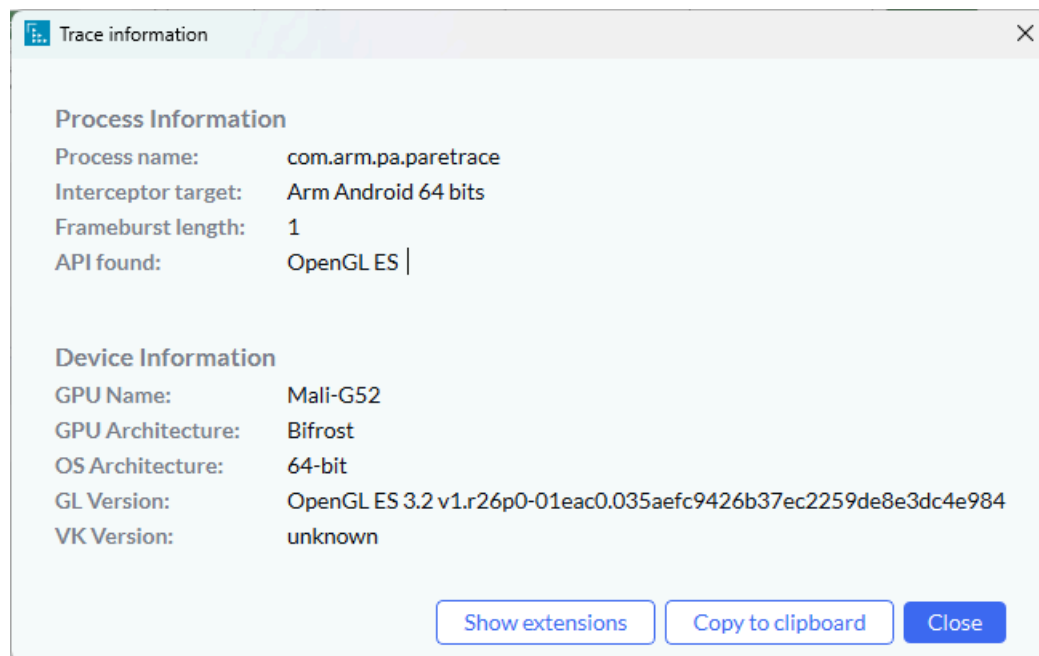
We aim to increase the number of supported image formats in future releases as we continue to develop Frame Advisor. If you experience any problems with unsupported image formats, please contact the Arm Performance Studio team. See [How to get help](#).

Minimum recommended hardware

For the best experience when using Frame Advisor, Arm recommends using a monitor with a minimum spec of 1080P 1920x1080 at 1:1 scaling.

See API and device information

After you have captured a trace, you can see some of this API and device information in the **Trace information** dialog. To open the **Trace information** dialog, click **Help -> Trace information** in the Frame Advisor menu.

Figure 1-7: Trace information dialog

2. Get started with Frame Advisor

Use Arm® Frame Advisor to quickly capture and analyze frames from a mobile game running on a connected Android device.

This tutorial helps you to learn how to interpret the data and find ways to optimize your application. It describes how to:

- Perform [Setup tasks](#) to prepare your computer and device.
- [Capture a frame burst](#) from a mobile game running on a connected device.
- [Analyze the results](#) to look for performance problems.



Frame Advisor is a new tool, which is still in active development. If you encounter problems, or have a feature request, please send feedback to performancestudio@arm.com.

2.1 Setup tasks

Follow these steps to set up your computer and device so that you can analyze your application with Arm® Frame Advisor.

Before you begin

- Frame Advisor supports applications built with OpenGL ES versions 2.0 to 3.2, or Vulkan versions 1.0 to 1.2. Your device must be running Android 11 or later.
- Ensure you have installed Android Debug Bridge ([adb](#)). adb is available with the Android SDK platform tools, which are installed as part of [Android Studio](#). Alternatively, you can download them separately as part of the [Android SDK platform tools](#).
- [Download Arm Performance Studio for free](#) and follow the installation instructions in the [Arm Performance Studio Release Note](#).

Procedure

1. Connect your device to your computer through USB and ensure that the device is switched on.
2. Enable [developer mode](#) on your device.
3. On your device, go to **Settings > Developer Options** and enable **USB Debugging**. If your device asks you to authorize connection to your computer, confirm the connection. Test the connection by entering the `adb devices` command in a command terminal. If successful, the command returns the device ID.

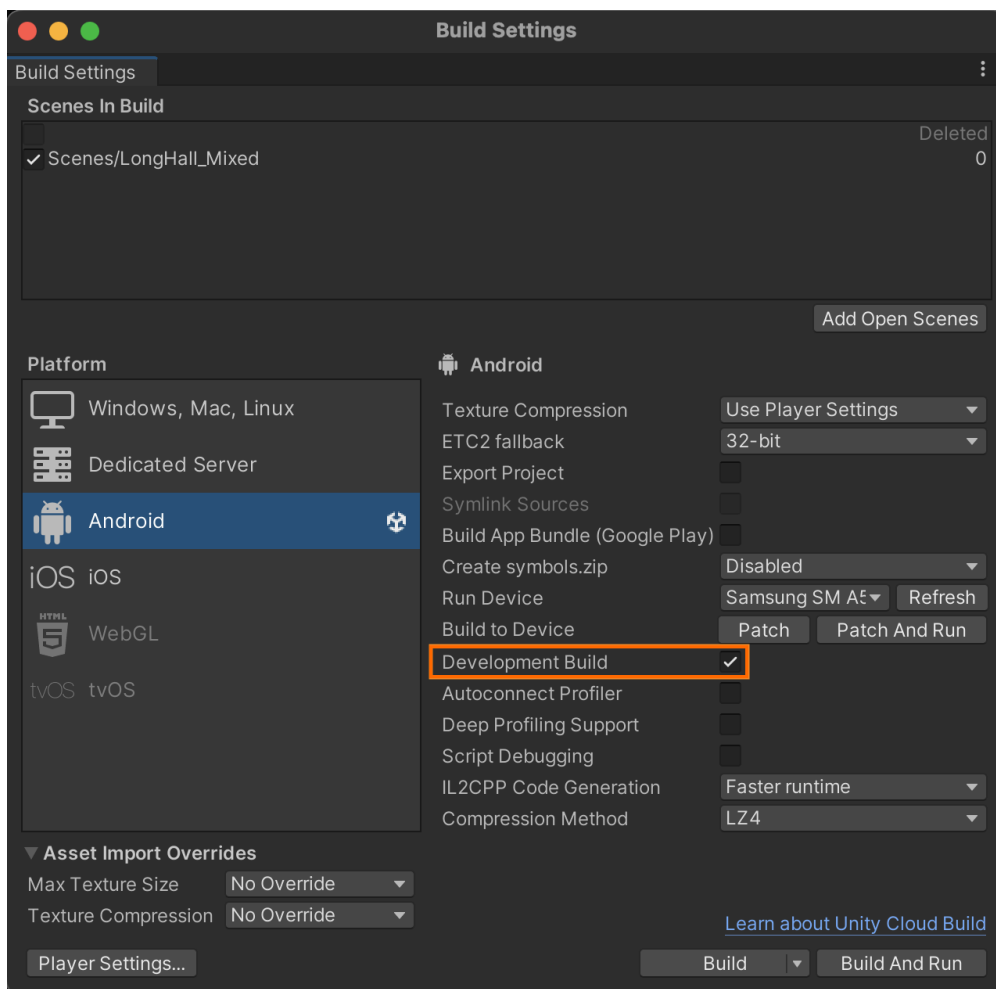
```
adb devices
List of devices attached
ce12345abcdef1a1234    device
```

If you see that the device is listed as unauthorized, try disabling and re-enabling **USB Debugging** on the device, and accept the authorization prompt to enable connection to the computer.

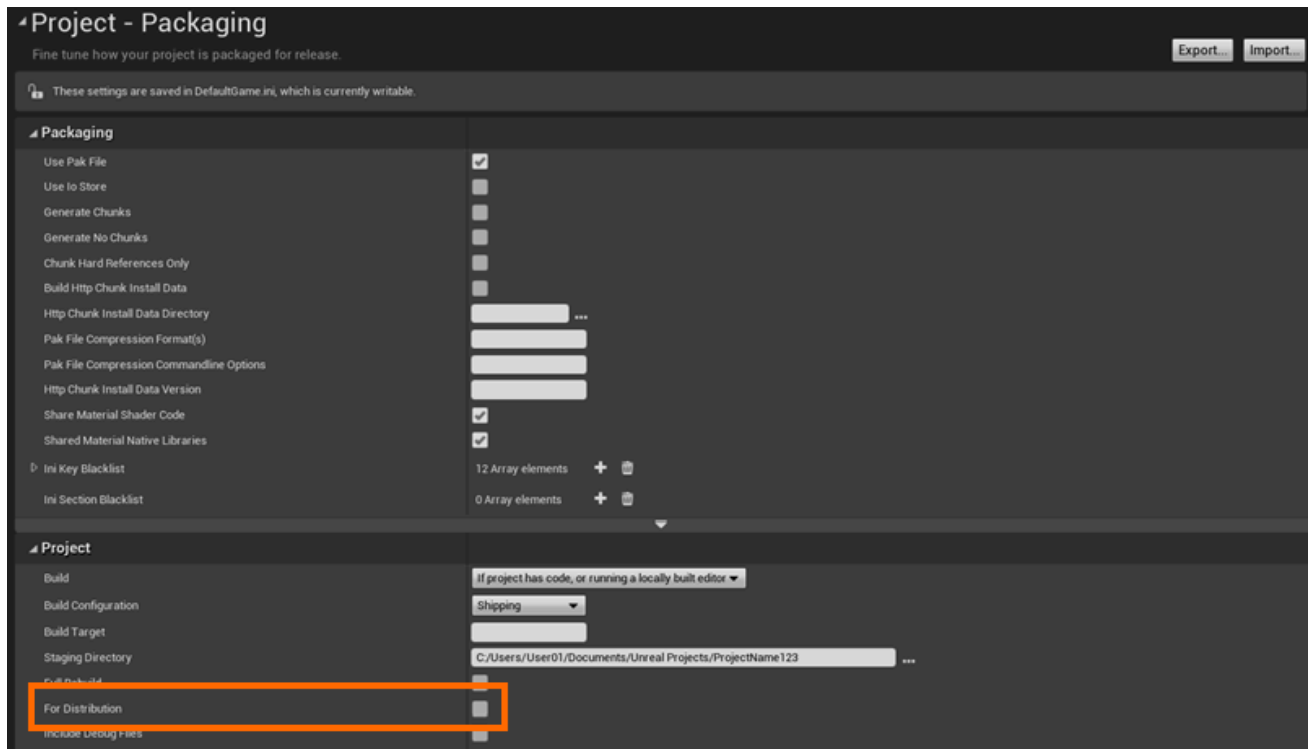
4. Disable permission monitoring. If your phone has **Settings > Developer options > Disable permission monitoring**, ensure that **Disable permission monitoring** is selected. If you do not have the option to disable permission monitoring, you can ignore this step.
5. Install a debuggable build of the application you want to profile on the device.
In Android Studio, create a build variant that includes `debuggable true` in the build configuration. Or you can set `android:debuggable=true` in the application manifest file.

In Unity, select **Development Build** under **File > Build Settings** when building your application.

Figure 2-1: Unity Build Settings



In Unreal Engine, open **Project Settings > Project > Packaging > Project**, ensure that the **For Distribution** checkbox is not set.

Figure 2-2: Unreal Engine Build Settings

6. Unreal Engine users only. You must disable PSO caching because it is not currently supported in Frame Advisor. To disable PSO caching, add the following arguments to your `DefaultEngine.ini` and `DefaultGame.ini` files:

```
[/Script/Engine.RendererSettings]
r.psoprecaching=0

[/Script/AndroidRuntimeSettings.AndroidRuntimeSettings]
Android.Vulkan.NumRemoteProgramCompileServices=0
```

Next steps

Now that your computer and device are connected and set up, the next step is to [Capture a frame burst](#).

2.2 Capture a frame burst

Use Arm® Frame Advisor to capture frames from your application running on the connected device.

Procedure

1. Open Frame Advisor:
 - On Windows, from the **Start** menu, search for **Arm Frame Advisor**, and select the version of the tool that you want to open.

- On macOS, navigate to the <install_directory>/frame_advisor folder, and double-click the FrameAdvisor-gui file.
- On Linux, navigate to the <install_directory>/frame_advisor directory in a terminal, and run the FrameAdvisor-gui file:

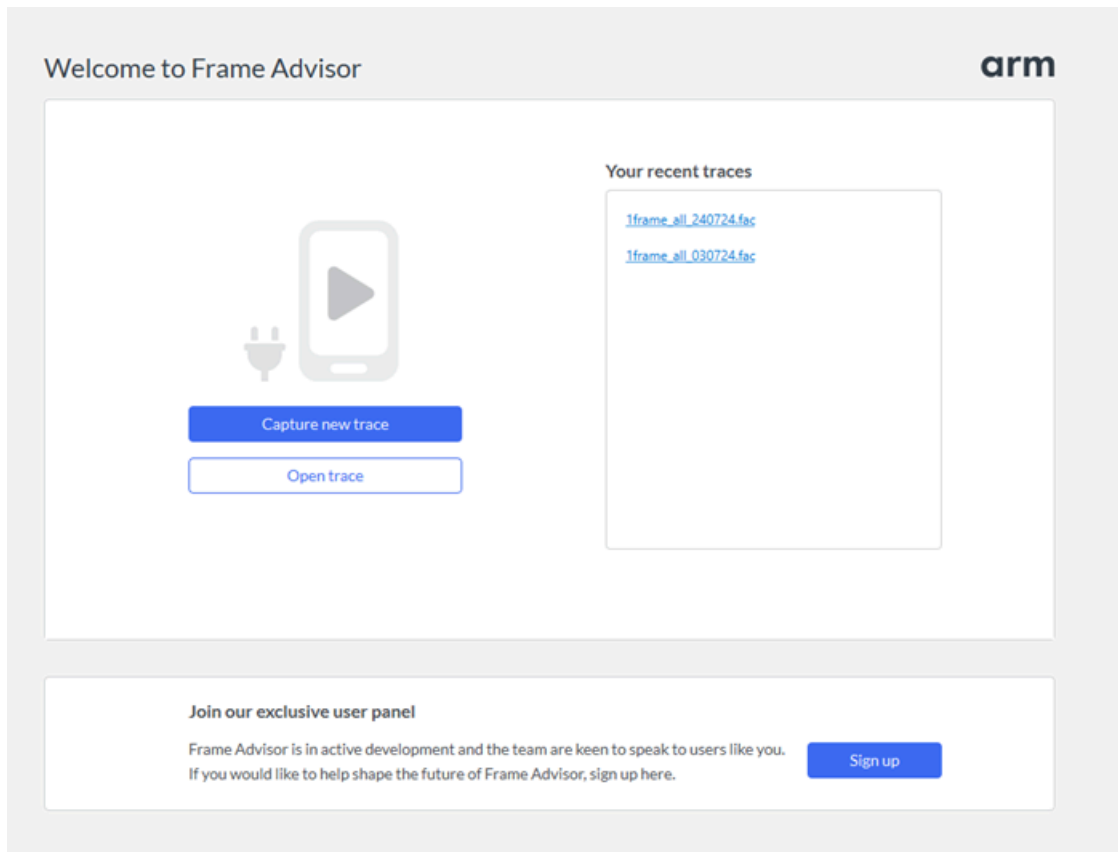
```
cd <install_directory>/frame_advisor  
./FrameAdvisor-gui
```



By default, Frame Advisor opens in the theme determined by your system settings. To view Frame Advisor in a dark or light theme, click **Configure -> Open preferences -> Display** in the Frame Advisor menu, and select your preferred theme option.

2. When Frame Advisor launches, you can either capture a new trace or open an existing one.

Figure 2-3: Frame Advisor launch screen



To capture your problem frames, select **Capture new trace**.

Frame Advisor lists the connected devices and the applications installed in the **Device Connection** screen.

Figure 2-4: Device connection screen

Select device

Filter search

Refresh

Status	Name	Serial
Connected	SM-A325F	RF8R41HN7DA

Select application

☒ Show debuggable only

Filter search

Refresh

Debug	Application	Activity
✓	com.android.gl2jni	.GL2JNIActivity
✓	com.Arm.ArmSampleAPK	com.unity3d.player.UnityPlayerActivity
✓	com.arm.pa.paretrace	.Activities.SelectActivity

Configure session

API settings

☒ OpenGL ES

☐ Vulkan

Application settings

☐ Pause on connect

☐ Terminate on disconnect

Next >

Cancel

3. Select your device, and the application and activity that you want to evaluate.

**Note**

Frame Advisor can only start applications and activities that are debuggable and launchable.

If your device or application is not listed, see [My device is not listed in Frame Advisor](#), or [My application is not listed in Frame Advisor](#) for help.

4. Alternatively, to start non-launchable activities you can use a trace tool.
Click **Show advanced options**.

If required, select the **Override activity** checkbox, then enter the activity name in the text box.

Optionally enter any command-line arguments that you want to run. You can see valid argument options in [Specification for intent arguments](#).

Select the application.

Figure 2-5: Override activity

Select device

Filter device Refresh

Status	Name	Serial
Connected	SM-A325F	RF8R41HN7DA

Select application

Hide advanced options Filter application Refresh

☒ Show debuggable only ☒ Show launchable only

☒ Override activity: PlayerActivity

Arguments: --es fileName /apitrace/trace_repo/tracefile.pat

Application

- com.android.gl2jni
- com.Arm.ArmSampleAPK
- com.arm.pa.paretrace

Configure session

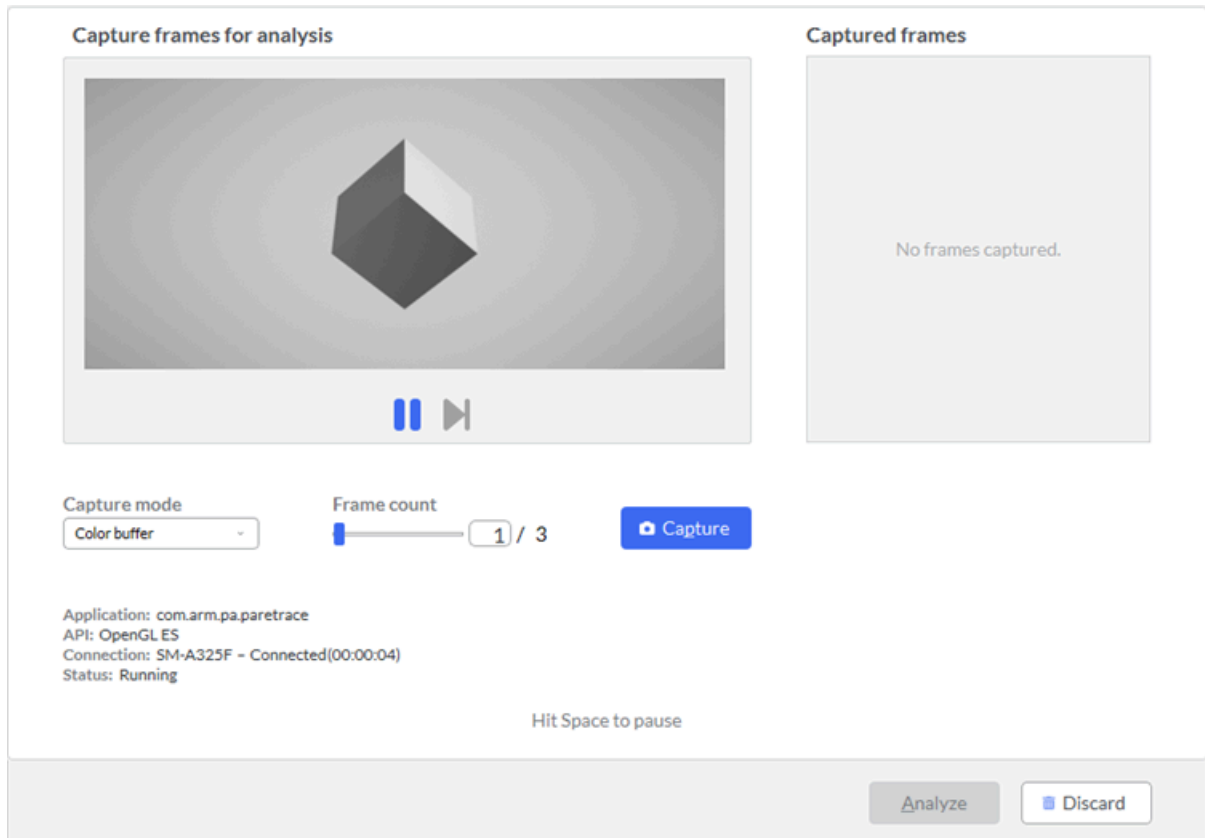
5. Configure options for your capture session.
Frame Advisor automatically selects the OpenGL ES API. If your application uses the Vulkan API, change the selection in the API settings to **Vulkan**.

Optionally, you can use the **Application settings** to pause the application when Frame Advisor connects to it, and to stop the application when the capture is complete.

Click **Next >** to continue.

Unless you chose the **Pause on connect** option in the **Device connection** screen, the application starts automatically on the device.

6. Select options for your capture session in the **Capture** screen.

Figure 2-6: Frame Advisor capture screen

The default **Capture mode**, **Color buffer**, captures only color framebuffer content. Optionally change the mode to **All attachments**, to capture color and depth framebuffer content, or capture only **Overdraw** content, if you would like to measure the number of times that pixels are overdrawn during rendering. You can also adjust the **Frame count** to the number of frames that you want to capture.



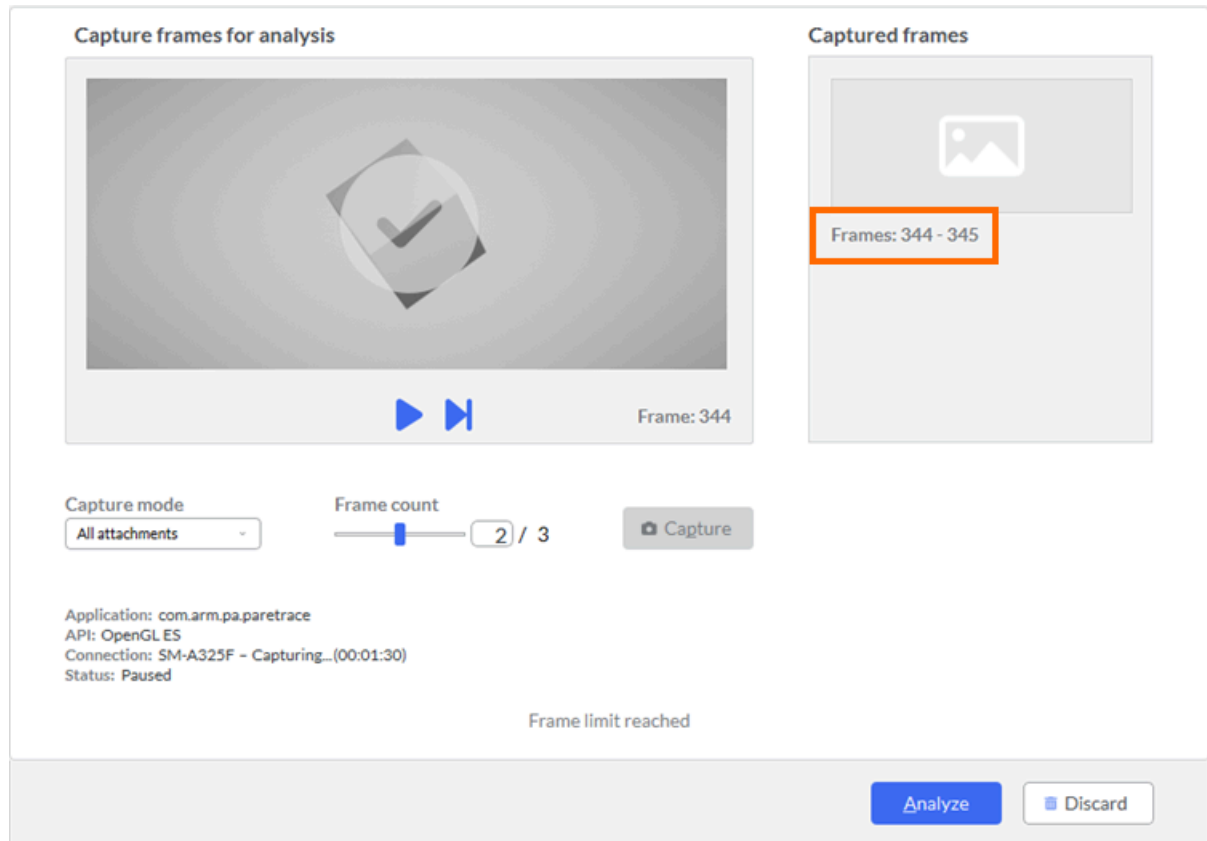
Note

You can capture up to 3 frames. The larger the frame count, the longer the time it takes for Frame Advisor to capture the frames.

7. When you get to the part of your application that you are interested in, optionally click the **Pause** button to pause the application just before you get to the problem frame. The current frame number is displayed.
When the application is paused, use the **Advance one frame** button to step through the frames more accurately.
8. Click the **Capture** button to start capturing the frames.
Frame Advisor starts capturing from the start of the next frame. This may take a few minutes to complete, depending on how many frames you are capturing.

When Frame Advisor finishes capturing the frames, the frame burst is displayed with the captured frame numbers.

Figure 2-7: Captured frame numbers



9. Click the **Analyze** button to see the results. It may take a few minutes to analyze the data.

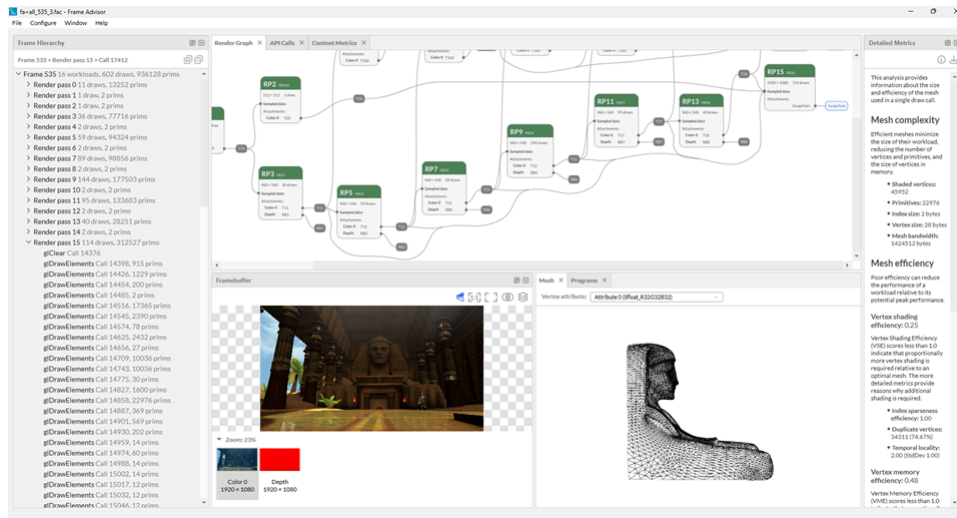
Next steps

Now that you have captured a frame burst, it's time to look at the results. See [Analyze the results](#).

2.3 Analyze the results

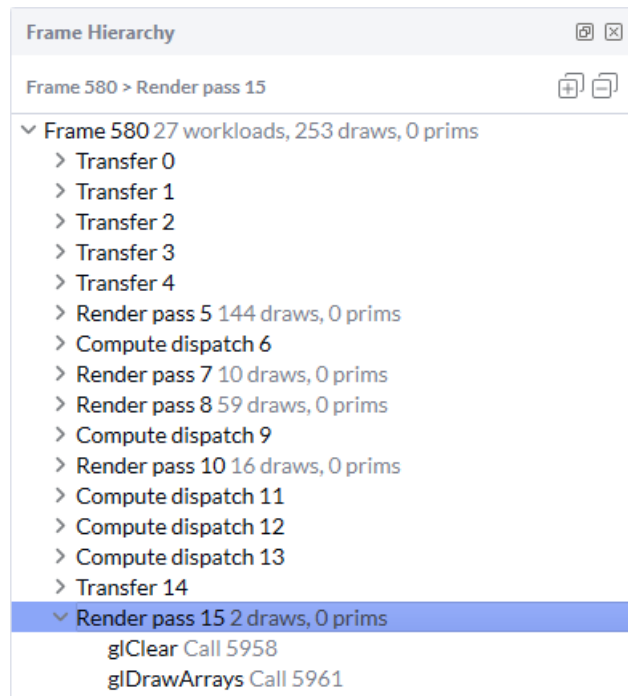
When you have captured some frames, click **Analyze**. Alternatively, open an existing trace from the **Welcome** screen. Arm® Frame Advisor processes the data and presents it in the **Analysis** screen. Explore each frame to evaluate how efficiently they rendered on the device.

Figure 2-8: Example Analysis screen



Frame Hierarchy view

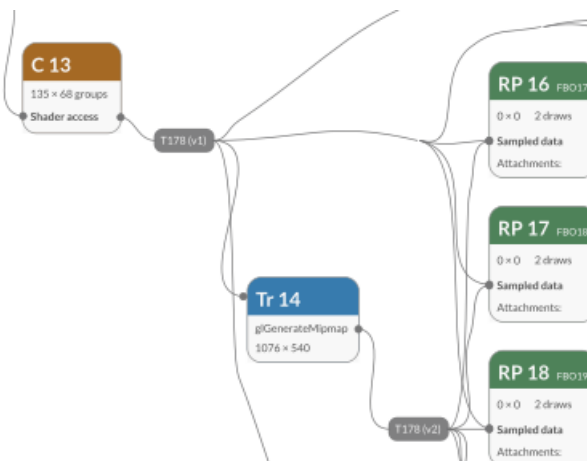
The **Frame Hierarchy** view lists the captured frames in a tree structure. Expand a frame to view the workloads, and calls within it. Select an item in the hierarchy to navigate to it, and to show its output data for the item in other views of the **Analysis** screen.

Figure 2-9: Frame Hierarchy view

Contextual information helps you to find frames that need further investigation. For example, frames with the highest number of draws or primitives are expensive to process. See [Navigating your frames](#).

Render Graph view

The **Render Graph** shows an overview of the processing and data management operations that are performed to create the final rendered frame. Use the graph to see the data flow between workloads in the frame, and how resources, such as textures and render buffer objects, are produced and consumed.

Figure 2-10: Render Graph

Data flows from the left side of the render graph to the right side of the render graph. The **Swapchain** node represents the end of the rendering process, and shows the final image displayed on the screen of your device.

Transfer workloads

Transfer workloads are non-rendering operations that output an image or a texture for an associated transfer command. They are an important part of managing resources in the GPU pipeline. For OpenGL ES only, when you select a transfer workload in the **Frame Hierarchy** or **Render Graph** view, the output image or texture shows as an attachment in the **Framebuffer** view.

A transfer workload node in the render graph, indicated by **Tr**, contains useful information about the transfer command and resolution of the image or texture.

Figure 2-11: Transfer workload label



Note

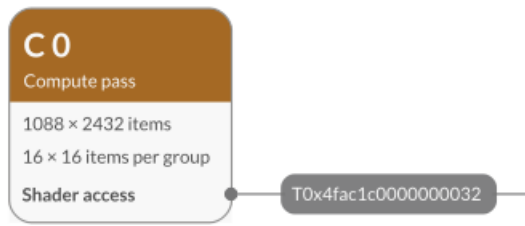
Transfer workloads that only read and write buffers are hidden in the render graph.

Compute workloads

Compute workloads are data-intensive tasks that run on the GPU and use compute shaders to process data. They are identified by `Dispatch` commands, and perform tasks in parallel to the rendering pipeline, such as image processing and lighting calculations. To optimize performance, distribute and synchronize compute workloads across a frame for best use of memory bandwidth.

A compute workload node in the render graph, indicated by **C**, shows the following useful information:

- Label that shows the compute node name and user debug label, if set.
- Work space size shows the space size as the number of groups for processing. Work space size is displayed for direct calls only when total work items cannot be determined.
- Total work items is shown instead of work space size only when both the work space size and work group size are known. Total work items shows the total amount of work that requires processing in the form of total items and number of items in each group.
- Work group size is displayed as the number of items in each group. The work group size helps you to understand how the workload is split when the compute task is processed. This value is available for Vulkan traces only.

Figure 2-12: Compute workload label**Note**

- Some compute nodes have inputs and outputs that cannot connect to any resource nodes on the render graph. These floating nodes are included in the render graph to show that they exist.
- Compute workloads that only read and write buffers are hidden in the render graph.

Render passes

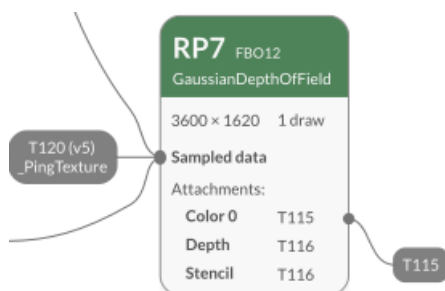
Render passes are the main processing blocks that define a rendering operation in the graphics rendering pipeline.

Render passes are identified when either:

- An API call changes the framebuffer, or when queued rendering is flushed in the OpenGL ES API.
- A `VkRenderPass` is specified in the Vulkan API.

A render pass node in the render graph provides useful information about the render pass:

- Labels that show the render pass name and API name. The label might also contain a user debug label, or a decoded debug label.
- Draw count and resolution
- Output attachments

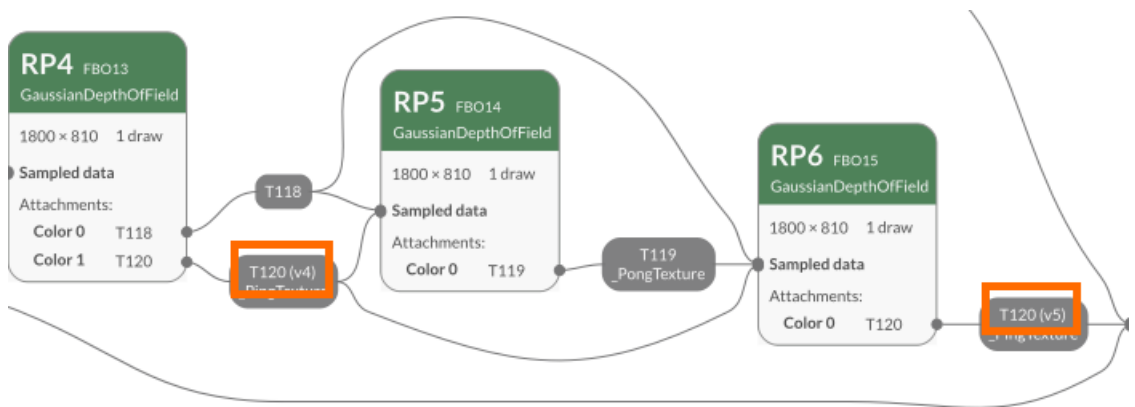
Figure 2-13: Render graph label

Short form names are used on the label:

- RP: Render pass
- FBO: Framebuffer object
- T: Texture
- RB: Renderbuffer

Render passes consume resources and input attachments, then render an output image as a set of attachments. Resources are represented by the gray nodes. These resources are textures, images, and renderbuffers that are optionally consumed and modified by later render passes. When a resource is modified, the version number on the label increments each time the resource is modified. Resources that are not modified do not show a version number.

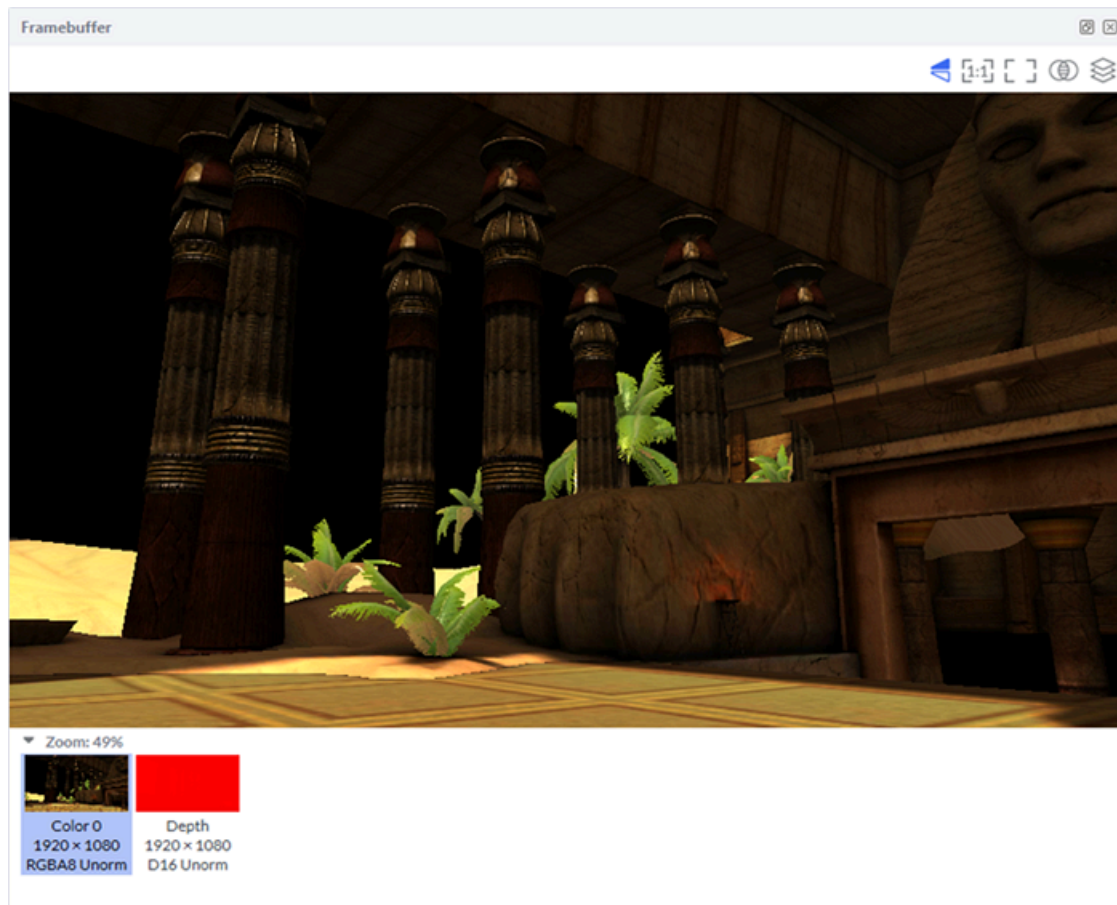
Figure 2-14: Render graph resource version



Use the **Render Graph** to review your frame construction, management of data flow, and identify expensive parts of a frame that process workloads inefficiently. See where you can improve workload performance by minimizing the number of external memory accesses. See [Analyze your frame using the render graph](#).

Framebuffer view

The **Framebuffer** view visualizes the rendered output for your captured frames. To see how your frames are composed from the GPU rendered output, step through draw calls and transfer workloads in the **Frame Hierarchy** view and see how the output changes in the **Framebuffer** view.

Figure 2-15: Framebuffer view

Any attachments used in the framebuffer are shown as thumbnails underneath the rendered framebuffer image. You can check that the resolution and color format of the input and output attachments provides the appropriate visual precision without exceeding the GPU bandwidth or memory requirements.

To help you analyze rendering efficiency, use the **Framebuffer** view where you can check object ordering, and levels of overdraw and shading. See [Analyze object rendering](#) and [Analyze overdraw](#).

Mesh view

The **Mesh** view shows the mesh of objects in your rendered output. Use the **Mesh** view with the **Framebuffer** view to analyze the complexity of objects in relation to their size and position on screen. Use the **Vertex attribute** menu to visualize different attributes of your mesh.

Figure 2-16: Mesh view

This view helps you to identify issues such as micro triangles, long and thin triangles, and draw calls that submit the same geometry, all of which cause unnecessary processing. See [Analyze object complexity](#).

Programs view and Source view

The **Programs** view shows a table of performance metrics for shader programs that are used for a selected workload and stage in the shader pipeline. Use the **Programs** view to find which shader programs you can optimize to ensure the shader cores are kept busy with work to process, and help the GPU run more efficiently.

Figure 2-17: Programs view

Mesh X Programs X

Filter by frame

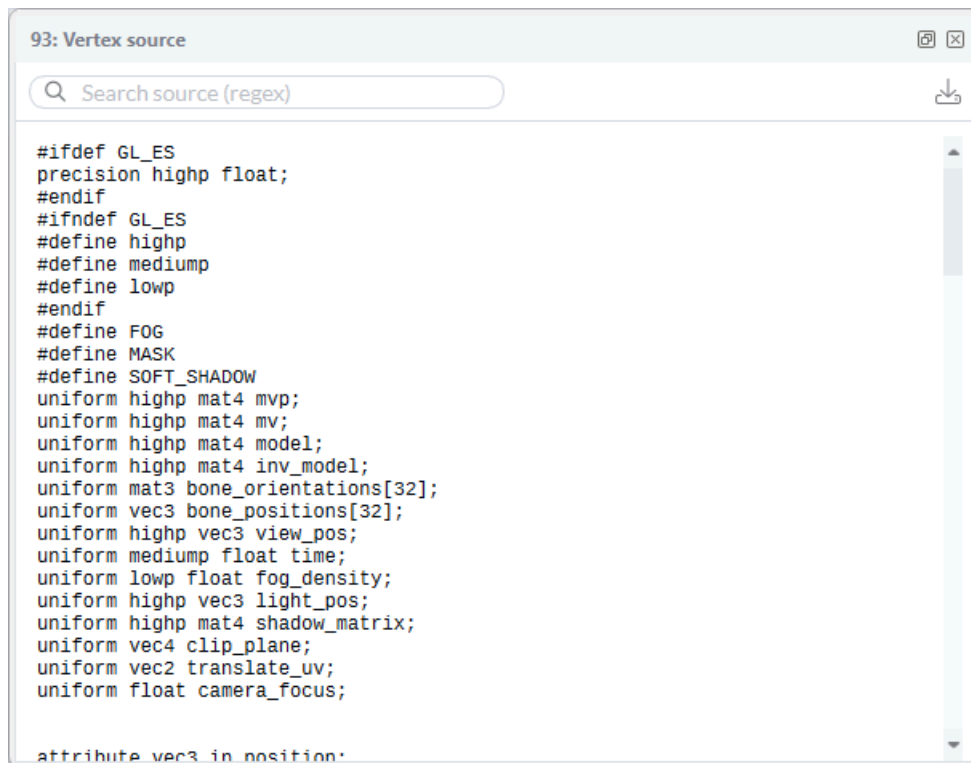
Stage: Vert position Vert main Fragment

Program		Shortest path				Longest path				Total emitted				Occupancy	Work re
		A	LS	T	Max	A	LS	T	Max	A	LS	T	Max		
	38	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
	37	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
	36	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
	34	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
	33	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	1.27	3.00	0.00	3.00	100	
	30	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	28	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	27	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	26	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	25	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	24	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	23	0.91	3.00	0.00	3.00	0.91	3.00	0.00	3.00	0.91	3.00	0.00	3.00	100	
	22	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	21	1.48	3.00	0.00	3.00	1.48	3.00	0.00	3.00	1.48	3.00	0.00	3.00	100	
	20	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	19	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	16	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	13	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	1.40	3.00	0.00	3.00	100	
	5	0.25	3.00	0.00	3.00	0.25	3.00	0.00	3.00	0.25	3.00	0.00	3.00	100	

To find computationally expensive shader programs, and see how efficiently your shader programs are processed by the GPU in the connected device, use the filter to focus your attention on a specific workload or shader stage. The table of metrics provides a breakdown of cycle cost for the functional units in the shader core, as well as resource usage.

Sort the metrics to identify parts of the shader code that you can simplify to improve processing. Double-click anywhere along a row that is of interest or right-click and select **Open source** from the menu. The **Source** view opens where you can review the associated shader code.

Figure 2-18: Source view



For example, to help keep the GPU busy and working efficiently, see if the shaders are spilling variables to stack memory and where you need to reduce precision to 16-bit. To determine if you can improve object meshes or rendering to reduce processing cost, compare the metrics in the **Programs** view with the visualizations in the **Framebuffer** view and **Mesh** view.

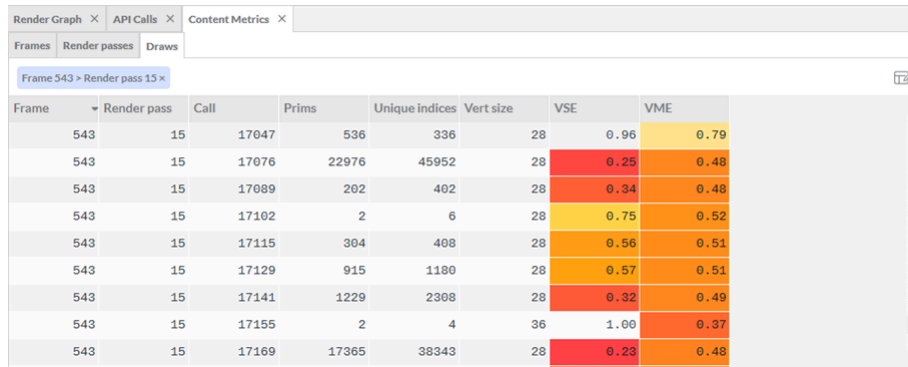
See [Analyze shader programs](#) to learn more about shader analysis.

Content Metrics view

The **Content Metrics** view shows a table of calculated rendering metrics to help you find inefficient rendering that is affecting the performance of your application. To focus on a range of interest, you can filter the data by frame, render pass, or draw call. You can also use the visual color guidance on the vertex efficiency metrics to instantly see how a draw call is performing. A white background indicates that the efficiency metric is over a specified threshold value. If the efficiency metric is below the specified threshold value, the background color gradually changes shade from yellow

through to orange and then red, depending on its numeric distance from the threshold value. Set the **Color ramp threshold** in the **Display** settings of the **Configure -> Preferences** dialog.


Figure 2-19: Content Metrics view



Frame	Render pass	Call	Prims	Unique indices	Vert size	VSE	VME
543	15	17047	536	336	28	0.96	0.79
543	15	17076	22976	45952	28	0.25	0.48
543	15	17089	202	402	28	0.34	0.48
543	15	17102	2	6	28	0.75	0.52
543	15	17115	304	408	28	0.56	0.51
543	15	17129	915	1180	28	0.57	0.51
543	15	17141	1229	2308	28	0.32	0.49
543	15	17155	2	4	36	1.00	0.37
543	15	17169	17365	38343	28	0.23	0.48

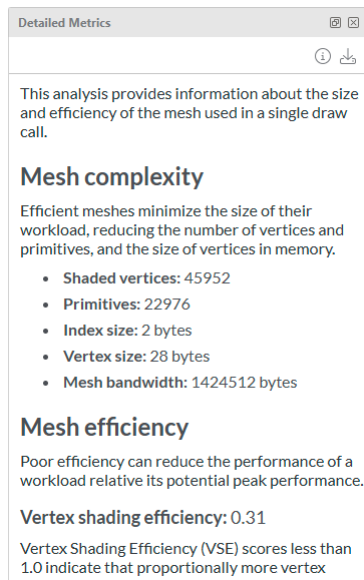
Click a column heading to sort the table by that metric from lowest to highest. Click the column heading again to change the sorting from highest to lowest.

For example, you can identify the most computationally expensive render pass containing the most draw calls. Or you can find the most complex draws in the render pass with the highest number of primitives. See [Analyze model geometry](#).

Add and remove columns of data using the **Column selector**. To open the **Column selector**, click the **Edit columns** icon . You can also change the order of the selected columns that are shown in the **Content Metrics** view.

Detailed Metrics view

The **Detailed Metrics** view helps you to understand the efficiency of a mesh for a selected draw call, and its impact on memory bandwidth and processing cost.

Figure 2-20: Detailed Metrics view

The metric data is shown as a summary that enables you to quickly identify aspects of your mesh that you can optimize. To see a description about each metric, click the **Show detailed information** button ⓘ.

Complicated meshes containing [many triangles](#) are expensive for the GPU to process. Depending on the position and size of the object on the screen, the mesh might not gain any benefit from the added complexity. Reducing the number of triangles in a mesh dramatically reduces shader cost. To add detail efficiently, use normal maps to simulate details as a texture within the accurately drawn silhouette of an object. [Normal maps](#) change the surface normals of an object, and how light interacts with the surface, instead of using expensive geometry in the mesh.

Just as important as complexity, is whether the mesh of the object is drawn with good [index reuse](#), to minimize the number of unique vertices required per triangle. See [Content metrics in detail](#).

API Calls view

The **API Calls** view shows the function calls that were made by the application, and what values were returned from the graphics system. When you select a workload, or draw call in the **Frame Hierarchy** view, the associated API call is highlighted in the **API Calls** view.

Figure 2-21: API Calls view

Render Graph × API Calls × Content Metrics ×		
<input type="text" value="Search function calls (regex)"/>		
Call	Return value	Function call
40		glEnableVertexAttribArray(index=2)
41		glVertexAttribPointer(index=2, size=3, type=GL_FLOAT, normalized=GL_FALSE, stride=48, pointer=8)
42		glEnableVertexAttribArray(index=0)
43		glVertexAttribPointer(index=0, size=4, type=GL_FLOAT, normalized=GL_FALSE, stride=48, pointer=28)
44		glEnableVertexAttribArray(index=1)
45		glVertexAttribPointer(index=1, size=4, type=GL_UNSIGNED_BYTE, normalized=GL_FALSE, stride=48, pointer=44)
46		glDrawElements(mode=GL_TRIANGLES, count=19080, type=GL_UNSIGNED_SHORT, indices=116664)
47		glDisableVertexAttribArray(index=2)

The triangle icon on a function call ▴ indicates that you can navigate to the call in the frame hierarchy. Either double-click the function call, or right-click and select **Navigate to call**.

Search the function calls table for mis-used rendering commands, and check for error messages that might indicate issues that affect performance. See [Analyze function calls](#).

Related information

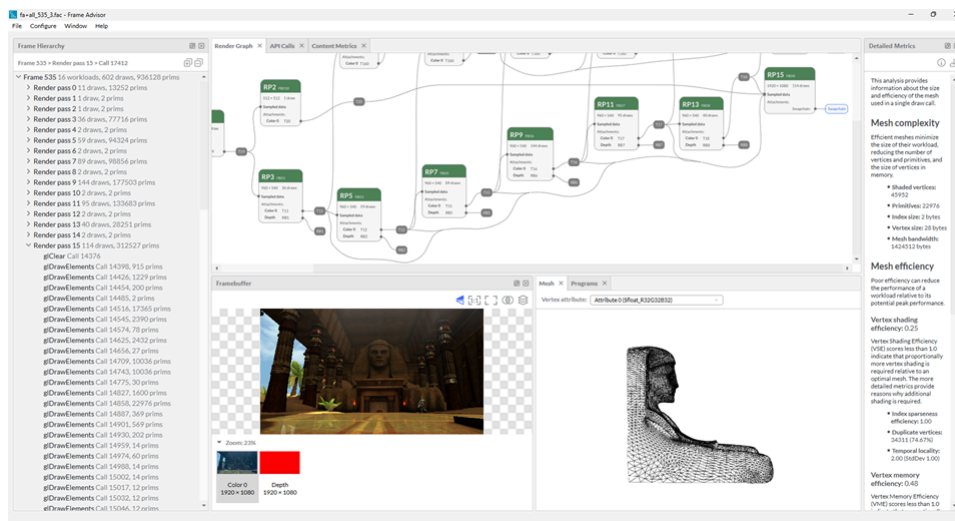
- [Analyzing your frames](#)
- [How to get help](#)

3. Analyzing your frames

The **Analysis** screen presents analysis data and visualizations in a set of distinct views, which enable you to investigate different aspects of your captured frames. Explore the views to identify problem areas in your application, or areas where you can improve performance. The views synchronize with each other as you navigate the workloads and calls in your captured frames.

The **Analysis** screen opens after you have captured your frames and clicked the **Analyze** button. To analyze an existing trace file, open the **Landing** screen and click **Open trace**.

Figure 3-1: Example Analysis screen



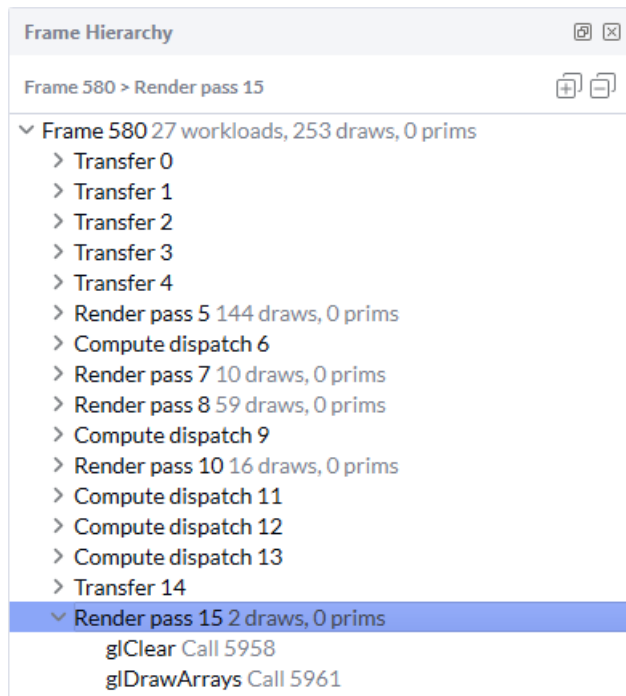
3.1 Navigating your frames

The **Frame Hierarchy** view lists the frames that you captured from your application in a tree structure. Expand the frames to see the workloads, and calls contained within them. When you select an item in the hierarchy, all views in the **Analysis** screen synchronize to the selected navigable item.

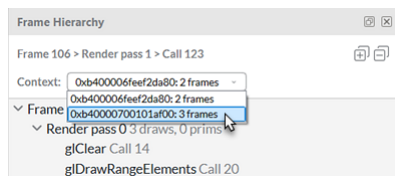


Some views are arranged in tabs. To see the hidden views, select or undock the **Mesh**, **API Calls** and **Content Metrics** views.

To help you to find items that require further analysis, each item in the **Frame Hierarchy** view is presented with contextual information. For example, you may want to select a frame that contains the most draws or primitives to investigate it further. The selected item is displayed in the breadcrumb navigation at the top of the **Frame Hierarchy** view.

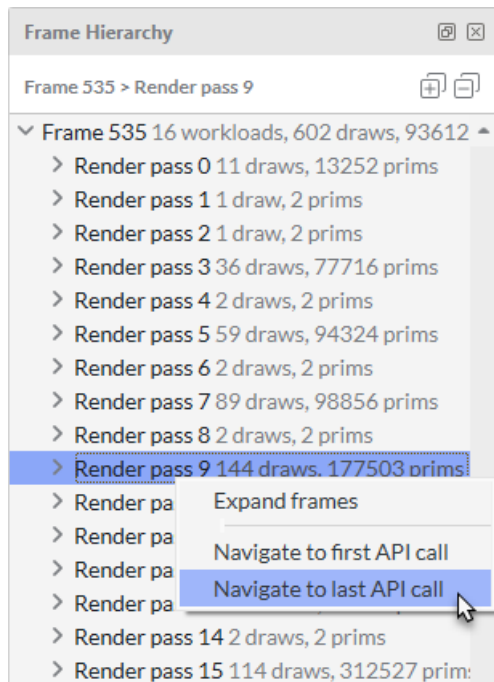
Figure 3-2: Frame Hierarchy view

If your trace uses multiple contexts, you can choose the context from the drop-down menu. If your trace uses one context only, the drop-down menu is not visible.

Figure 3-3: Context selection

As you select a frame, the relating views update to show the output from the last API call in the frame. Use these views with the **Frame Hierarchy** to find the problem area.

To help navigation, you can use the right-click menu to navigate directly to the first or last API call, for example when a render pass contains a high number of draw calls. Alternatively, use the mouse or the arrow keys to step through items in the navigation tree.

Figure 3-4: Navigate to the last API call

Here are the kinds of problems to look for:

- Look for render passes with high numbers of draw calls. Draw calls are expensive for the CPU to process, so it is important to reduce them where possible. Check if these draw calls actually render visible changes to the framebuffer. If not, look at [software culling techniques](#) to see if they can be eliminated. Draws could be outside of the frustum, or behind other objects.
- Look for instances where many identical objects are each being drawn individually. There could be an opportunity to reduce the number of draw calls by [batching multiple objects](#) into a single draw.



Tip

To learn more about how to avoid common API problems when building graphics for mobile, watch *Episode 2.4* of the *Mali GPU training series* [Engine and API best practices](#)

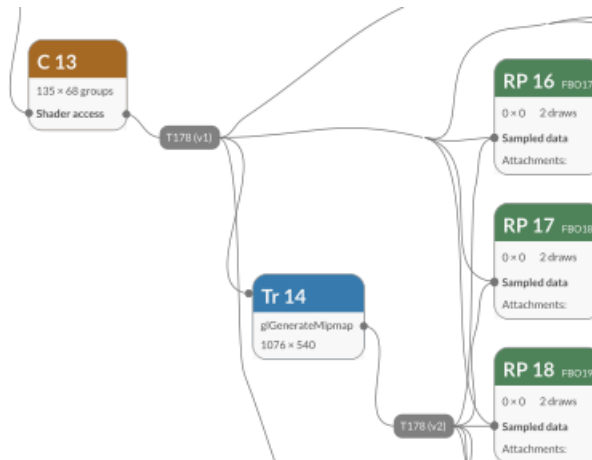
3.2 Analyze your frame using the Render Graph

The **Render Graph** shows the rendering workloads, data flow, and resources that are produced and consumed for the frame selected in the **Frame Hierarchy** view. Use the render graph to analyze the performance of your frame and identify where you can improve data flow or workload processing. For good performance, ensure that your render graph shows a data flow that minimizes external memory accesses and makes good use of tile memory. Also ensure that workloads and resources are processed efficiently, and that they all contribute to the final **Swapchain** output.

Procedure

1. In the **Frame Hierarchy** view, select the frame you are interested in exploring further. The **Render Graph** visualization updates to show the workloads, and resources for the selected frame.

Figure 3-5: Example render graph



Click on a workload in the graph to navigate to it, or right-click and navigate to the first or last API call. Notice that information shown in the other views of the **Analysis** screen update for the selected item.

2. In the **Render Graph** view, use your mouse to zoom and pan around the graph. See how workloads and resources are processed to create the rendered **Swapchain** output for the selected frame. Identify any render passes that are doing the most work, which you can optimize.



Right-click in the graph area and select **Fit to view** for an overall view of rendering operations performed for the selected frame. Click **Reset view** to go back to the initial zoom level.

To help you understand what is happening in your frame, labels provide contextual information about the workload performed.

Information shown for render passes:

- Index of the render pass, and the framebuffer object that it relates to.
- Number of draw calls in the render pass.
- Rendered resolution
- Input and output attachments

Information shown for transfer workloads:

- API function command

- Image resolution

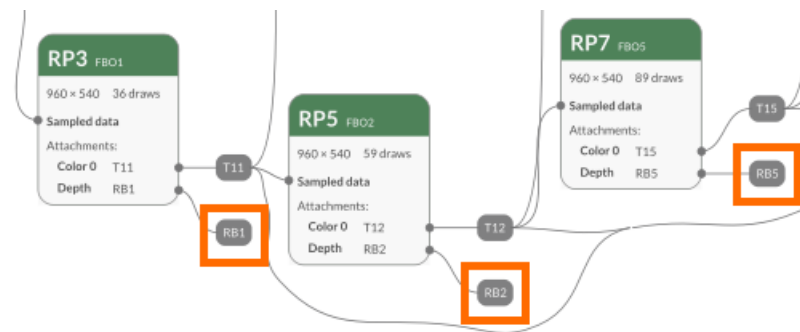
Information shown for compute workloads:

- API compute command and user debug label, if set.
 - Work space size shows space size as the number of groups for processing. Work space size is displayed for direct calls only when total work items cannot be determined.
 - Total work items is shown instead of work space size only when both the work space size and work group size are known. Total work items shows the total amount of work that requires processing in the form of total items and number of items in each group.
 - Work group size, displayed as the number of items in each group. The work group size helps you to understand how the workload is split when the compute task is processed. This value is available for Vulkan traces only.
3. Look for render passes that are consuming bandwidth and accessing memory unnecessarily:
- Render passes where the attachment of one render pass is immediately input to another without the version changing. Merging unused render passes can prevent excess memory reads and writes.
 - Render passes that do not clear or invalidate input attachments at the start of the render pass show the attachments as inputs to the render pass node. Reading these input attachments from external DRAM at the start of the render pass consumes read memory bandwidth. If you do not want to use the rendered output from a previous frame, ensure that you clear or invalidate all input attachments at the start of the render pass, before any draw calls are made.

You can see if clears and invalidates exist within a render pass in the **Frame Hierarchy** view, and in the **API Calls** view.

- Render passes that do not invalidate output attachments at the end of the render pass show as an output from the render pass node. Writing these output attachments to external DRAM at the end of a render pass consumes write memory bandwidth. If the output attachment is no longer needed and does not contribute to the final rendered output, ensure that you invalidate it so that it is discarded at the end of the render pass.

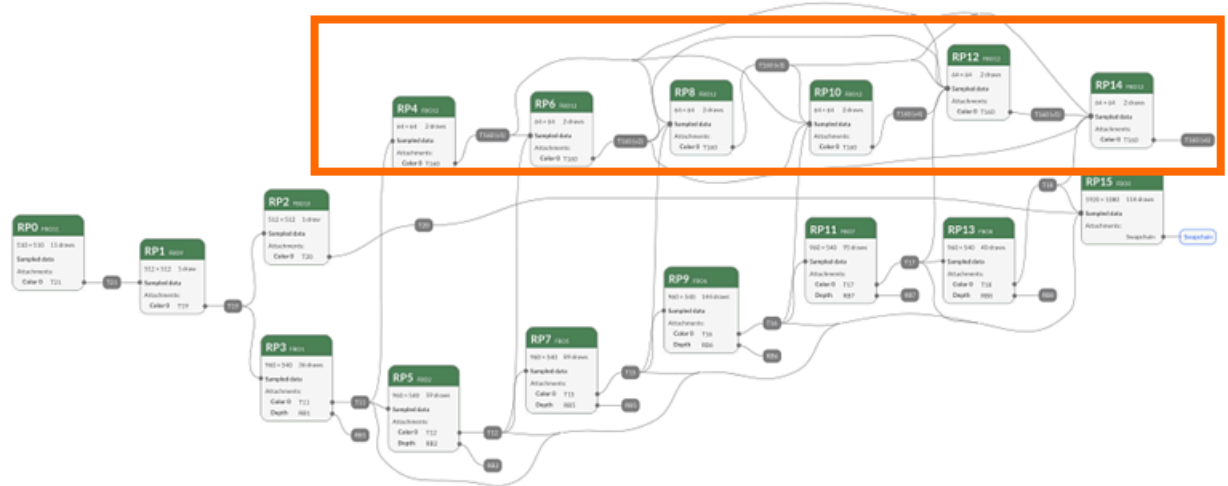
Figure 3-6: Output attachments that require discarding



- Render passes that are not used in the final **Swapchain** output produce unnecessary data in your model. Ensure that all render passes that are submitted to the GPU are actually useful rendering operations, with no redundant rendering. To prevent unnecessary processing of

workloads and memory consumption, discard any unused attachments at the end of the render pass.

Figure 3-7: Unused nodes that are not used in the final output



Next steps

When you are happy with the construction of your frame, you can now analyze your rendering operations and check the complexity of your model. See [Analyze object rendering](#) and [Analyze object complexity](#) for more information.

3.3 Analyze object rendering

The **Framebuffer** view shows a visualization of the rendered output of your captured frames. The visualization enables you to review what was rendered to the screen for each draw call. Review the framebuffer output to identify problem objects that do not follow best practice standard, and see where you can improve the performance cost of draw calls.

Procedure


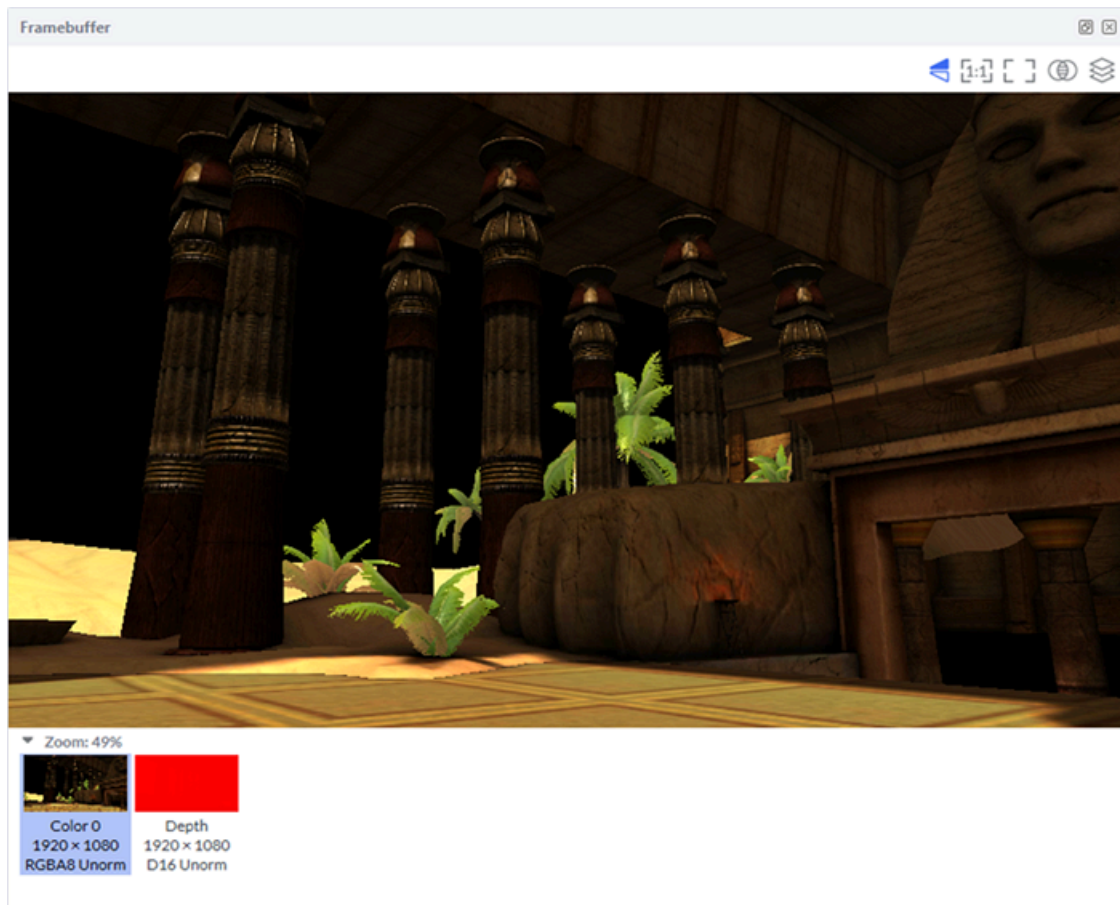
1. In the **Frame Hierarchy** view, expand a frame that you are interested in exploring further to see the render passes and draw calls within it.
The **Framebuffer** visualization shows the rendered output after the final draw call of the selected frame. If required, use the **Flip vertically** button  to adjust the vertical orientation so that it appears the correct way around.

Figure 3-8: Framebuffer view

- Expand a render pass in the **Frame Hierarchy** view. To see how your frame is composed, use the arrow keys or mouse to step through each draw call in sequence, and watch how the framebuffer changes.
Any color, depth and overdraw attachments used in the framebuffer are shown as thumbnails underneath the framebuffer visualization. Click the thumbnail to show the rendered attachment.

As you step through the draw calls, check that the input and output attachments use color or depth formats with the appropriate bit depth, data type and normalization mode. Using the correct format improves the quality of your rendered output, and performance of GPU memory accesses. Use high-precision formats, such as `RGBA32F`, only when it is required. Using high-precision formats can unnecessarily increase memory bandwidth, and impact GPU performance. To ensure the framebuffer renders the output correctly, and avoid visual artifacts in your rendered output, check that the resolution of the input and output attachments is suitable for the expected resolution of the render pass.

Figure 3-9: Format and data type of framebuffer attachments

If you need more space for the framebuffer visualization, click the **Hide framebuffer attachments** button to temporarily hide the attachment thumbnails.

Figure 3-10: Hide the framebuffer attachments



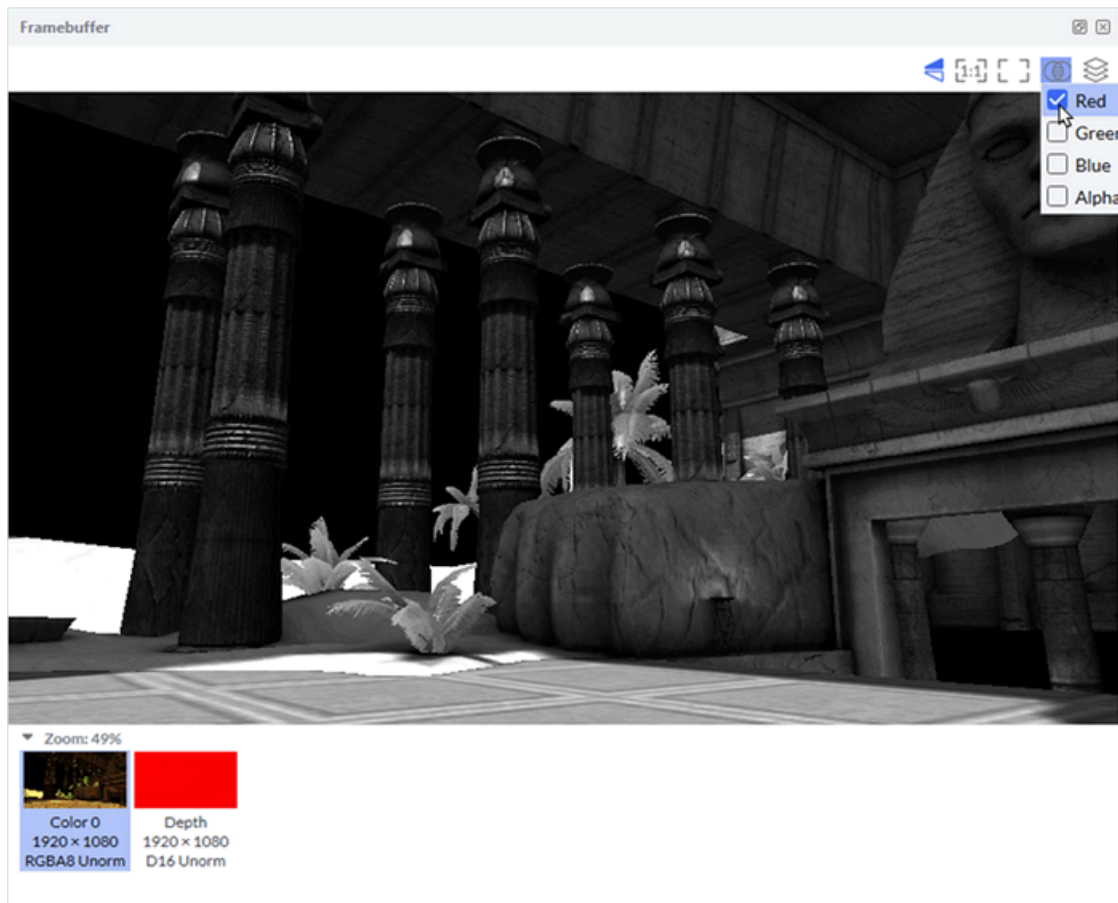
3. To see more detail, use the mouse controls to zoom and pan around the visualization. To quickly rescale the image, use the appropriate button:
 - **Zoom to 1:1** button  to resize the image to the original size.
 - **Fit to view** button  to resize the image inside the visualization window.
4. If you store texture information in RGBA channels, you can select any combination of the channels so that you can look at just the data that you care about. Click the **Select RGBA channels** button, and select the channels that you want to show.

Figure 3-11: Select RGBA channel

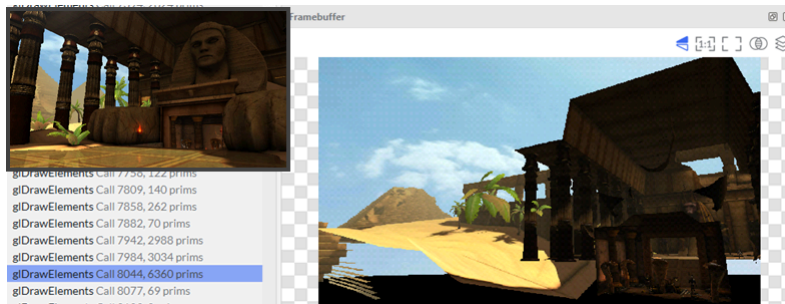
The RGBA color values, and the image coordinates, for the pixel under the cursor are displayed in the framebuffer by default. Click the **Select pixel information** button  and clear the checkbox if you do not want to see this information.

5. Check the ordering of objects to avoid high levels of overdraw.
Ensure that opaque objects are rendered in order from front-to-back, starting with objects closest to camera. This order prevents shading of occluded objects, which reduces overdraw and unnecessary processing before fragment shading.

Ensure transparent objects are rendered in a back-to-front order, starting with the objects that are furthest away from the camera. This order helps blending because it ensures that objects that are further away from the camera are already in the framebuffer when the transparent layer is rendered.

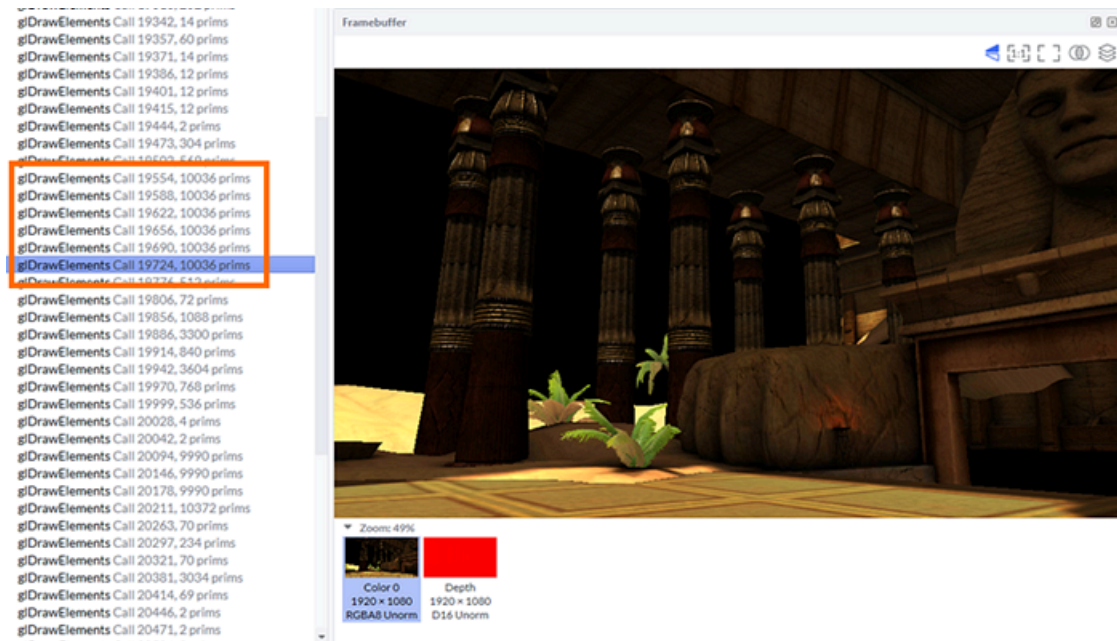
See [Analyze overdraw](#) for more information.

6. Identify any draw calls that do not make visible changes to the framebuffer output.
Check if your application is drawing objects that are either not visible on screen, or are occluded by other objects. Ensure your GPU or CPU is [culling](#) any primitives that are not visible on screen.

Figure 3-12: Unused objects in the Framebuffer

In this example, you can see the output of the selected draw call in the framebuffer visualization. Only a small part of this output is shown in the final rendered frame, which is displayed in the inset image.

7. Check for multiple identical objects that are drawn individually, instead of being processed as a batch in a draw call.

Figure 3-13: Multiple identical objects in the Framebuffer

In this example, the draw calls in the **Frame Hierarchy** show that each pillar is drawn as an individual object in separate draw calls. Processing individual objects is expensive. To help improve performance, merge the identical objects into a single draw call to reduce the draw call count.

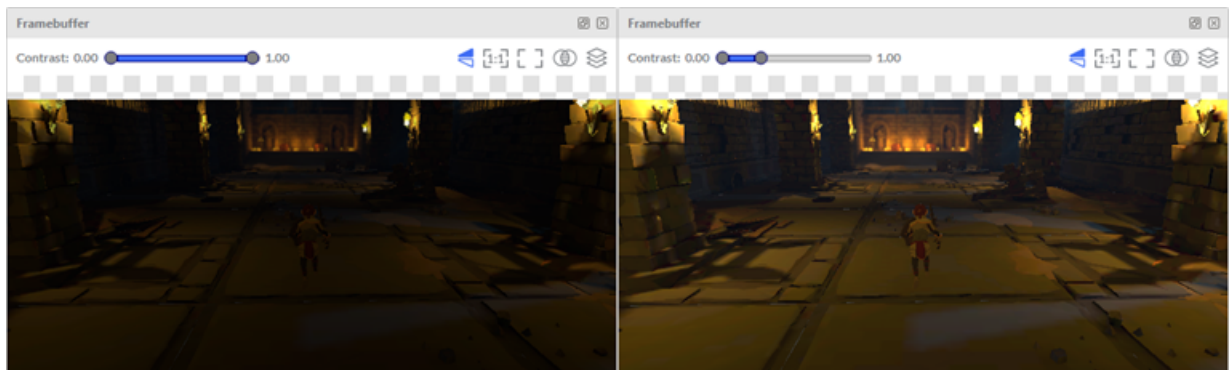
8. If your captured frames contain HDR images, the **Framebuffer** displays a range slider that you can use to adjust the exposure level in the visualization. The minimum and maximum values of the range are determined by data from the image. To reveal details of interest in the darkest or brightest areas of the HDR image, move the minimum and maximum slider controls to a suitable range.



The adjusted minimum and maximum values are maintained as you continue to step through draw calls.

The following example shows the same HDR image, but the range has been adjusted in the second image to reveal more details and lighting effects in the rendered output.

Figure 3-14: HDR image comparison



9. To check if the complexity of objects is appropriate for their size on screen, compare the framebuffer output with the **Mesh** view as you step through the draw calls. Very fine-detailed objects that are further away from the camera can negatively impact performance because of the [triangle density](#) in the mesh of an object. Large numbers of very small triangles are expensive for the GPU to process, often with no visual benefit. See [Analyze object complexity](#).

Use mesh LODs to select simpler geometry as objects move further away from the camera. Check the efficiency of your mesh in the **Detailed Metrics** view. See [Content metrics in detail](#).

Next steps

Investigate the efficiency of your mesh further in the **Detailed Metrics** view. See [Analyze object complexity](#) and [Content metrics in detail](#) for more information.

3.4 Analyze overflow

Frames that contain multiple layers of overlapping objects can experience overflow, which causes unnecessary shading of pixels that are hidden on lower layers. High levels of overflow impacts performance of your application and wastes GPU resources by rendering pixels that are not shown on screen. Frame Advisor enables you to capture overflow for your frames, and explore which objects have the highest levels of overflow.

Before you begin

[Capture a frame burst](#) with the **Capture mode** set to **Overflow**.

Procedure

1. In the **Frame Hierarchy** view, expand a frame that you are interested in exploring further to see the render passes and draw calls within it.

The **Framebuffer** visualization updates to show the rendered output after the final draw call of the selected frame.

The overdraw attachments used in the framebuffer are shown as thumbnails underneath the framebuffer visualization. If required, click the thumbnail to show the rendered attachment.


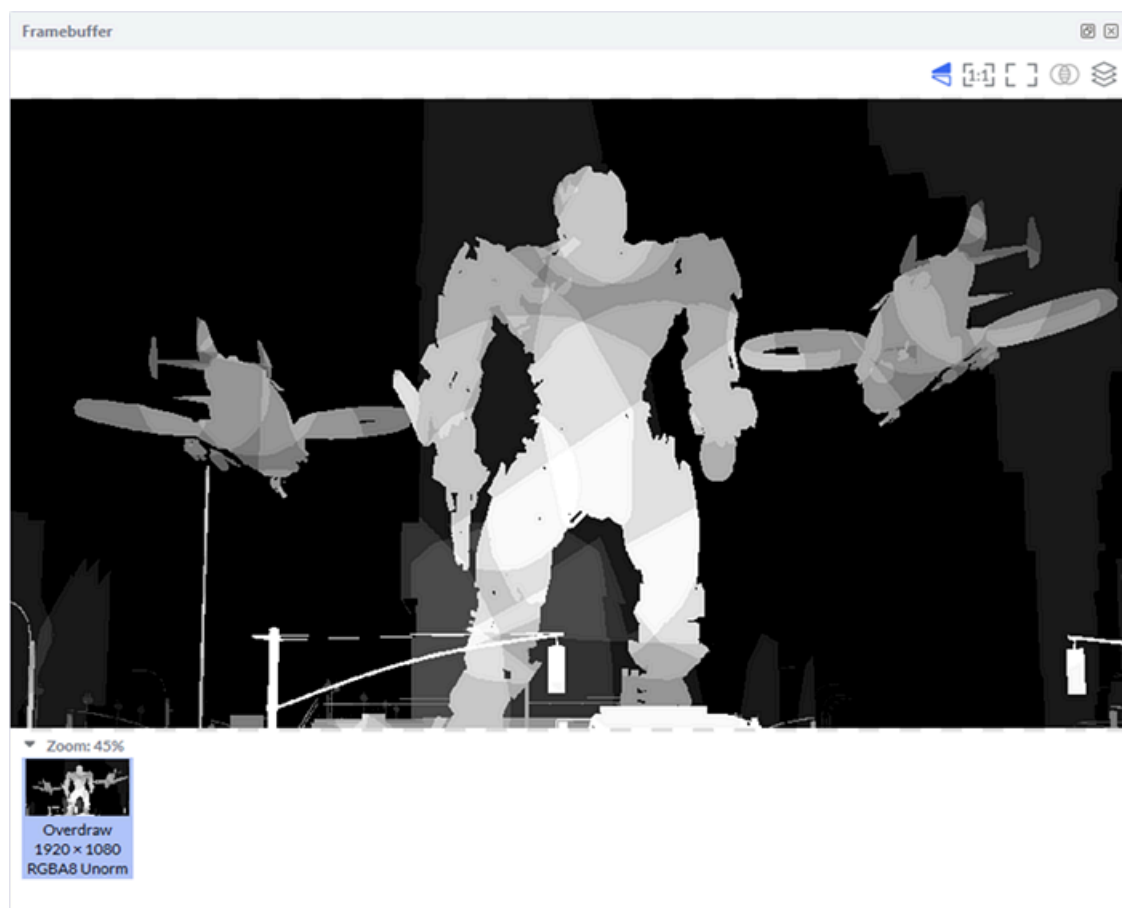


If your image is shown the incorrect way around, use the **Flip vertically** button  to adjust the vertical orientation.

Figure 3-15: Framebuffer view



2. Move the mouse pointer around the visualization to see the levels of overdraw in the rendered output. Every time a pixel is rendered to the framebuffer, the overdraw value increases. The more white the area, the higher the level of overdraw.

Figure 3-16: Framebuffer overflow view

3. To see how your frame is composed, in the **Frame Hierarchy** view expand a render pass in the selected frame, and use the arrow keys or mouse to step through each draw call in sequence. To see more detail, use the mouse controls to zoom and pan around the visualization. To quickly rescale the image, use the appropriate button:
 - **Zoom to 1:1** button  to resize the image to the original size.
 - **Fit to view** button  to resize the image inside the visualization window.
4. Check that objects are ordered by distance to the camera. Ensure that opaque objects are rendered in order from front-to-back, starting with objects closest to camera. This order enables the GPU to use [early ZS testing](#). Early ZS testing prevents shading of occluded objects, which reduces overdraw and unnecessary processing before fragment shading.

Minimize the use of transparent objects because they are expensive to process. If your frame uses transparent objects, expect to see some level of overdraw. Ensure that transparent objects are rendered in a back-to-front order, starting with the objects that are furthest away from the camera. This order helps blending because it ensures that objects that are further away from the camera are already in the framebuffer when the transparent layer is rendered.

5. As you step through the draw calls, identify any draw calls that do not make visible changes to the framebuffer output or are occluded by other objects. Ensure your GPU or CPU is [culling](#) any primitives that are not visible on screen.
6. If object rendering is acceptable, check that the complexity of the object mesh is appropriate and does not use small triangles. Compare the framebuffer output with the **Mesh** view as you step through the draw calls. Large numbers of very small triangles cause [triangle density](#) and are expensive for the GPU to process, often with no visual benefit. See [Analyze object complexity](#) for more information.

Next steps

Investigate the efficiency of your mesh further in the **Detailed Metrics** view. See [Analyze object complexity](#) and [Content metrics in detail](#) for more information.

Related information

[Avoiding overdraw in 2D games](#)

[High overdraw optimization advice](#)

3.5 Analyze object complexity

The **Mesh** view shows a visualization of the primitives drawn by the selected draw call. The visualization enables you to see how an object was constructed, and whether objects are the right level of detail for their size and position on screen. Compare the **Mesh** view with data in the **Framebuffer** and the **Detailed Metrics** views to identify problem objects that do not follow best practice standards, and reduce the processing cost of draw calls.

Procedure

1. In the **Frame Hierarchy** view, expand a frame that you are interested in exploring further to see the render passes within it.
2. Expand a render pass in the **Frame Hierarchy** view to see the draw calls within it, then step through the draw calls to look for objects that have degenerate primitives, or draw calls that have lots of primitives compared to other draw calls.

When you select a draw call, the **Mesh** view shows a visualization.

Figure 3-17: Mesh view



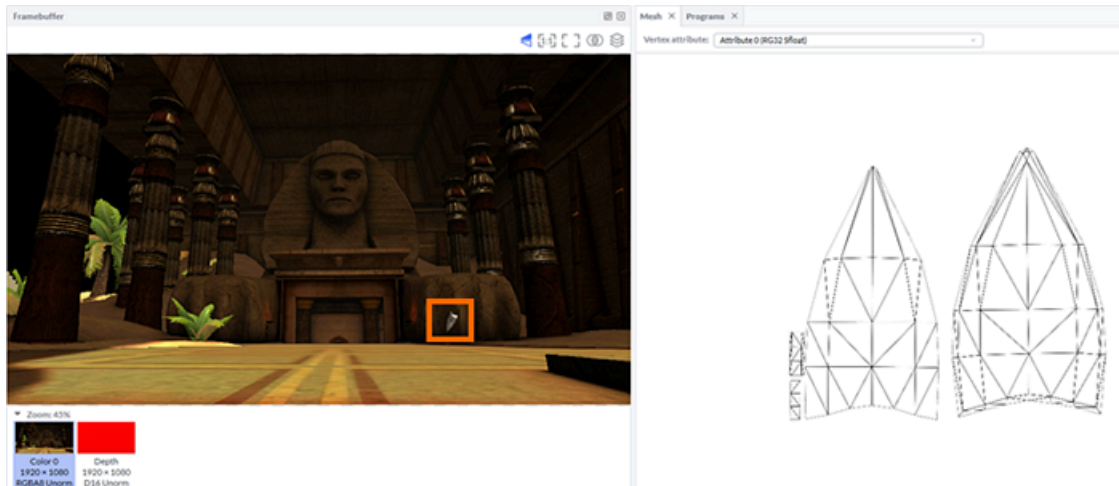
You can select different attributes in **Vertex Attribute** menu.

Use the mouse controls to move around the visualization:

- Drag to rotate
 - Shift+click, or hold middle mouse button, to pan
 - Scroll the wheel button to zoom
 - Right-click and select **Reset view** from the context menu to return the visualization to the default position
3. Look at the visualization in the **Mesh** view to see if all of the primitives are required. Check that the level of detail of the object is suitable based on its distance to the camera. Objects that have a high level of detail, and are further away from the camera, can negatively impact performance because of the [triangle density](#). Large numbers of very small triangles are expensive for the GPU to process, often with no visual benefit. Reduce the level of detail of objects as they move further away from the camera.

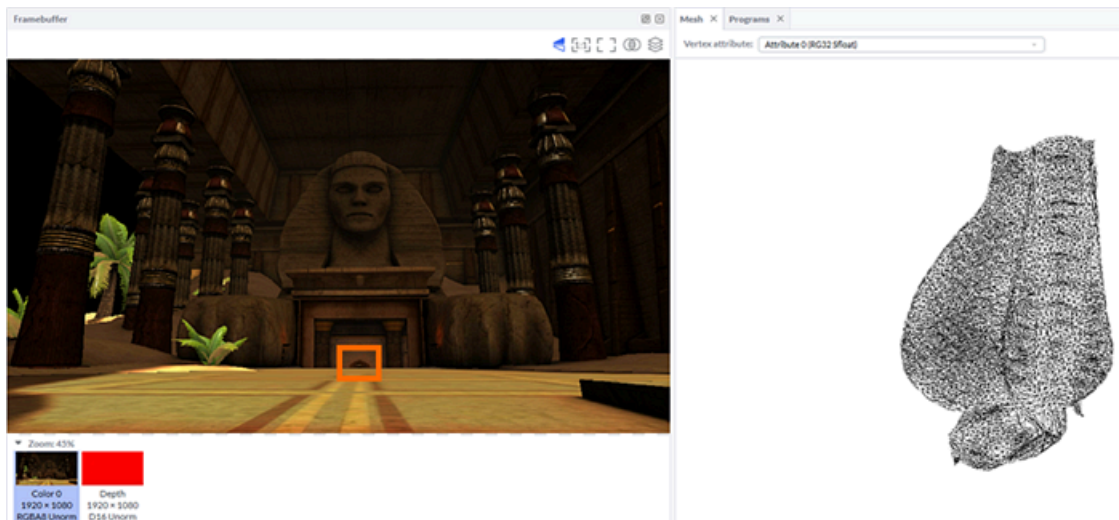
The following image shows an appropriate level of detail for the object, considering its distance to the camera.

Figure 3-18: An example of a good draw call



The following image shows an excessive level of detail for the object, considering its distance to the camera.

Figure 3-19: An example of a complex draw call



Remove unseen primitives. If your mesh has primitives that are never seen, they do not need to be part of the mesh. Processing occluded primitives wastes memory bandwidth, even if they are culled in a later step.

4. To see where the draw call appears in the **Framebuffer** view, use the arrow keys or mouse to step through the draw calls in the **Frame Hierarchy** view. If you cannot see the drawn mesh

appear in the **Framebuffer** view, it is possible that you do not need it. Consider culling the mesh before it is submitted to the GPU.



Merge draw calls that submit the same geometry. Arm® recommends drawing the same geometry in a single instanced draw call to improve efficiency.

5. Look at the **Detailed metrics** view to see a range of metrics about the mesh, including the number of primitives, how many vertices were actually shaded, and how much bandwidth the GPU used to read the mesh. See **Content metrics in detail** for more information.

Figure 3-20: Mesh complexity in Detailed metrics

Mesh complexity

Efficient meshes minimize the number of vertices and primitives, and the size of each vertex in memory.

- Vertices shaded: 45952
- Primitives: 22976
- Index size: 2 bytes
- Vertex size: 28 bytes
- Mesh bandwidth: 781184 bytes

Related information

[Arm GPU best practices](#)

3.6 Analyze model geometry

The **Content Metrics** view shows you data about all the frames, render passes, and draw calls in your trace. Filter and sort the metrics to help you identify opportunities to optimize your model geometry. From any draw call, you can navigate to the corresponding API call so that you can see exactly which model to modify.

About this task

In this example, we are looking at the data of a capture to see if there are any Vertex Shading Efficiency (VSE) issues.

Procedure

1. In the **Content metrics** view, select a captured frame.
2. Click the **Render passes** tab.
3. To help you find problems related to the model geometry, click a column heading to organize the data by size. For this example, click the **Draw calls** column heading until the render passes are sorted by the highest number of draw calls. Select the top row of the table that shows the render pass containing the highest number of draw calls.

Figure 3-21: Sorted draw calls column

Render Graph × API Calls × Content Metrics ×				
Frames Render passes Draws				
Frame 330 > Render pass 2 ×				
Frame	Render pass	Draw calls	Prims	Unique indices
330	2	175	110934	236672
330	0	154	110449	236055
330	8	11	40	80
330	1	1	2	4
330	3	1	2	4
330	4	1	2	4

- To see all the draw calls in this render pass, click the **Draws** tab.
- Now click the **VSE** column heading to show the draw call with the lowest **VSE** in the top row. A low VSE is caused by high average temporal locality, duplicate vertices, or low index sparseness efficiency metrics.
In this example, call **6491** has a **VSE** of **0.00**.

Figure 3-22: Sorted VSE column

Render Graph × API Calls × Content Metrics ×								
Frames Render passes Draws								
Frame 330 > Render pass 2 ×								
Frame	Render pass	Call	Prims	Unique indices	Vert size	VSE	VME	
330	2	6491	4834	9725	4	0.00	1.00	
330	2	6556	192	194	24	0.74	1.00	
330	2	6739	5	7	24	0.88	1.00	
330	2	5651	45	110	32	0.90	1.00	
330	2	5691	45	110	32	0.90	1.00	
330	2	5791	45	110	32	0.90	1.00	

- To find out what is causing the low VSE, open the **Detailed Metrics** view. Either right-click on the call in the table, then select **Navigate to call**, or double-click the call. The call is now also selected in the other views, and the **Detailed Metrics** view is populated with data about that call.
- The **Vertex shading efficiency** metrics section shows the metrics that contribute to a low VSE score. In this example, you can see that the number of **Duplicate vertices** in your model is high, nearly all the vertices are redundant duplicates, which are unnecessarily increasing processing cost and memory bandwidth. You can avoid most of the **Shaded vertices** in the model by reusing a vertex index rather than physically duplicating the vertex data.

Figure 3-23: Vertex shading efficiency

Vertex shading efficiency: 0.00	
Vertex Shading Efficiency (VSE) scores less than 1.0 indicate that proportionally more vertex shading is required relative to an optimal mesh. The more detailed metrics provide reasons why additional shading is required.	
• Index sparseness efficiency: 1.00	
• Duplicate vertices: 9681 (99.55%)	
• Temporal locality: 2.63 (StdDev 0.72)	

- In another example, a draw call has a low **VSE** of **0.08**. When exploring the **Vertex shading efficiency** metrics section of the **Detailed Metrics**, we can see that there are no **Duplicate vertices** in this part of the model, but the **Temporal locality** is very high.

Figure 3-24: Vertex shading efficiency**Mesh complexity**

Efficient meshes minimize the size of their workload, reducing the number of vertices and primitives, and the size of vertices in memory.

- Shaded vertices: 64404
- Primitives: 9990
- Index size: 2 bytes
- Vertex size: 36 bytes
- Mesh bandwidth: 247012 bytes

Mesh efficiency

Poor efficiency can reduce the performance of a workload relative to its potential peak performance.

Vertex shading efficiency: 0.08

Vertex Shading Efficiency (VSE) scores less than 1.0 indicate that proportionally more vertex shading is required relative to an optimal mesh. The more detailed metrics provide reasons why additional shading is required.

- Index sparseness efficiency: 1.00
- Duplicate vertices: 0 (0.00%)
- Temporal locality: 4089.29 (StdDev 4057.38)

This high number means that vertices are likely to be evicted from the post-transform cache before an index is reused, resulting in the reshading of vertices. The **Shaded vertices** metric shown in the **Mesh complexity** section confirms a high number of shaded vertices. To improve the temporal locality and avoid reshading, reorder the data to move reuses closer together in the index buffer.

3.7 Content metrics in detail

The **Detailed Metrics** view helps you to understand the cost of meshes and find opportunities to improve their efficiency.

To see detailed metrics about a draw call, right-click a row in the table on the **Draw** tab in the **Content metrics** section, then select **Navigate to call**.

3.7.1 Mesh complexity

Efficient meshes minimize the number of vertices and primitives, and the size of vertices in memory.

Shaded vertices

The number of vertices shaded by the GPU, factoring in any instancing, reshading, or over-shading effects.

Primitives

The total number of non-degenerate primitives in all instances.

Index size

The size of an index in bytes.

Vertex size

The actual size of a vertex in bytes.

Mesh bandwidth

An estimate of the GPU bandwidth used to read the mesh using its current memory layout.

3.7.2 Mesh efficiency

Poor efficiency can reduce the performance of a workload relative to its potential peak performance. These detailed efficiency metrics provide you with guidance on what you can change to improve performance.

3.7.2.1 Vertex shading efficiency

Vertex Shading Efficiency (VSE) scores less than 1.0 indicate that proportionally more vertex shading is required relative to an optimal mesh. The more detailed metrics provide reasons why additional shading is required.

Vertex shading efficiency

The ratio of vertex shader invocations to the number of used input vertices.

An efficiency of 1.0 indicates optimal shading efficiency, with 1 shader invocation per useful input vertex. Values less than 1.0 indicate that additional vertex shading is occurring. For example, an efficiency of 0.5 indicates 2 shader invocations per used input vertex.

Low shading efficiency is caused by several issues:

- Referenced indices that are shaded multiple times because of non-optimized temporal locality in the index buffer.
- Referenced indices that duplicate the data payload of other indices that are shaded.
- Non-referenced indices that are shaded because they are in the same group of 4 indices as a referenced index. Arm® GPUs always shade indices in groups of 4 consecutive indices, even if those indices are not used in the draw call.

To maximize vertex shading efficiency, improve the **index sparseness efficiency**, **duplicate vertices**, and **temporal locality** metrics.

Duplicate vertices

The number of vertices that have identical data to another vertex in the model.

Processing duplicate vertices wastes bandwidth because the system processes the same vertex more than once.

To reduce the number of duplicate vertices, use an index buffer to reference common vertex data, instead of uploading duplicated vertex data.

Temporal locality

The number of indices, mean and standard deviation, between reuse of an index value.

For example, in the sequence of vertices “0, 1, 2, 0, 5, 0”, the mean is 1.5 because the average number of vertices between the reused vertex of 0 is 1.5.

Arm® Frame Advisor models a 1024 entry post-transform cache for computing vertex reshading.

To improve cache efficiency of the post-transform cache during vertex shading, reduce the average temporal locality so that it does not exceed 1024 indices. To reduce your average temporal locality, reduce the number of indices between reuse of an index value.

3.7.2.2 Vertex memory efficiency

Vertex Memory Efficiency (VME) scores less than 1.0 indicate that proportionally more vertex memory bandwidth is required relative to an optimal mesh. The more detailed metrics provide reasons why additional bandwidth is required.

Some detailed metrics for VME are duplicates of the metrics for VSE, where the same pathology can impact both shading and memory bandwidth. These are documented in the [Vertex shading efficiency](#) section.

Vertex memory efficiency

The ratio of data that is fetched to how much is needed for processing this mesh.

An efficiency of 1.0 indicates optimal memory layout. Values less than 1.0 indicate that additional vertex data fetch is occurring. For example, an efficiency of 0.5 indicates that twice as much data is fetched because of a suboptimal memory layout.

Memory inefficiency is caused by:

- Padding between fields in a packed vertex.
- Padding between packed vertices.
- Attributes using an interleaved layout, that causes non-position data to load during position shading.

To maximize vertex memory efficiency, improve the **Mesh memory layout**, and optimize the [precision](#) and [layout](#) attributes as detailed in the [Arm GPU Best Practices Developer Guide](#).

Vertex padding size

The number of bytes of unused padding in each vertex, accounting for space between attributes or between vertices.

When fetching vertex attribute data from memory, the GPU also fetches any unused bytes between attributes or vertices. These padding bytes waste bandwidth.

To improve memory performance, ensure that vertex data is as tightly packed as possible.

Degenerate primitives

The number of primitives that have zero area.

Degenerate primitives occur when the vertices of a primitive form a zero area triangle, a common technique for encoding spatial jumps in a model. Processing degenerate primitives wastes bandwidth because those primitives are not displayed. A significant percentage of degenerate primitives might indicate a mesh encoding issue.

Remove as many degenerate primitives as possible from your model. If you use degenerate primitives for encoding spatial jumps in the mesh, use primitive restart instead.

Average spatial locality

The index difference, mean and standard deviation, between neighboring indices.

To improve memory access and cache efficiency during vertex shading, reduce the average spatial locality between neighboring indices. To reduce your average spatial locality, reduce the index difference between neighboring indices.

3.7.2.3 Mesh encoding efficiency

Efficient meshes reuse each vertex multiple times, sharing it across multiple primitives to reduce total vertex count.

Indices/primitive:

The number of unique indices per primitive, a measure of how well vertices are shared across multiple primitives.

Reusing a vertices for multiple primitives reduces the number of vertices needed to complete a model, reducing the overall size of mesh needed. Aim to get your **Indices/primitive** value as low as possible by sharing as many vertices as you can.

3.7.2.4 Attribute stream efficiency

Efficient meshes optimize their attribute stream memory layout to reduce the amount of redundant data fetched from main memory. Arm GPUs, and many other mobile GPUs, compute position and perform culling before running the remaining part of the vertex shader.

The recommended mesh memory layout splits the input attributes into two streams. One stream for the attributes needed to compute position, and one stream for the remaining attributes. This layout is more efficient for the GPU because it looks at data in each stream that is relevant to the process it is running. The ideal layout also repacks the attributes to remove any padding, unless it is required for type alignment.

Mesh bandwidth

The current memory bandwidth as a percentage of the ideal memory bandwidth. Values higher than 100% indicate additional data is being fetched from memory.

Ideal mesh bandwidth

An estimate of the GPU bandwidth that would be used with the recommended mesh layout.

3.8 Analyze function calls

The **API Calls** view shows you every call that was made for all the frames that are in your trace. You can search the data to help you identify opportunities for optimization or find errors in your API calls. See return values for applicable calls, and find out if the CPU is the limiting factor for a call.

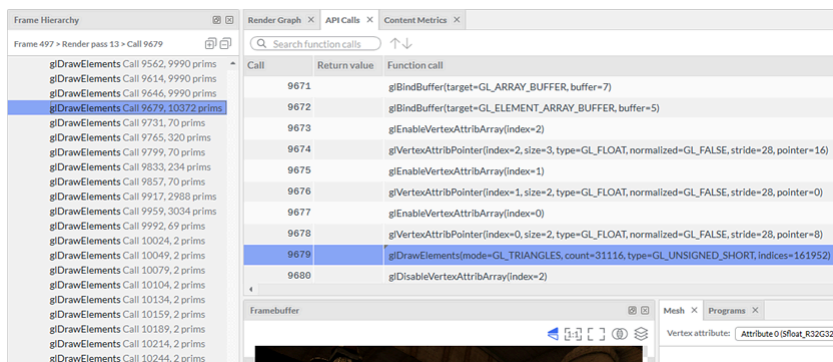
About this task

Here is an example of how you can use the **API Calls** view to identify an issue with function calls in your captured frame.

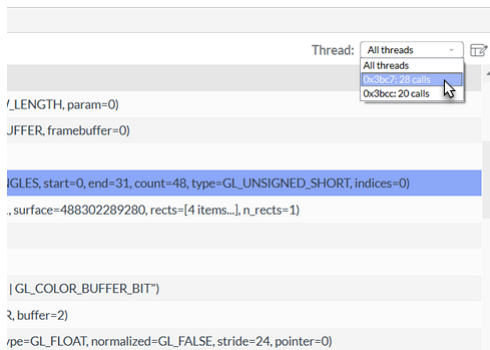
Procedure

1. See how draw calls render in the **Framebuffer** view. Select a draw call in the **Frame Hierarchy** view, and use either the arrow keys or your mouse to step through the draw calls. You can see that the draw call selected in the **Frame Hierarchy** view is also selected in the **API Calls** view.

Figure 3-25: Draw calls selected



2. If your trace uses multiple threads, choose the appropriate thread from the drop-down menu. If your trace uses one thread only, the drop-down menu is not visible.

Figure 3-26: See function calls for a specific thread


3. To show different data for your function calls, click the **Edit columns** icon  and select the data that you want to see from the menu. For example, you can show errors that are associated with a function call, or show unmodeled function calls.
4. In the **API Calls** view, look for any values that are not correct for that draw call. To help you find inefficient uses of the API, enter regex terms in the search box. For example, you can search for opaque draws that have blending enabled, or if back-face culling is disabled.

Figure 3-27: Search function calls

Render Graph × API Calls × Content Metrics ×		
<input type="text" value="back"/> 348 ↑↓ <input checked="" type="checkbox"/> Filter results		
Call	Return value	Function call
27		glCullFace(mode=GL_BACK)
52		glCullFace(mode=GL_BACK)
110		glCullFace(mode=GL_BACK)
139		glCullFace(mode=GL_BACK)
168		glCullFace(mode=GL_BACK)
198		glCullFace(mode=GL_BACK)
227		glCullFace(mode=GL_BACK)
255		glCullFace(mode=GL_BACK)

The function calls are filtered using the regex term. If you would prefer to see the function calls in call order, clear the **Filter results** checkbox.

Next steps

If you see an error, or an unexpected return value or function call, review the code in your application.

3.9 Analyze shader programs

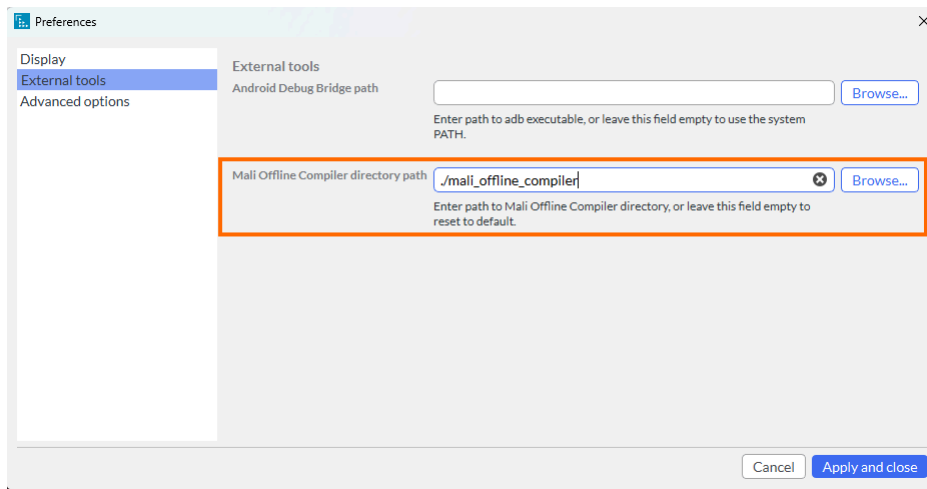
Arm® Frame Advisor reports information about the shader programs used in your application. Follow these steps to discover how your shaders were used in the captured frame. Explore which shaders are in use for each frame, render pass or draw call, find the most expensive shaders and look for best practice violations.

Before you begin

For MacOS only, before capturing a trace, you must set the path to the Mali Offline Compiler directory so that Frame Advisor can generate and display metrics for shaders in the **Programs** view:

1. In the Frame Advisor menu, select **Configure -> Open preferences -> External tools**.
2. For **Mali Offline Compiler directory path**, either enter the path to the Mali Offline Compiler directory or click **Browse** to find the directory. This directory is normally located in the Arm Performance Studio install directory.

Figure 3-28: Frame Advisor Preferences



Procedure

1. [Capture a frame burst](#) for a frame that you want to analyze.
2. Select a frame or render pass, or step through the draws in the **Frame Hierarchy** view. Look at the **Programs** view to see which shaders are used in the currently selected object, and review their performance characteristics in the table of metrics. Use the filter buttons to show different shader stages.

Figure 3-29: Frame Advisor Programs View

Mesh X Programs X																	
		Filter by render pass															
		Stage: Vert position Vert main Fragment															
Program	Shortest path				Longest path				Total emitted				Occupancy	Work regs	Uniform regs	Stack spills	FP16 usage
	A	LS	T	Max	A	LS	T	Max	A	LS	T	Max					
28	1.00	7.00	0.00	7.00	1.00	7.00	0.00	7.00	1.00	7.00	0.00	7.00	100	19	40	0	0
27	0.00	3.00	0.00	3.00	0.00	3.00	0.00	3.00	0.00	3.00	0.00	3.00	100	7	24	0	100
3	0.00	4.00	0.00	4.00	0.00	4.00	0.00	4.00	0.00	4.00	0.00	4.00	100	9	12	0	0



For Vulkan applications the metrics are generated without using the application pipeline state, so the numbers might not exactly match the runtime driver.

3. The **Programs** view table includes an approximate cycle cost breakdown for the major functional units in the shader core, the arithmetic unit, the load/store unit, the varying unit, and the texture unit. Sort the table by highest cost to find the shaders that are the best candidates for optimization. Look for the functional unit with the highest cycle cost in either or both of the shortest and longest path cycles. Consider how you could optimize the shader to reduce cost for that functional unit first.

Shortest path

An estimate of the number of cycles for the shortest control flow path through the shader program.

Longest path

An estimate of the number of cycles for the longest control flow path through the shader program.

Total emitted

The cumulative number of cycles for all instructions that are generated for the program, irrespective of program control flow.



The table shows the highest cost per thread. However, Frame Advisor does not know how many threads were executed in total, therefore this number does not represent the total shader cost per frame.

4. Sort the table by shaders with the most **Stack spills**. Shaders that spill to stack are expensive for a GPU to process, so try to reduce register pressure by:
 - a. Reducing variable precision.
 - b. Reducing the live ranges of variables.
 - c. Simplifying the shader program.
5. The **FP16 usage** column reports the percentage of arithmetic operations that are performed at 16-bit precision or lower for each shader program. A higher number here is better, because 16-bit precision is twice as fast as 32-bit precision. Sort the table by the lowest use of FP16 to find shaders to optimize by reducing precision. Reducing precision also reduces both energy consumption and register pressure, and can double the performance. There are situations where highp is always required, such as for position and depth calculations. However, in many cases there is little noticeable difference on-screen when reducing precision to 16-bit.

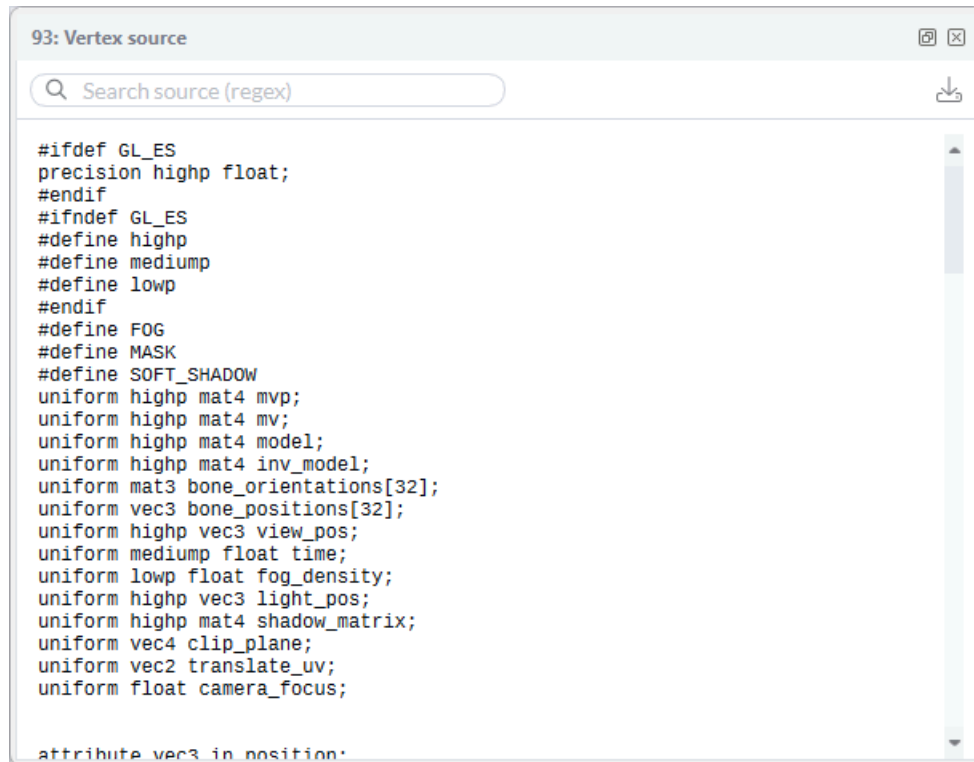
For OpenGL ES applications, set precision to `mediump`. For Vulkan, use `RelaxedPrecision`. Alternatively, use explicit 16-bit type extensions to set precision.

6. Work registers are general purpose read-write registers that are allocated to each running thread. Sort the table by the highest number of work registers in use, and look for shaders using 32 or more.

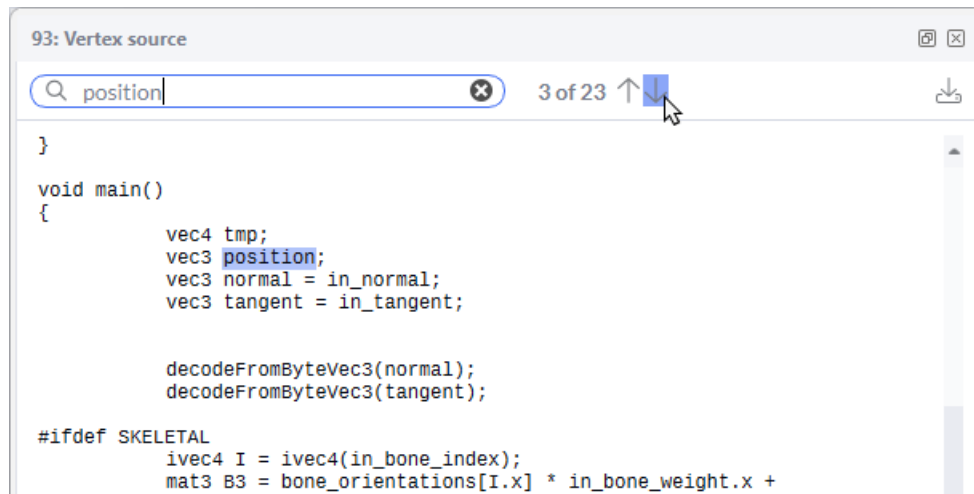
The available physical register pool is divided among the shader threads that are executing. Therefore, reducing work register usage can increase the number of threads that can be executed simultaneously, helping to keep the GPU busy. Aim to reduce work register usage to below 32 to improve occupancy. Try reducing precision from highp (32-bit) to mediump (16-bit), to enable the GPU to store twice as many variables per register.

7. Uniform registers are read-only registers that are allocated to each running program. Uniform registers are used to store uniform and literal constant values. Shaders that run out of uniform storage need to fall back to per-thread memory loads for additional values. Look for shaders that use the highest numbers of uniform registers and aim to reduce them by using 16-bit data types, or by reducing the number of uniforms and constants in the shader program.
8. To view the source code for a shader program, double-click anywhere along a row to open the **Source** view.

Figure 3-30: Source view



9. In the **Source** view, you can search your shader code, or download it to an external file.

Figure 3-31: Source view

Next steps

- See the Arm GPU best practices guide for information on [vertex shading](#) and [fragment shading](#)
- To analyze a single shader program and how it would perform on any Arm GPU, use [Mali Offline Compiler](#).

4. How to get help

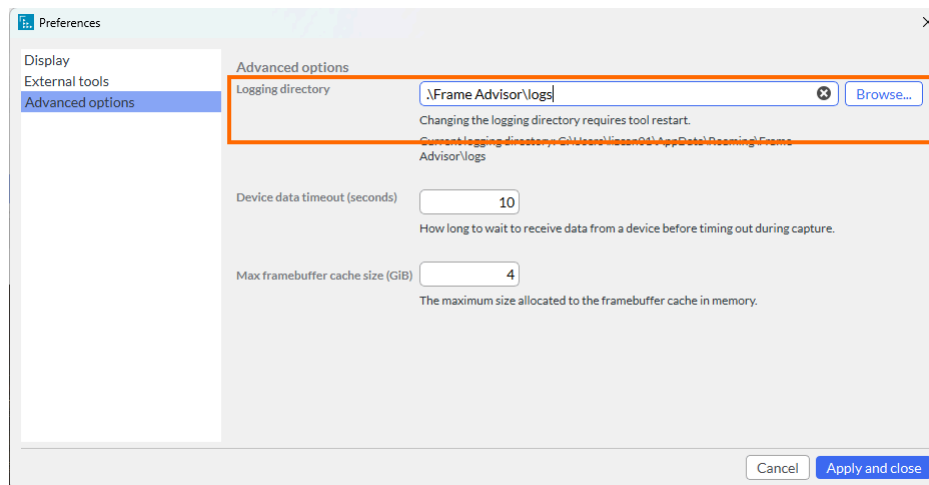
Arm® provides many resources for you to find more information about Frame Advisor or other Arm Performance Studio tools. You can also engage with other users, or ask us a question directly.

Here are the various ways for you to find more information or to get in touch:

- See the latest discussions, learn from experienced users, or ask a question in the [Graphics, Gaming, and VR community forum](#).
- Find information and resources for Arm Performance Studio on the [Developer website](#).
- To ask a question directly, you can email the Arm Performance Studio team at performancestudio@arm.com.
- To give feedback about Frame Advisor, fill in this [feedback form](#). You can also access this form in Frame Advisor. Select **Help > Send feedback**.

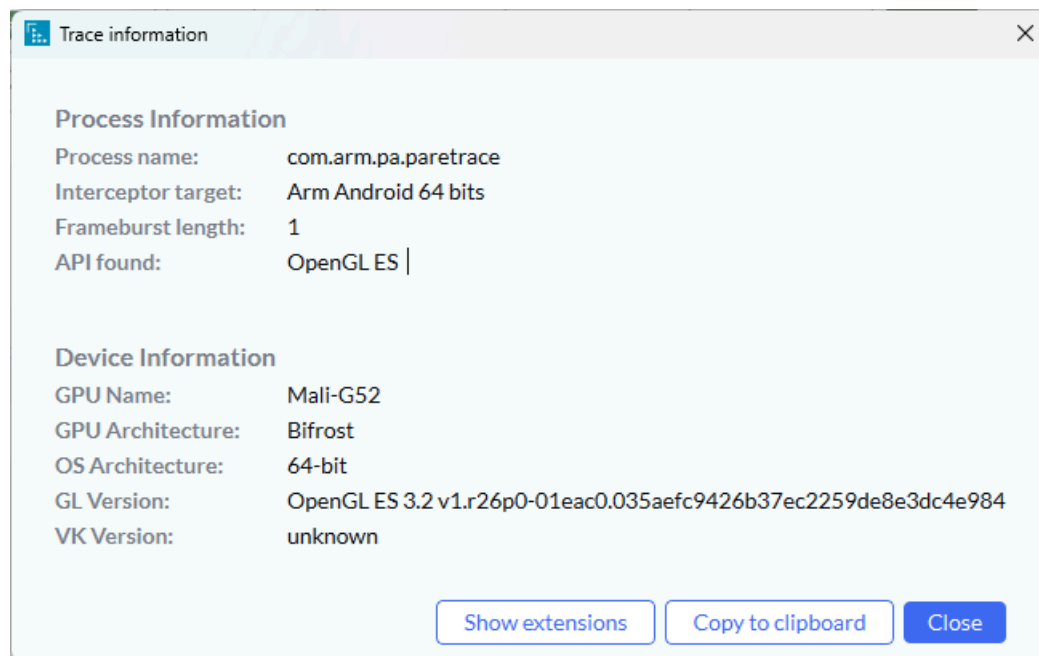
Your activity data in Frame Advisor is saved to a log file in your user directory. Arm recommends that you include the log file when you contact the Support team for help with unexpected issues. To change the path to your Frame Advisor log file, click **Configure -> Open preferences -> Advanced options** in the Frame Advisor menu.

Figure 4-1: Frame Advisor Preferences



See information about your trace

After you have captured a trace, you can see information about your trace, API, and the device it was captured on, in the **Trace information** dialog. To open the **Trace information** dialog, click **Help -> Trace information** in the Frame Advisor menu.

Figure 4-2: Trace information dialog

This information is useful when contacting the Arm Performance Studio team for support. Copy the information to the clipboard, where you can then paste it into a support message for the team. For OpenGL ES traces, you can also show the API extensions and filter for a specific extension used in the trace.

5. Troubleshooting Frame Advisor

Find answers to common problems that might occur when capturing or analyzing data in Arm® Frame Advisor.

5.1 My device is not listed in Frame Advisor

When starting a new trace, my device is not listed on the connection screen.

You may not have set up your computer and device correctly

The [Setup tasks](#) explain how to set up your computer and device to use Arm® Frame Advisor.

Solution

1. Check that the device is connected to your computer via USB.
2. Check that the device is set to [Developer mode](#).
3. Check that the device has USB debugging enabled in Settings > Developer options. When enabling USB debugging, your device may ask you to authorize connection to your computer. Confirm this.
4. Ensure you have installed Android Debug Bridge (adb).
5. In a shell terminal, run the `adb devices` command. This command returns the ID of all connected devices. For example, with one device connected:

```
adb devices
List of devices attached
RZ8MC03VVEW device
```

If the device is listed as unauthorized, this means that your device has USB debugging enabled, but the computer it is connected to has not been given authority to access it.

6. Go to Settings > Developer Options, then disable and re-enable USB debugging. You can also try revoking USB debugging authorizations here. When re-enabling USB debugging, your device asks you to authorize access from your computer.

5.2 My application is not listed in Frame Advisor

When starting a new trace, my application is not listed on the connection screen.

You may not have set up your application correctly

Ensure that adb is authorized, and that your application is built properly.

Solution

1. In a shell terminal, run the `adb devices` command. This command returns the ID of all connected devices. For example, with one device connected:

```
adb devices
List of devices attached
RZ8MC03VVEW device
```

If the device is listed as unauthorized, this means that your device has USB debugging enabled, but the computer it is connected to has not been given authority to access it.

2. Check the Android manifest file to ensure your application is set to debuggable.
3. Check that the activity is marked as main when you build your application in the Android manifest.

5.3 The Framebuffer view is slow to load images

When you click through the draw calls in your captured frame, the **Framebuffer** view is slow to load images.

Increase your Framebuffer cache

Arm® Frame Advisor stores recently accessed framebuffer images. To enable Frame Advisor to store more images, increase the size of the cache so that the images load faster in the **Framebuffer** view when they are accessed again.

Solution

1. To open the **Preferences** dialog box, click **Configure -> Open preferences -> Advanced options**.
2. Increase the **Max framebuffer cache size (GiB)**, then click **Apply and close**.



Note

Do not allocate all of your available memory in the **Max framebuffer cache size (GiB)**. Using too much of your available memory in the framebuffer cache can adversely affect the performance of your system.

3. Restart Frame Advisor.
4. Click some draw calls in the **Frame Hierarchy** view. The images in the **Framebuffer** view load faster.

5.4 A timed out error message appears when I start a capture

After you click the **Capture** button, the *Timed out waiting for a response from the daemon.* message is displayed.

Your application is not sending API calls

Arm® Frame Advisor shows this message when your application is not sending API calls. If you think that your application is sending API calls, but the timeout is happening too soon, you can increase the time that Frame Advisor checks for API calls.

Solution

1. In Frame Advisor, open the **Preferences** dialog box. Click **Configure -> Open preferences -> Advanced options**.
2. For **Device data timeout (seconds)**, increase the number of seconds for Frame Advisor to perform the API calls check. Then click **Apply and close**.
3. Restart Frame Advisor.
4. Capture a new trace.

If the timed out message is displayed again:

- Check your code for any obvious problems.
- If you have checked your code, see [How to get help](#).

5.5 An unsupported image format message is displayed in the Framebuffer

As you click through draw calls in the **Framebuffer** view, the 'Unsupported image format' message is displayed instead of the visualized output.

The image format used cannot be interpreted

Arm® Frame Advisor shows this message when the version of Frame Advisor that you are using does not support the image formats used.

Solution

For more information about supported image formats, email the Arm Performance Studio team at performancestudio@arm.com. Include your log file in the email so that the team can check which image formats you are using. To find where your log file is saved, click **Configure -> Open preferences -> Advanced options** in the Frame Advisor menu.

5.6 A no image data message is displayed in the Framebuffer

As you click through draw calls in the **Framebuffer** view, the 'No image data' message is displayed instead of the visualized output.

The image data cannot be interpreted

Arm® Frame Advisor shows this message when it cannot interpret the image data.

Solution

To help the team to identify the problem with your image data, email your Frame Advisor log file to the Arm Performance Studio team at performancestudio@arm.com. To find where your log file is saved, click **Configure** -> **Open preferences** -> **Advanced options** in the Frame Advisor menu.

5.7 Capturing frames from an Unreal Engine application ends unexpectedly

When capturing frames from an Unreal Engine application, the capture session pauses for a long time, and then fails with a **Capture terminated** error message.

Frame Advisor cannot access the shader source because PSO caching is enabled

When PSO caching is enabled, Unreal Engine can run multiple processes to compile the shader objects. The main thread interceptor in Frame Advisor cannot access the shader source because it is on another thread or interceptor.

Solution

Before you use Frame Advisor with an Unreal Engine application, you must disable PSO caching as described in [Setup tasks](#).

Proprietary Notice

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

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

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

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

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

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0108-00	24 July 2025	Non-Confidential	New document for v1.8
0107-00	17 April 2025	Non-Confidential	New document for v1.7
0106-00	6 February 2025	Non-Confidential	New document for v1.6
0105-00	28 November 2024	Non-Confidential	New document for v1.5
0104-00	5 September 2024	Non-Confidential	New document for v1.4
0103-00	7 June 2024	Non-Confidential	New document for v1.3
0102-00	18 April 2024	Non-Confidential	New document for v1.2
0101-00	15 February 2024	Non-Confidential	New document for v1.1
0100-00	21 November 2023	Non-Confidential	New document for v1.0

Change history

For information about the functional changes to Frame Advisor, see the [Arm® Performance Studio Release Notes](#).

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



Caution

We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.

Arm product resources	Document ID	Confidentiality
Android performance triage with Streamline Tutorial	102540	Non-Confidential
Arm® GPU Best Practices Developer Guide	101897	Non-Confidential
Arm® Mali GPU Training	Arm® Mali GPU Training	Non-Confidential
Arm® Performance Studio Release Notes	107649	Non-Confidential
Understanding Render Passes	102479	Non-Confidential

Non-Arm resources	Document ID	Organization
Android Debug Bridge (adb)	Android Debug Bridge adb	Android Developers
Android Studio	Android Studio	Android Developers
Configure on-device developer options	Configure on-device developer options	Android Developers
Specification for intent arguments	Specification for intent arguments	Android Developers