



Arm[®] Frame Advisor

Version 1.3

User Guide

Non-Confidential

Copyright © 2023–2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 00

102693_0103_00_en



Arm® Frame Advisor

User Guide

Copyright © 2023–2024 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
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

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. 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

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

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>.

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.

Contents

1. Introduction.....	7
1.1 Conventions.....	7
1.2 Useful resources.....	8
1.3 Other information.....	9
2. Overview of Frame Advisor.....	10
2.1 Platform support.....	13
3. Get started with Frame Advisor.....	14
3.1 Setup tasks.....	14
3.2 Capture a frame burst.....	16
3.3 Analyze the results.....	21
4. Analyzing your frames.....	29
4.1 Navigating your frames.....	29
4.2 Analyze workloads using the Render Graph.....	31
4.3 Analyze object rendering.....	33
4.4 Analyze object complexity.....	38
4.5 Analyze model geometry.....	41
4.6 Content metrics in detail.....	43
4.6.1 Mesh complexity.....	44
4.6.2 Mesh locality.....	44
4.6.3 Mesh redundancy.....	46
4.6.4 Mesh memory layout.....	47
4.7 Analyze function calls.....	47
5. How to get help.....	49
6. Troubleshooting Frame Advisor.....	50
6.1 My device is not listed in Frame Advisor.....	50
6.2 My application is not listed in Frame Advisor.....	50
6.3 The Framebuffers view is slow to load images.....	51
6.4 A timed out error message appears when I start a capture.....	52
6.5 An unsupported image format message is displayed in the Framebuffer.....	52

6.6 A no image data message is displayed in the Framebuffer.....	53
--	----

1. Introduction

1.1 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.

1.2 Useful resources

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

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

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
Understanding Render Passes	102479	Non-Confidential

Non-Arm resources	Document ID	Organization
<i>Android Debug Bridge (adb)</i>	Android Debug Bridge adb	Android Developers
<i>Android Studio</i>	Android Studio	Android Developers
<i>Configure on-device developer options</i>	Configure on-device developer options	Android Developers

1.3 Other information

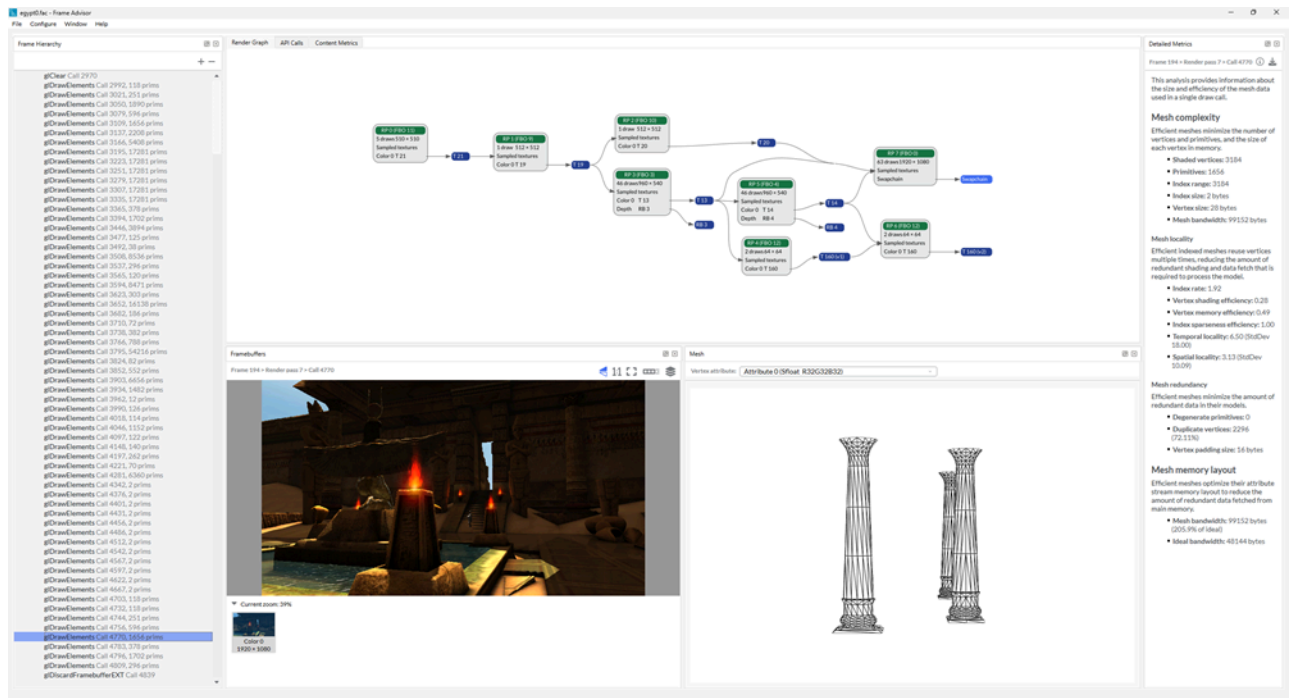
See the Arm website for other relevant information.

- [Arm® Developer](#).
- [Arm® Documentation](#).
- [Technical Support](#).
- [Arm® Glossary](#).

2. 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, save processing power, and improve use of memory bandwidth.

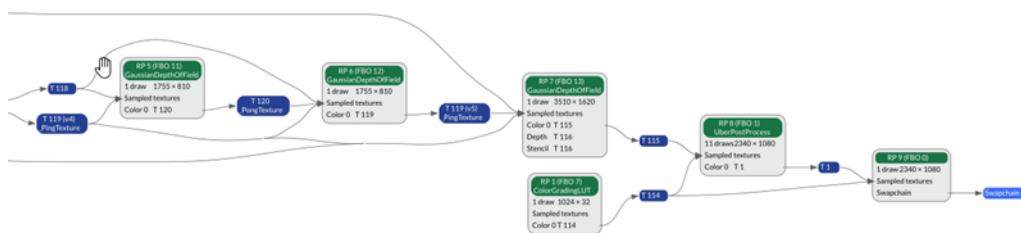
Figure 2-1: The Frame Advisor Analysis screen



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 are processed by GPU workloads, the rendering commands in each workload, and the data flows between them. This information helps you to optimize render pass processing and memory bandwidth, both of which are essential for best performance using a tile-based Arm® GPU.

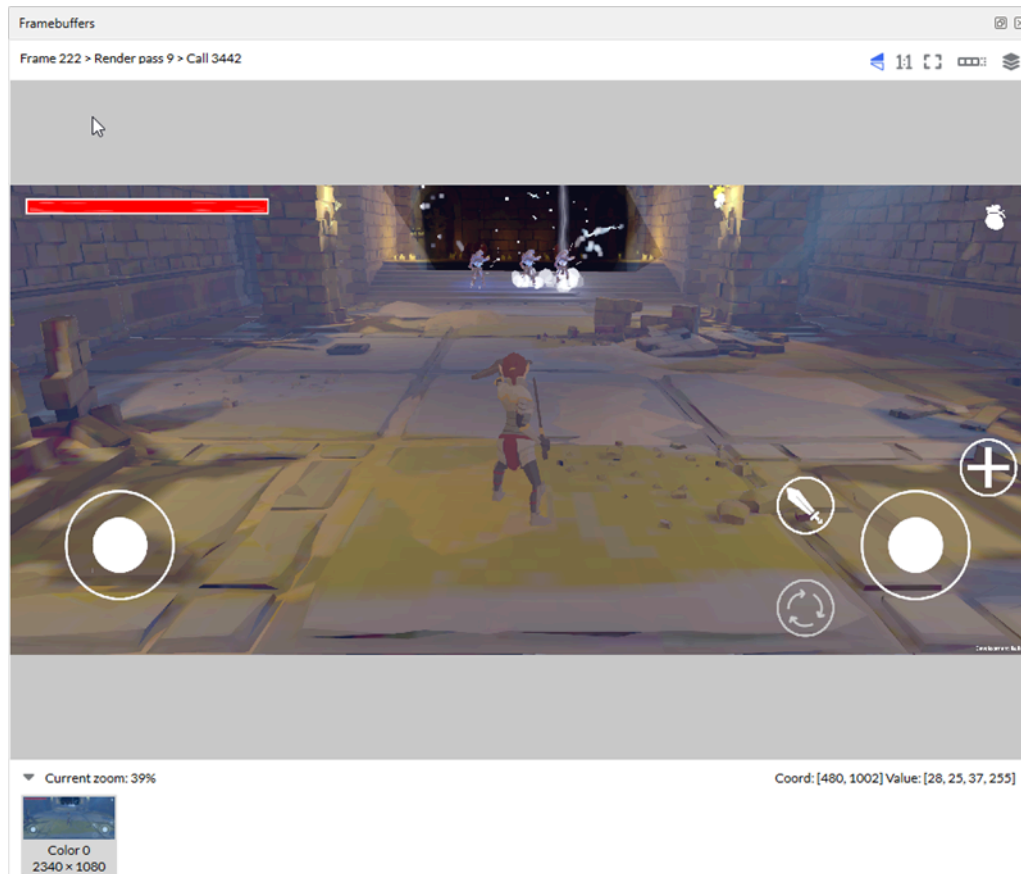
Figure 2-2: Render Graph



See the framebuffer output

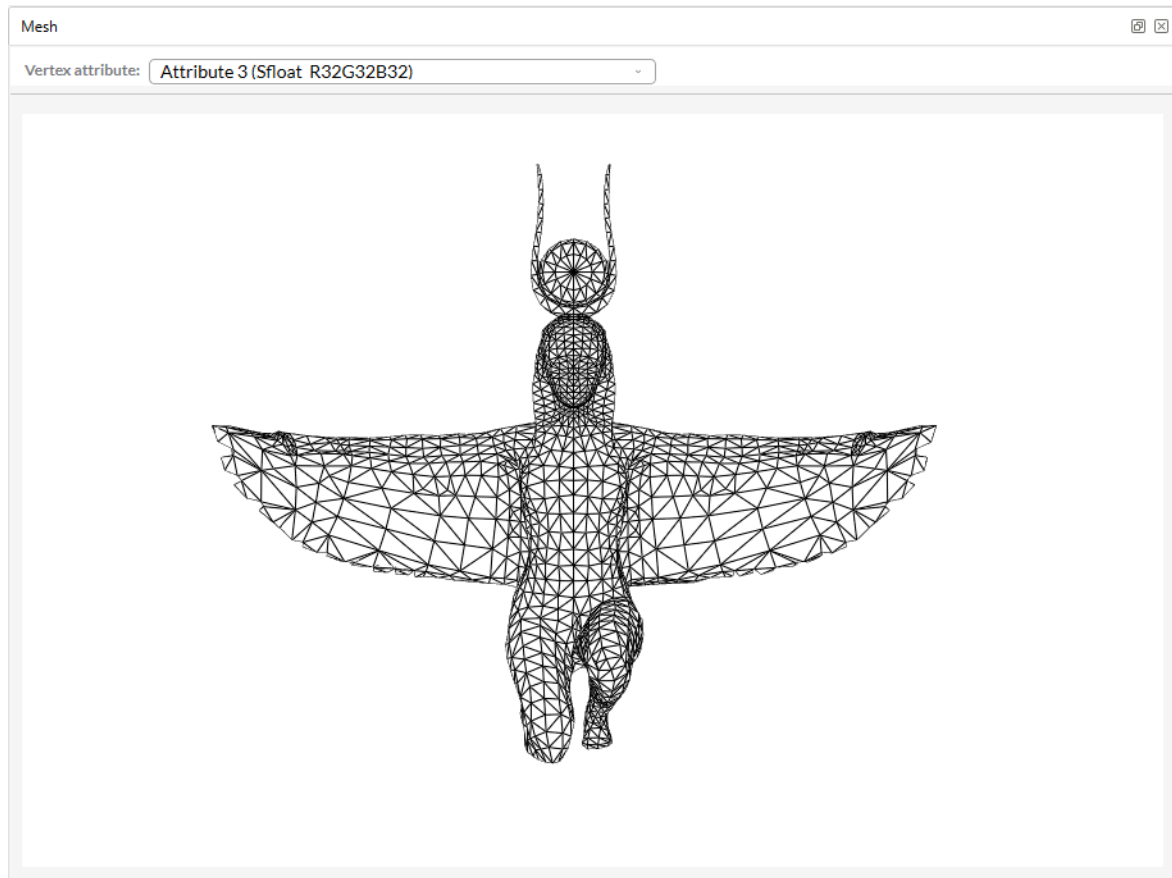
See what the GPU rendered to the framebuffer for each draw call. Step through the draw calls and watch the visual output to see how your application constructed each frame. Identify draws are rendering in the correct order, or any draws that render no visual output. When used with capture modes, such as overdraw mode, a false-color visualization of useful key performance metrics is shown.

Figure 2-3: Framebuffers view



Inspect rendered object complexity

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

Figure 2-4: Mesh view

Evaluate your model geometry

Use the content metrics provided in Frame Advisor together 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 content metrics to find draw calls that score poorly on one of the built-in efficiency metrics that Frame Advisor provides.

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.

Explore OpenGL ES or Vulkan API calls

See what functions and parameters the application requested 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.

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](#).

2.1 Platform support

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

Host support

Frame Advisor supports the following host OS versions:

- Windows 10 or later
- Ubuntu 20.04 or later
- macOS 10.15 (Catalina) or later

Device support

Frame Advisor supports the following device OS versions:

- OpenGL ES: Android 10 or later
- Vulkan: Android 9 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.

3. 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.

3.1 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. For OpenGL ES applications, your device must be running Android 10 or later. For Vulkan applications, your device must be running Android 9 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.

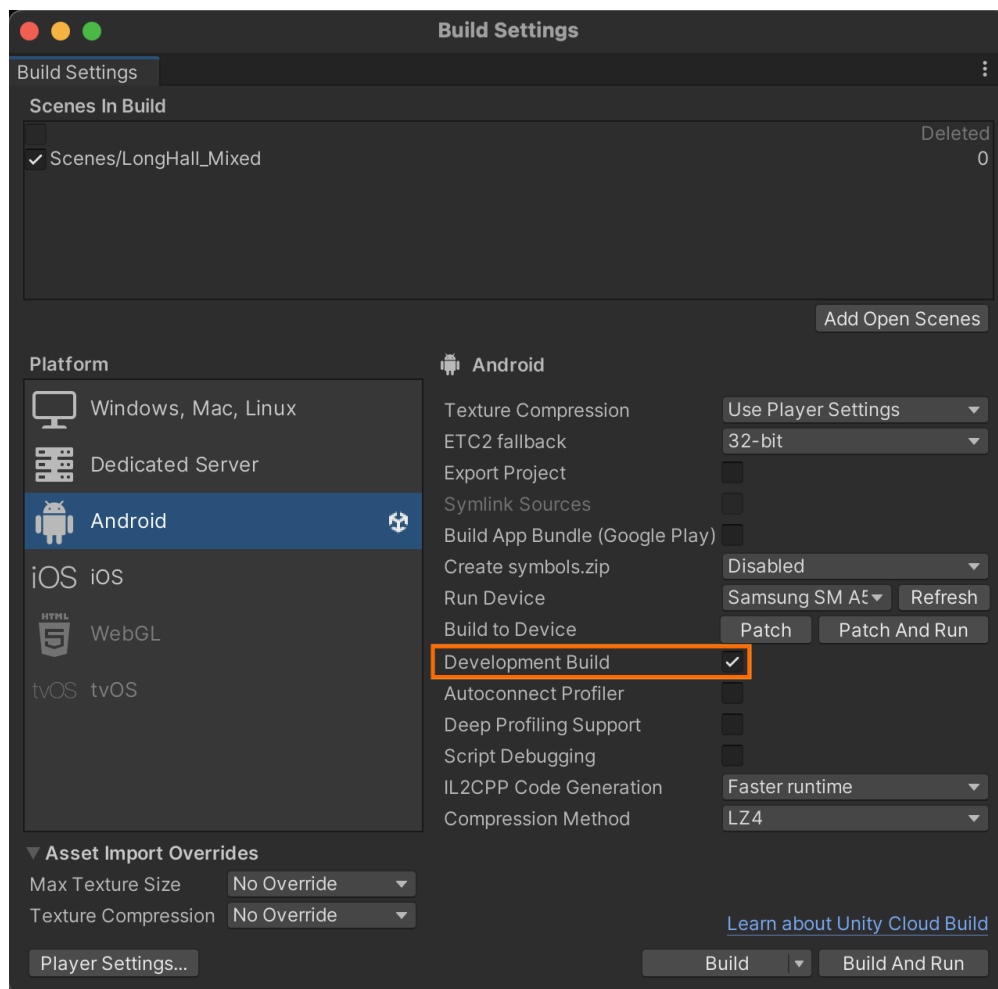
```
adb devices
List of devices attached
ce12345abcdef1a1234    device
```

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.

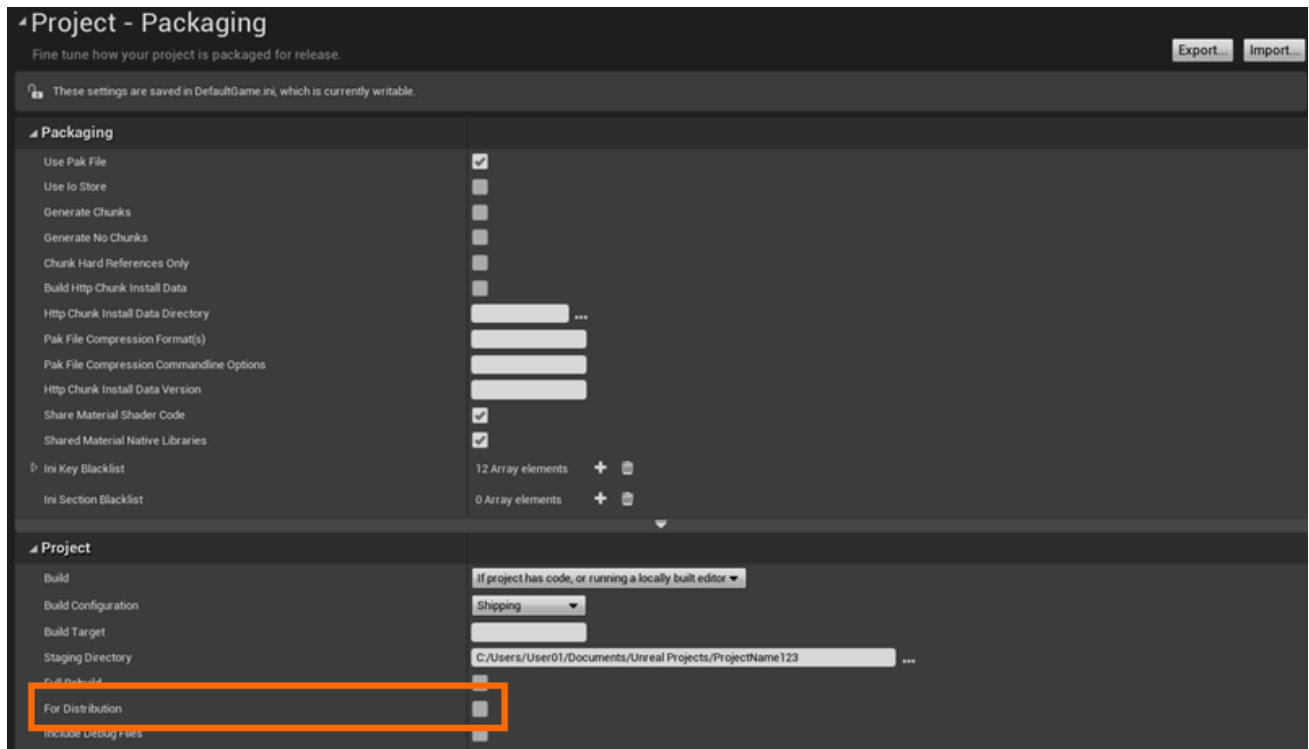
4. Disable permission monitoring. If your phone has **Settings > Developer options > Disable permission monitoring**, ensure that **Disable permission monitoring** is selected. If you do not have the option to disable permission monitoring, you can ignore this step.
5. Install a debuggable build of the 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 3-1: Unity Build Settings



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

Figure 3-2: Unreal Engine Build Settings

Next steps

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

3.2 Capture a frame burst

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

Procedure

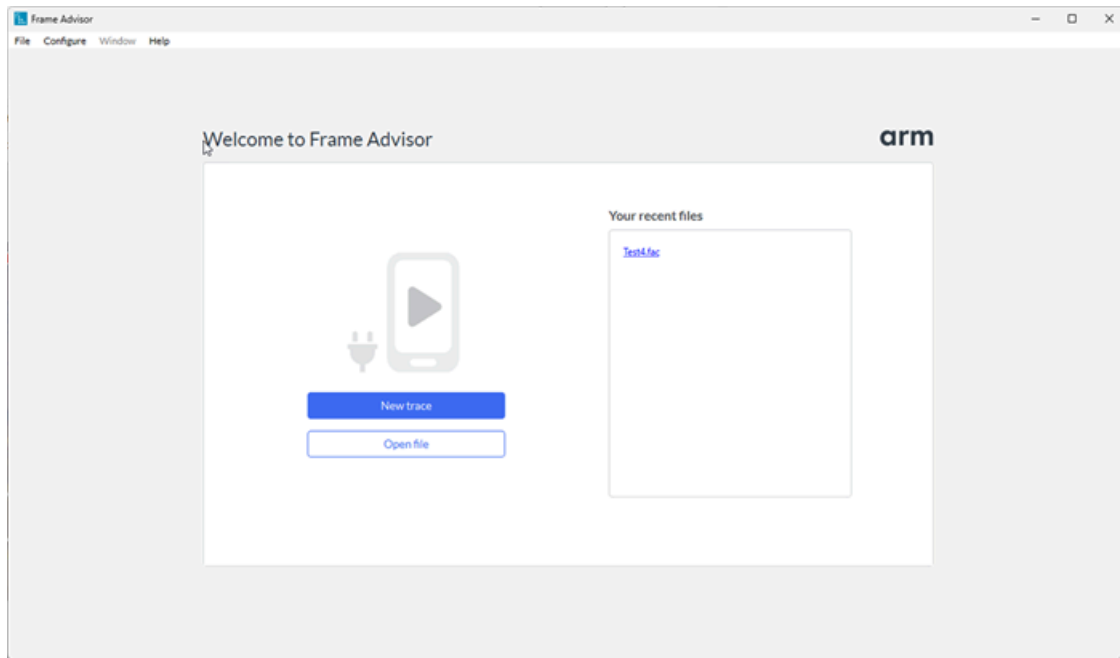
1. Open Frame Advisor:

- On Windows, from the **Start** menu, navigate to **Arm Performance Studio <version>** and select **Arm Frame Advisor <version>**.
- 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
```


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

Figure 3-3: Frame Advisor launch screen



Select **New trace** to start a new trace.

3. Frame Advisor lists the connected devices and the applications installed.

Figure 3-4: 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

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

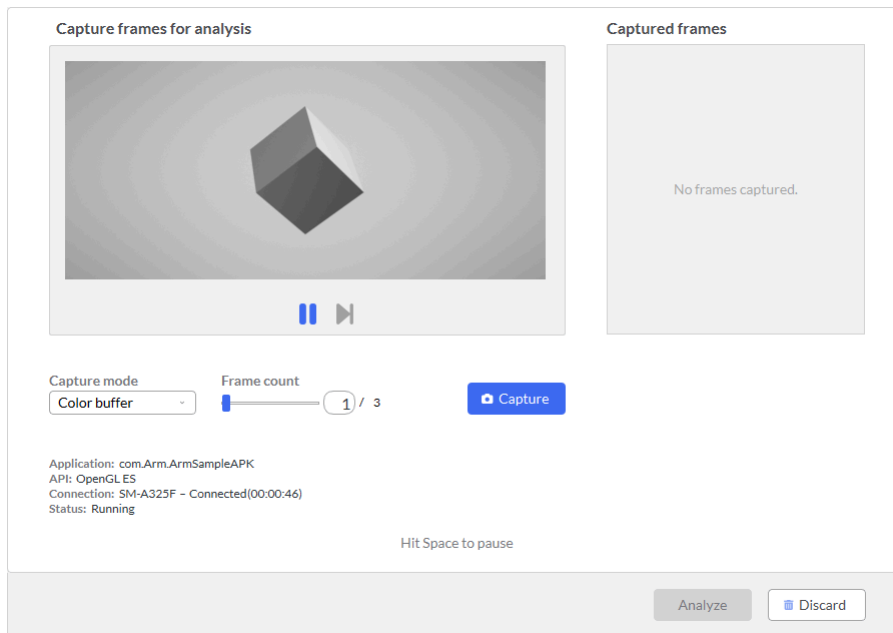
If your application uses the Vulkan API, change the selection in the API settings to **Vulkan**.

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.

- The **Capture** screen provides options for your capture session.

Figure 3-5: 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 that is what you are interested in. You can also adjust the **Frame count** to the number of frames that you want to capture.

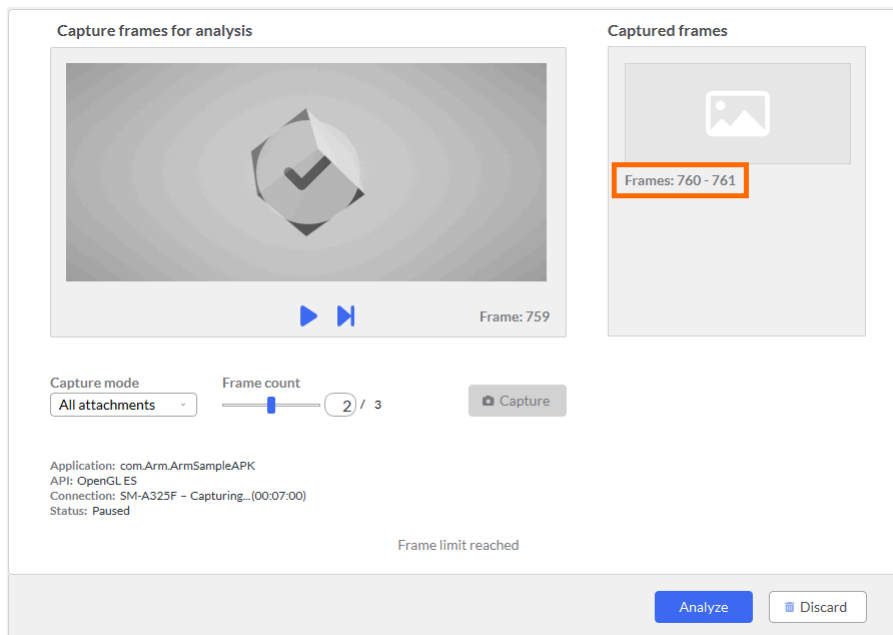


Note

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.

5. When you get to the part of your game 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.
6. 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 3-6: Captured frame numbers

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

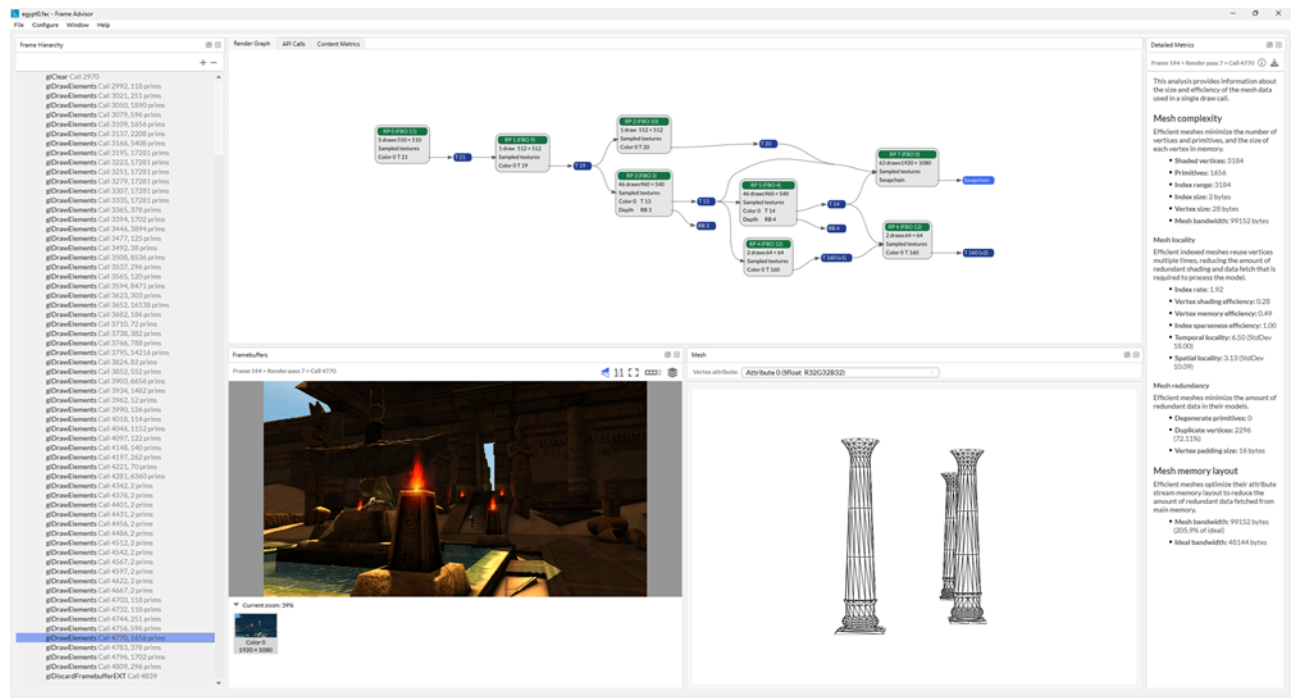
Next steps

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

3.3 Analyze the results

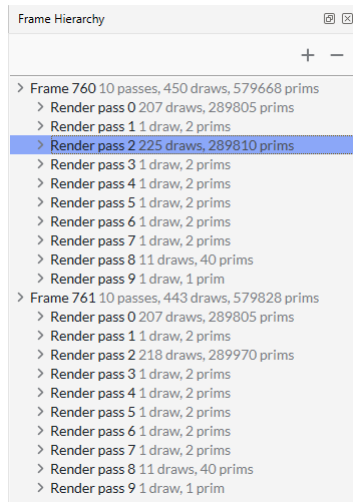
When you have captured some frames, click **Analyze**. Arm® Frame Advisor processes the data and presents it in the **Analysis** screen. Explore each frame to evaluate how efficiently they were rendered on the device.

Figure 3-7: Example Analysis screen



Frame Hierarchy view

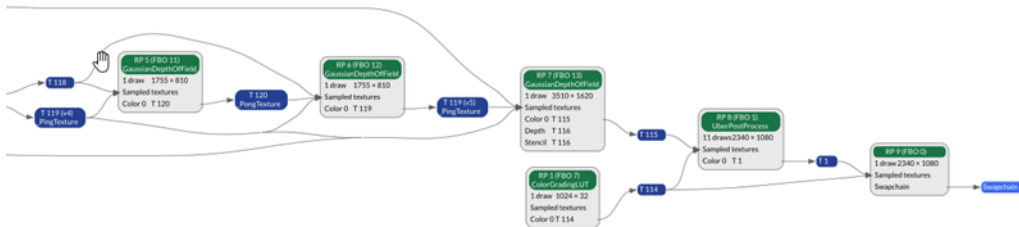
The **Frame Hierarchy** view lists the captured frames in a tree structure. Expand a frame to view the render passes and draw calls within it. Select an item in the hierarchy to navigate to it, and to show its output data for the item in other views of the **Analysis** screen.

Figure 3-8: Frame Hierarchy view

Contextual information helps you to find frames that need further investigation, such as those with the highest number of draws or primitives. See [Navigating your frames](#).

Render Graph view

The **Render Graph** shows an overview of the processing operations that are performed to create the final rendered frame. See the data flow between render passes in the frame, and how resources, such as textures, are produced and consumed.

Figure 3-9: Render Graph

Data is processed 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. Render passes are the main processing blocks in the rendering pipeline, which are identified when either:

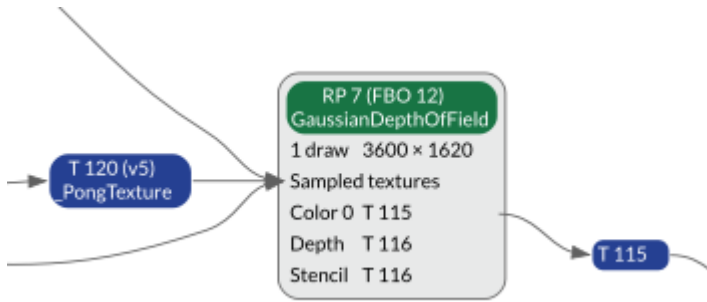
- An API call changes the framebuffer, or when queued rendering is flushed in the OpenGL ES API.
- A `VkRenderPass` is specified in the Vulkan API.

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

- Labels that show the render pass name, API name and user label
- Draw count and resolution

- Output attachments

Figure 3-10: Render graph label

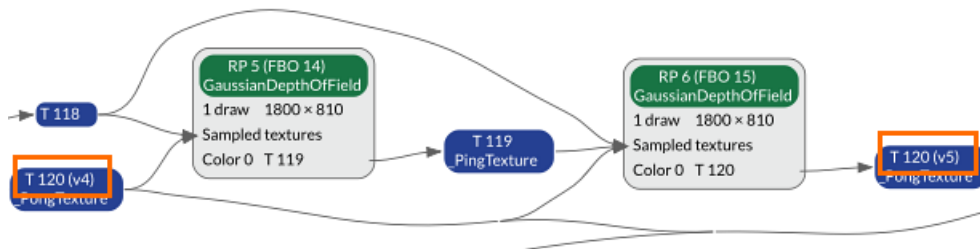


Short form names are used on the label:

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

Render passes consume resources and input attachments to render an output image to a set of attachments. Resources, represented by the dark blue nodes, are textures and renderbuffers that are consumed and modified by render passes. When a resource is modified by a renderpass, a version number, which increments each time the resource is modified is displayed on the label. Resources that are not modified do not show a version number.

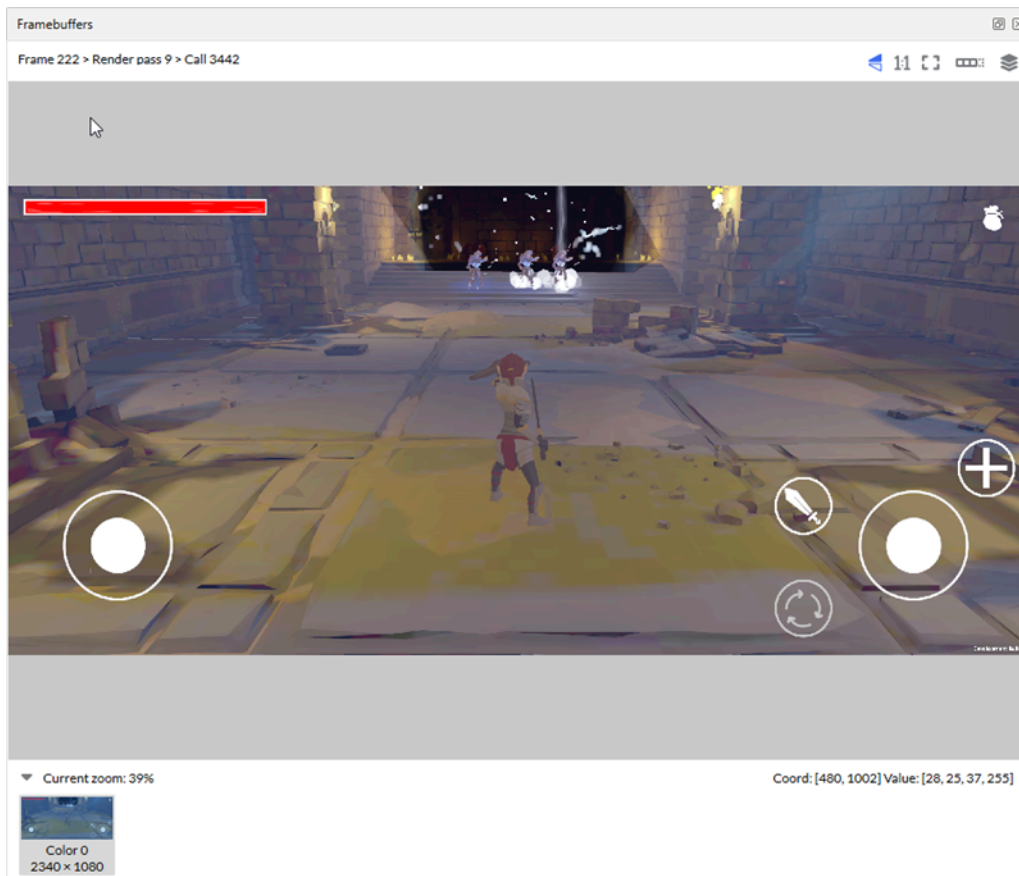
Figure 3-11: Render graph resource version



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

Framebuffers view

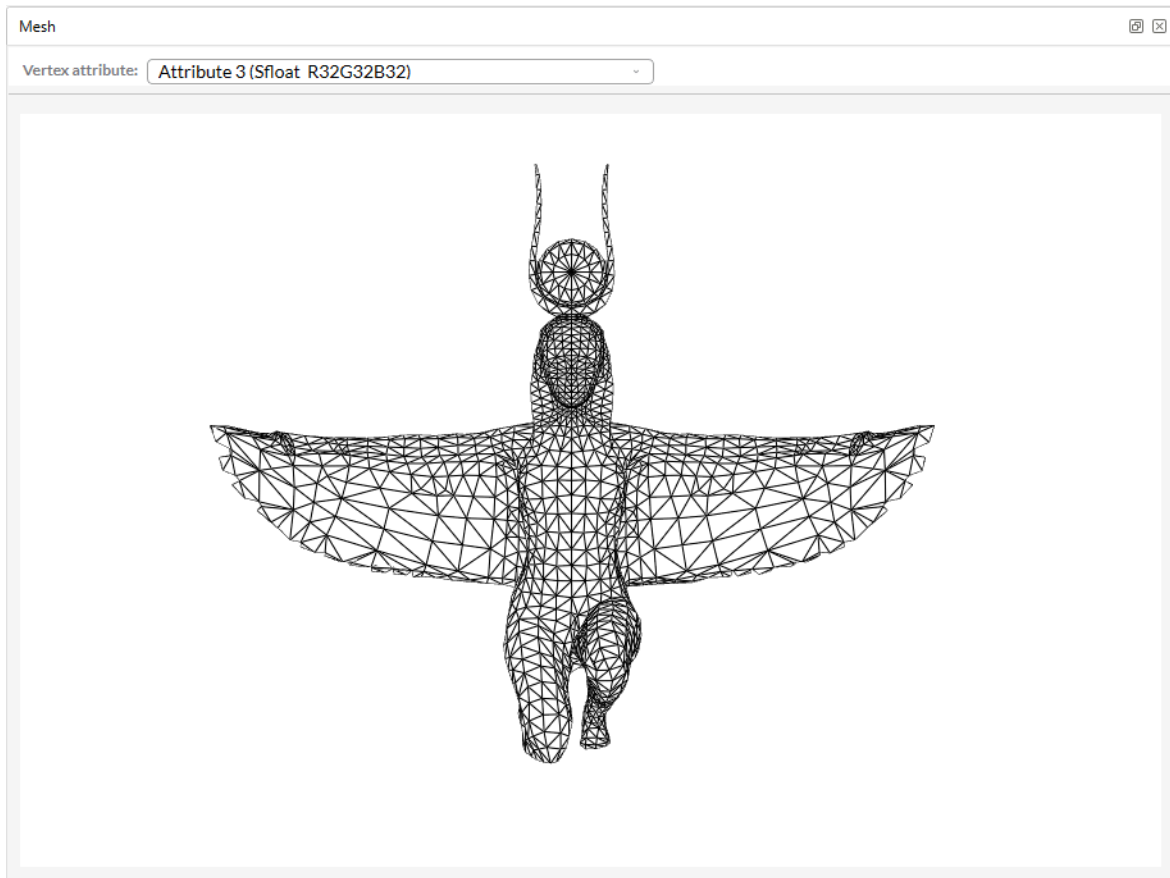
The **Framebuffers** view visualizes the rendered output for your captured frames. See how your frames are composed from the GPU rendered output by stepping through each draw call.

Figure 3-12: Framebuffers view

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

Mesh view

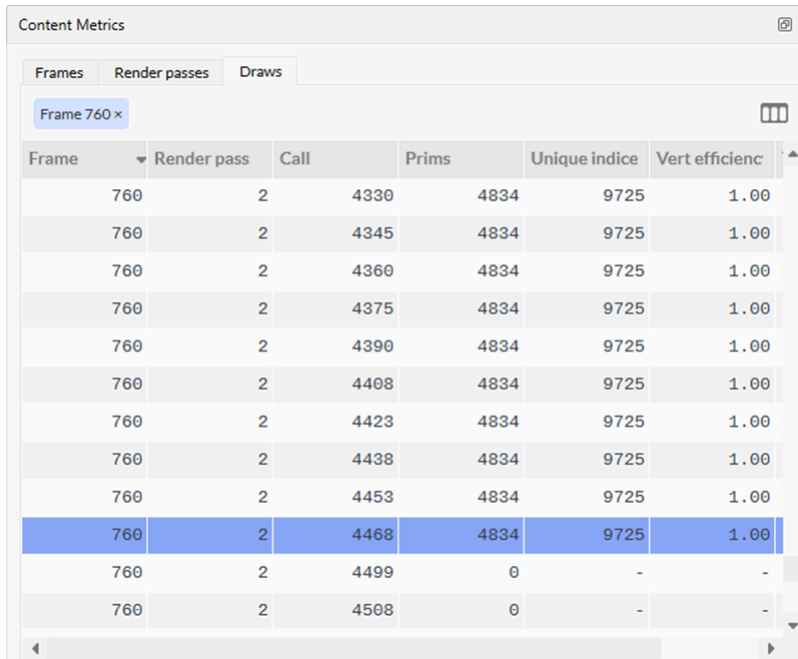
The **Mesh** view shows the mesh of objects in your rendered output. Use the **Mesh** view with the **Framebuffers** view to check complexity of objects in relation to their size and position on screen. Use the **Vertex attribute** menu to visualize different attributes of your mesh.

Figure 3-13: Mesh view

This view helps you to identify very small triangles, long and thin triangles, and draw calls that submit the same geometry, all of which use a lot of processing power. See [Analyze object complexity](#).

Content Metrics view

The **Content Metrics** view shows a table of calculated rendering metrics broken down by frame, render pass, and draw call.

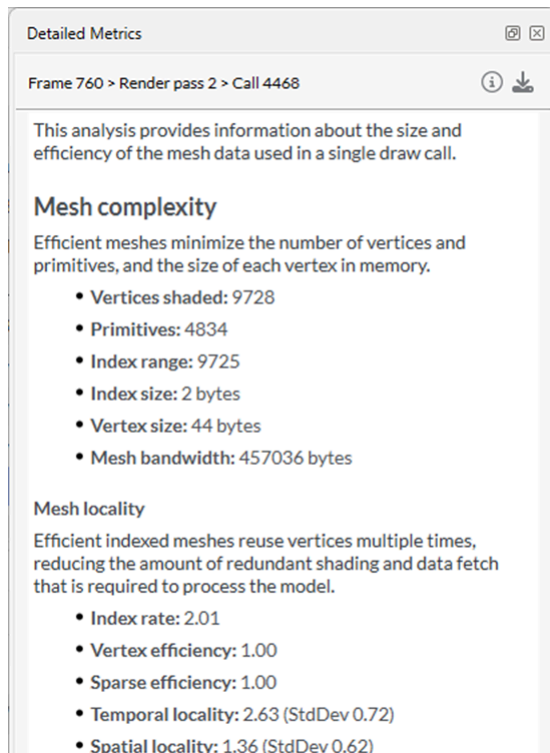
Figure 3-14: Content Metrics view


Frame	Render pass	Call	Prims	Unique indice	Vert efficienc
760	2	4330	4834	9725	1.00
760	2	4345	4834	9725	1.00
760	2	4360	4834	9725	1.00
760	2	4375	4834	9725	1.00
760	2	4390	4834	9725	1.00
760	2	4408	4834	9725	1.00
760	2	4423	4834	9725	1.00
760	2	4438	4834	9725	1.00
760	2	4453	4834	9725	1.00
760	2	4468	4834	9725	1.00
760	2	4499	0	-	-
760	2	4508	0	-	-

To help you find expensive draw calls, filter the data to narrow it down to a range of interest. Sort the metrics to identify large draw calls, or calls that show signs of inefficiency. For example, to find the most complex draws in the render pass, filter and sort the table by the number of primitives. See [Analyze model geometry](#).

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.

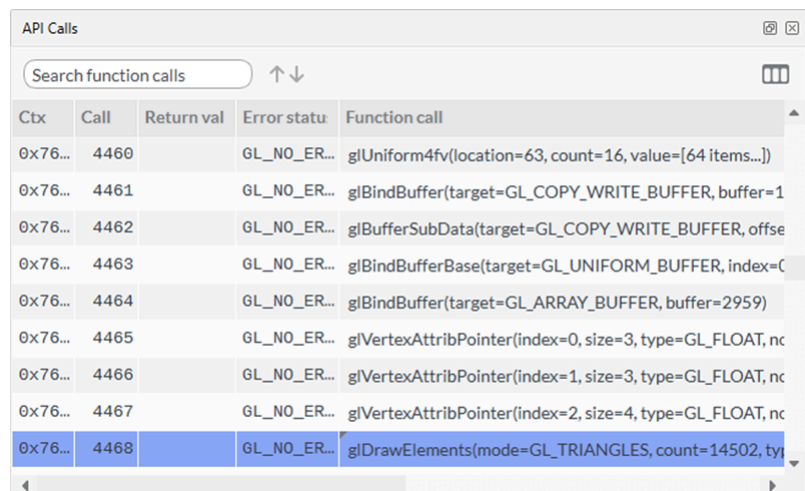
Figure 3-15: Detailed Metrics view

Complicated meshes containing [many triangles](#) are expensive for the GPU to process and, depending on the object's position and size on screen, may get no benefit from the added complexity. Reducing the number of triangles in a mesh dramatically reduces shader cost. If the object's silhouette is accurately drawn, you can use textures or normal maps to efficiently create the detail within the silhouette.

Just as important as complexity, is whether the object's mesh 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 draw call in the **Frame Hierarchy** view, the API call that requested the draw is highlighted in the **API Calls** view.

Figure 3-16: API Calls view


Ctx	Call	Return val	Error statu	Function call
0x76...	4460		GL_NO_ER...	glUniform4fv(location=63, count=16, value=[64 items...])
0x76...	4461		GL_NO_ER...	glBindBuffer(target=GL_COPY_WRITE_BUFFER, buffer=1
0x76...	4462		GL_NO_ER...	glBufferSubData(target=GL_COPY_WRITE_BUFFER, offse
0x76...	4463		GL_NO_ER...	glBindBufferBase(target=GL_UNIFORM_BUFFER, index=C
0x76...	4464		GL_NO_ER...	glBindBuffer(target=GL_ARRAY_BUFFER, buffer=2959)
0x76...	4465		GL_NO_ER...	glVertexAttribPointer(index=0, size=3, type=GL_FLOAT, nc
0x76...	4466		GL_NO_ER...	glVertexAttribPointer(index=1, size=3, type=GL_FLOAT, nc
0x76...	4467		GL_NO_ER...	glVertexAttribPointer(index=2, size=4, type=GL_FLOAT, nc
0x76...	4468		GL_NO_ER...	glDrawElements(mode=GL_TRIANGLES, count=14502, ty

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

Related information

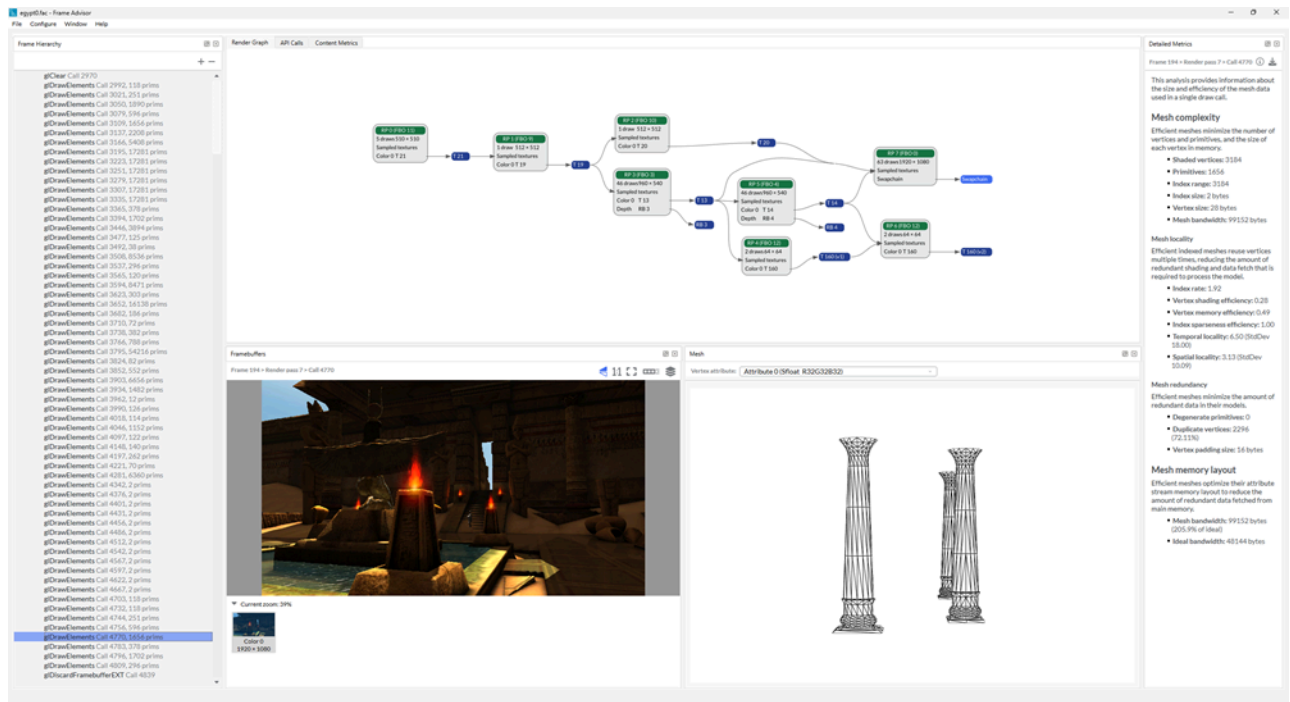
- [Analyzing your frames](#)
- [How to get help](#)

4. 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. Navigate between the views to identify problem areas in your application, or areas where you can improve performance.

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 file**.

Figure 4-1: Example Analysis screen



4.1 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 render passes and draw calls contained within them. Select an item to update the information shown in other views of the **Analysis** screen.

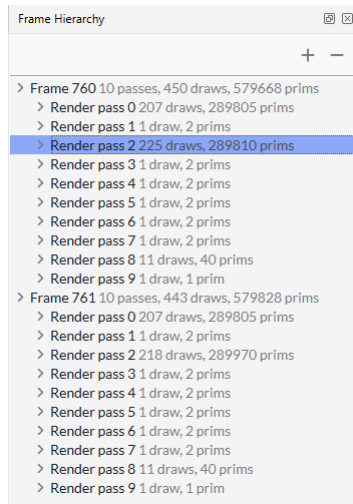


Note

You must select or undock the **API Calls** and **Content Metrics** views to see them.

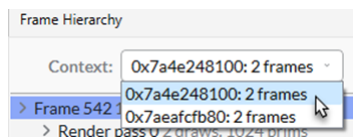
To help you to find items that require further analysis, each item in the **Frame Hierarchy** view is presented with contextual information. For example, you may want to select a frame that contains the most draws or primitives to investigate it further.

Figure 4-2: Frame Hierarchy view



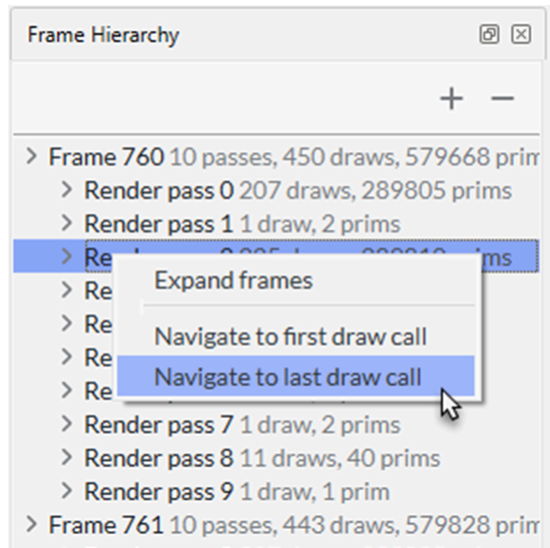
If your trace uses multiple contexts, you can choose the context from the drop-down menu. If your trace uses one context only, the drop-down menu is not visible.

Figure 4-3: Context selection



As you select a frame, the **Render Graph**, **Framebuffers**, and **API Calls** views update to show the output from the last draw call in the frame. Use these views with the **Frame Hierarchy** to narrow down the problem area.

You can use the right-click menu to navigate directly to the first or last draw call in the render pass, which is useful if a render pass contains a high number of draw calls. Alternatively, use the mouse or the arrow keys to step through the draw calls in a render pass.

Figure 4-4: Navigate to the last draw 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.



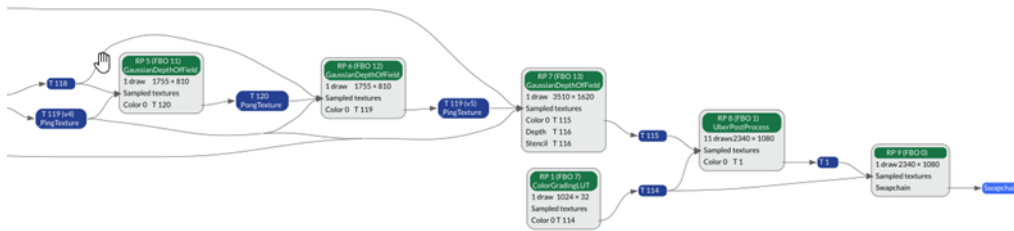
To learn more about how to avoid common API problems when building graphics for mobile, watch *Episode 2.4* of the *Mali GPU training series* [Engine and API best practices](#)

4.2 Analyze workloads using the Render Graph

The **Render Graph** shows the rendering operations performed for the frame selected in the **Frame Hierarchy** view. See how your GPU workloads are processed in the API sequence by analyzing the data flow between render passes, as well as the resources that are produced and consumed by the render passes. You can then use this information to identify expensive parts of a frame that process workloads inefficiently, and improve workload performance.

Procedure

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

Figure 4-5: Example render graph

Click on a render pass in the graph to navigate to it, or right-click and navigate to the first or last draw call of that render pass.

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 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 window** 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 for each render pass, contextual information is provided about the workload performed, such as:

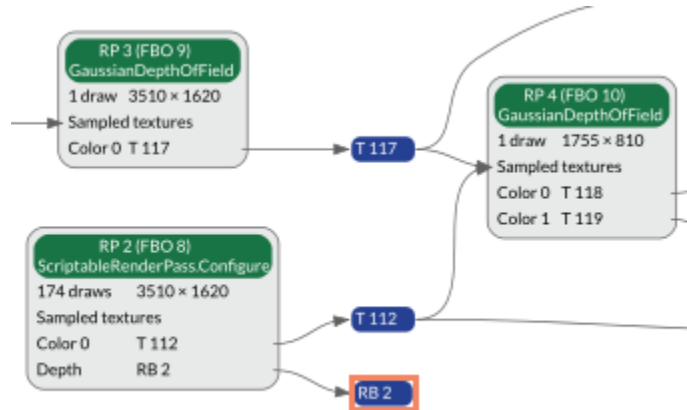
- Index of the render pass, and the framebuffer object that it relates to.
 - Number of draw calls in the render pass.
 - Rendered resolution
 - Input and output attachments
3. Click a render pass to navigate to it and update information shown in the other views of the **Analysis** screen.
 4. Look for render passes that are consuming bandwidth and accessing memory unnecessarily, such as:
 - Render passes where the attachment of one render pass is immediately input to another without being used. Merging these 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. 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. Writing these output attachments to external DRAM at the end of a render pass consumes

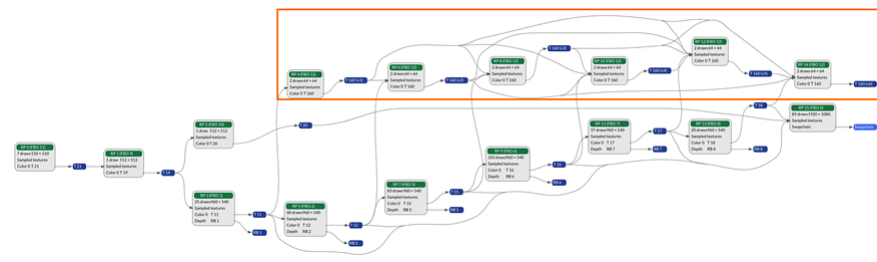
write memory bandwidth. If the output attachment is no longer needed after the render pass, ensure that you invalidate it so that it is discarded at the end of the render pass.

Figure 4-6: Output attachment that requires discarding



- Render passes where output attachments are not used in the final rendered output. Ensure that all render passes that are submitted to the GPU are actually useful rendering operations, with no redundant rendering. Discard any unused attachments at the end of the render pass so that they do not consume memory bandwidth.

Figure 4-7: Unused nodes

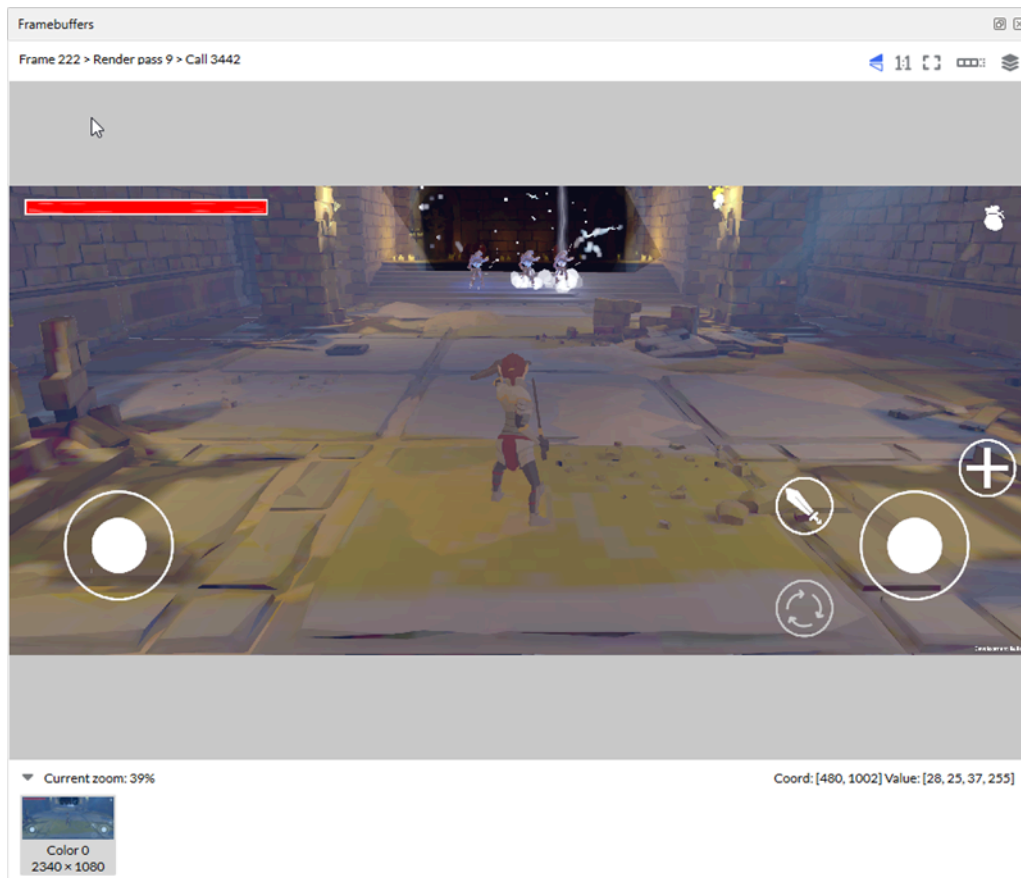


4.3 Analyze object rendering

The **Framebuffers** 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

- 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.

Figure 4-8: Framebuffers view

If required, use the **Flip vertically** button to adjust the vertical orientation so that it appears the correct way around.

2. Expand a render pass in the **Frame Hierarchy** view, then use the arrow keys or mouse to step through each draw call in sequence. See how your frame is composed by watching how the framebuffer changes as you step through the draw calls. 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.

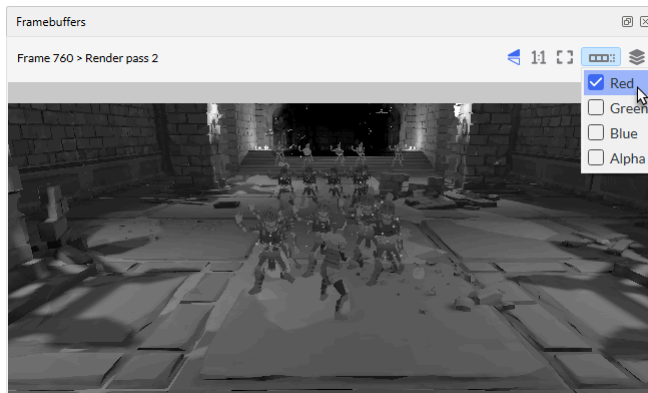


You can temporarily hide the thumbnails of the framebuffer attachments to provide more space for the framebuffer visualization.

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 window** 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 4-9: 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 check box if you do not want to see this information.

5. Check the ordering of objects to avoid high levels of overdraw. First 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](#) to prevent shading of occluded objects, which reduces overdraw and unnecessary processing before fragment shading.

Then render transparent objects 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.

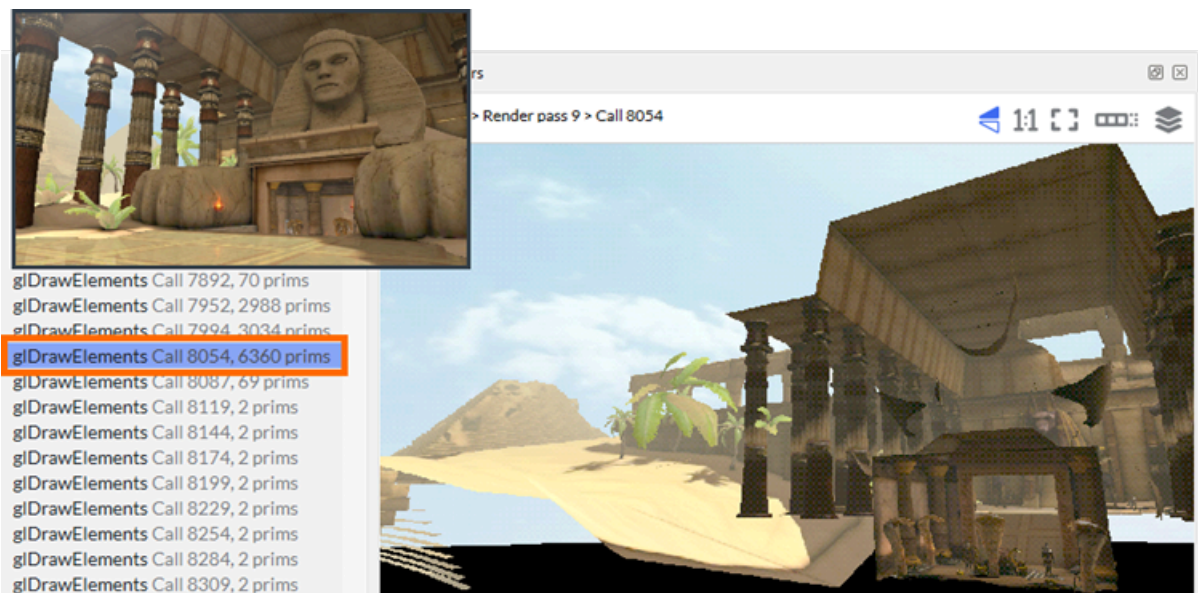
6. If you captured frames using **Overdraw mode**, click the overdraw thumbnail to show it in the visualization area. Move the cursor around to see the levels of overdraw in the rendered output. Every time a pixel is rendered to the framebuffer, the overdraw value increases, and the lighter it appears in the final image.

Figure 4-10: Framebuffers overdraw view

Overdraw occurs when the same pixel is shaded on many layers. High levels of overdraw affects performance negatively because shading pixels, which are then overwritten before they are shown, wastes GPU resources by unnecessary processing.

7. As you step through the draw calls, identify any draw calls that do not make visible changes to the framebuffer output.

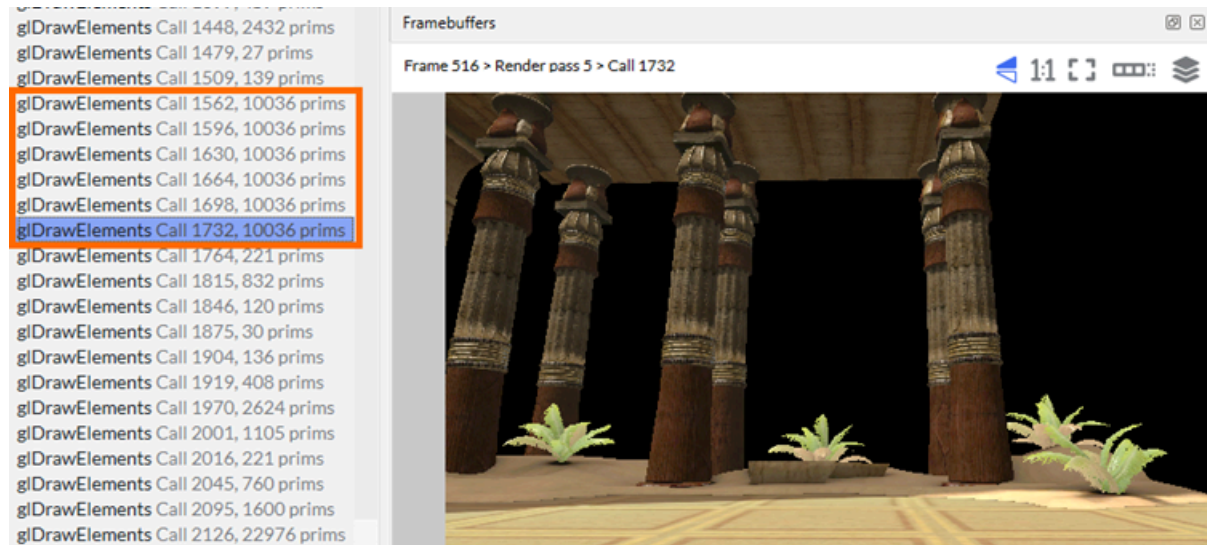
Check if your application is drawing objects that are either not visible on screen, or are occluded by other objects. Ensure your GPU is **culling** any primitives that are not visible on screen.

Figure 4-11: 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, displayed inset.

8. As you step through the draw calls, check for multiple identical objects that are drawn individually, instead of being processed as a batch in a draw call.

Figure 4-12: 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.

9. 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 4-13: HDR image comparison

10. 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

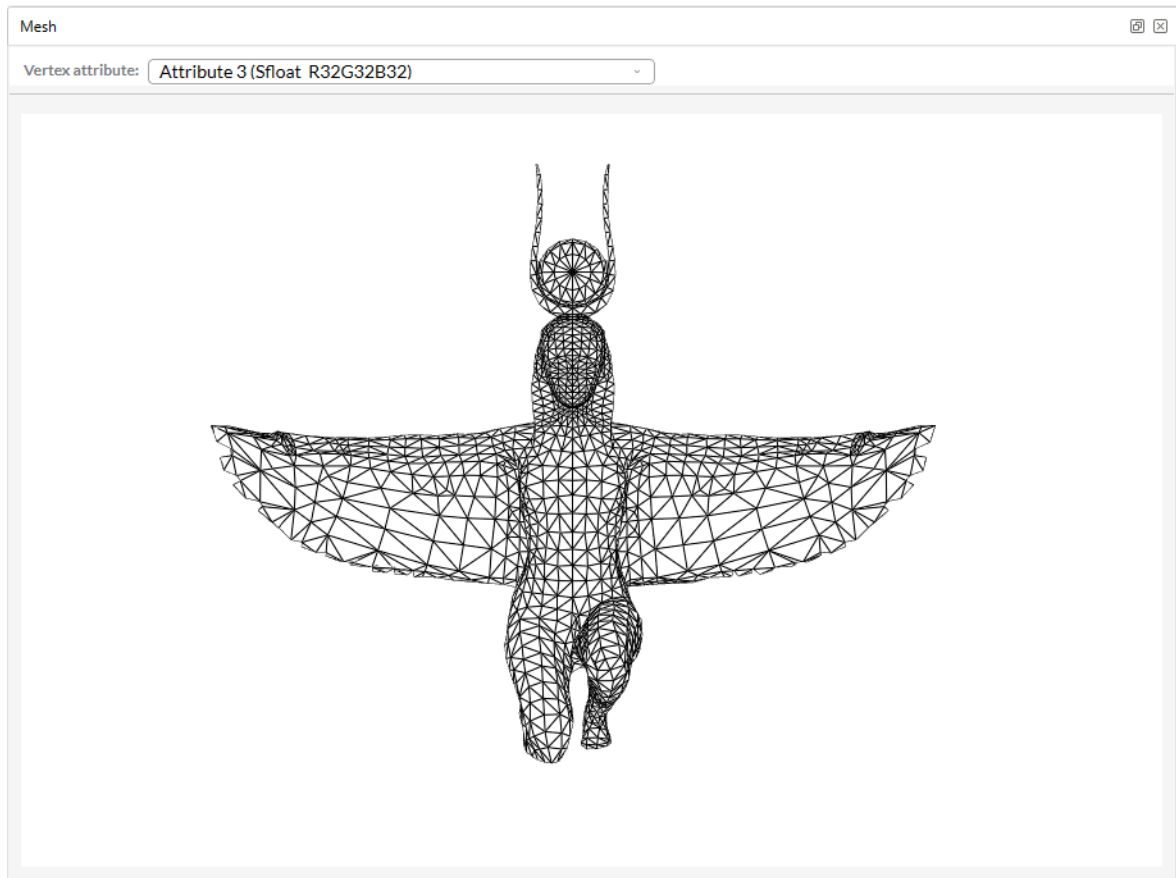
Investigate the efficiency of your mesh further in the **Detailed Metrics** view. See [Analyze object complexity](#) and [Content metrics in detail](#) for more information.

4.4 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.

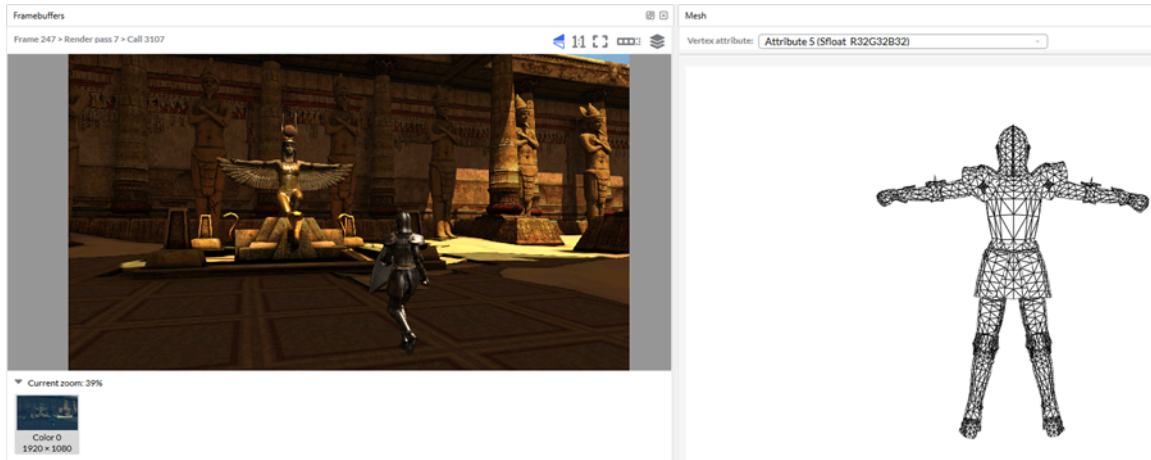
Figure 4-14: Mesh view

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

Use the mouse controls to move around the visualization:

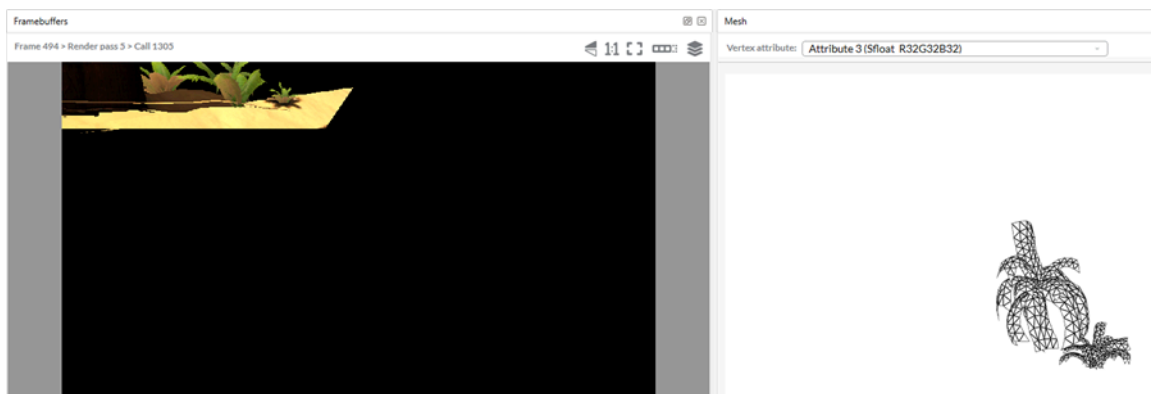
- Drag to rotate
 - Shift+click, or hold middle mouse button, to pan
 - Scroll the wheel button to zoom
 - Right-click and select **Reset view** from the context menu to return the visualization to the default position
3. Look at the visualization in the **Mesh** view to see if all of the primitives 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.



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

Figure 4-15: 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 bandwidth, even though they are culled in a later step.

- 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 4-16: 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 range: 45952
- Index size: 2 bytes
- Vertex size: 28 bytes
- Mesh bandwidth: 781184 bytes

Related information

[Arm GPU best practices](#)

4.5 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. For example, find out if you have any duplicated vertices or degenerative primitives. From any draw call, you can navigate to the corresponding API call so that you can see exactly which function call 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. In this example, we captured one frame only.
2. Click the **Render passes** tab.
3. You can sort the columns in the tables to help you find issues. To sort the tables, click a column heading. In this example, we click the **Draw calls** column heading to sort the table so that the render pass with the highest number of draw calls is at the top. Looking at a large number of draw calls gives us a higher chance to find issues.

Figure 4-17: Sorted draw calls column

Content Metrics				
Frames Render passes Draws				
Frame 332 x				
Frame	Render pass	Draw calls	Prims	Unique indice
332	2	192	247991	247991
332	0	163	247984	247984
332	8	11	80	80
332	1	1	4	4

- Select the top row of the table, which should be the render pass with the highest number of draw calls.
 - To see all the draw calls in this render pass, click the **Draws** tab.
 - Sort the table so that the draw call with the lowest **VSE** is shown 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, you can see that **Call 3084** has a **VSE** of 0.00.

Figure 4-18: Sorted VSE column

Content Metrics								
Frames Render passes Draws								
Frame 332 > Render pass 2 x								
Frame	Render pass	Call	Prims	Unique indice	Vert size	VSE	VME	
332	2	3084	4834	9725	4	0.00	1.00	
332	2	3333	5	7	24	0.88	1.00	
332	2	2193	45	110	32	0.90	1.00	
332	2	2233	45	110	32	0.90	1.00	

To find out what is causing the low VSE, open the **Detailed Metrics** view.

- To open the **Detailed Metrics** view for the chosen call, right-click on that call in the table, then select **Navigate to call**. The call is now also selected in the **Frame Hierarchy**, **Mesh**, and the **API Calls** views, and the **Detailed Metrics** view is populated with data about that call.
- In the **Detailed Metrics** view, you can see that nearly all the vertices are duplicated and they are all being shaded. This is why the **Vertex Shading Efficiency** is 0.00.

Figure 4-19: Vertex shading efficiency

Mesh redundancy

Efficient meshes minimize the amount of redundant data in their models.

- Degenerate primitives: 0
- Duplicate vertices: 9681
- Duplicate vertex percentage: 99.55%
- Vertex padding size: 0 bytes

To improve the VSE and the performance of the application, remove all duplicate vertices.

There are many other metrics you can look at to improve the performance of your application. We are now going to add a column to show degenerate primitives.


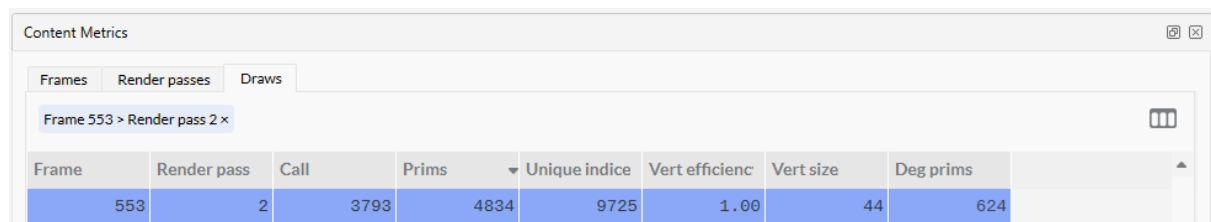
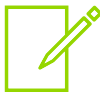
9. To add and remove columns of data in the **Content Metrics** view, ensure you are on the **Draws** tab, then click the **Edit columns** icon .
10. Select **Degenerative primitives** from the list of available columns.
11. Sort the **Deg prims** column to see the draw calls ordered by the highest number of degenerate primitives first.
12. Select a draw call that has degenerative primitives.

Figure 4-20: Degenerate primitives



Frame	Render pass	Call	Prims	Unique indice	Vert efficienc	Vert size	Deg prims
553	2	3793	4834	9725	1.00	44	624



Note

To clear the filters and see data for all the frames in your trace, click the filters.

13. To locate which API call is creating the degenerate primitives, right-click a draw call that has degenerative primitives, and select **Navigate to call**. The corresponding API call is highlighted in the **API Calls** view.

4.6 Content metrics in detail

The **Detailed Metrics** view helps you to understand the cost of meshes and find opportunities to improve the efficiency of meshes. 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. The recommended mesh memory layout also repacks the attributes to remove any padding, unless it is required for type alignment.

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**.

4.6.1 Mesh complexity

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

Vertices shaded

The number of vertices shaded by the GPU, factoring in any reshading or overshading effects.

Primitives

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

Index range

The difference between the minimum index and maximum index.

Index size

The size of a single 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 the current memory layout.

4.6.2 Mesh locality

Efficient indexed meshes reuse vertices multiple times, which reduces the amount of redundant shading and data fetch required to process the model.

Index rate

The number of unique indices per primitive.

The **Index rate** is a measure of how many vertices are shared across multiple primitives.

Sharing vertices reduces memory usage, which improves efficiency. An **Index rate** of 3 indicates that no vertices are shared, so aim to get your **Index rate** as low as possible.

To reduce your **Index rate**, share as many vertices as possible.

Vertex shading efficiency

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

An efficiency of 1 indicates optimal shading efficiency, with 1 shader invocation per useful input vertex. An efficiency of 0.5 indicates 2 shader invocations per useful 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 **average temporal locality**, **duplicate vertices**, and **index sparseness efficiency** metrics.

Vertex memory efficiency

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

An efficiency of 1 indicates optimal memory layout. An efficiency of 0.5 indicates that twice as much data is fetched because of an unoptimized 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](#).

Index sparseness efficiency

The ratio of vertex indices shaded to the number of unique indices in the index buffer.

Arm® GPUs always shade indices in groups of 4 aligned indices, even if those indices are not used in the model. For example, if you want to shade indices 0, 1, and 2, then index 3 is also shaded. If you want to shade indices 1, 6, and 9, then all indices from 0 to 11 are shaded. That is not very efficient because of the unnecessary shading. In this case, try to reconfigure the structure of your mesh so only 0, 1, 2, and 3 are shaded.

An efficiency of less than 1 indicates that the model is overshading because of unused indices in these groups of 4.

To maximize efficiency, minimize overshading by tightly packing meshes so as many indices between the minimum and maximum index value are used.

Average 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-byte 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 bytes. To reduce your average temporal locality, reduce the number of indices between reuse of an index value.

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.

4.6.3 Mesh redundancy

Efficient indexed meshes reuse vertices multiple times, which reduces the amount of redundant shading and data fetch required to process the model.

Instance degenerate primitives

The number of primitives that have zero area.

Degenerate primitives occur when the vertices of a primitive do not form a triangle. 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.

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 ensure each vertex is uploaded to the GPU only once. If that is not possible, remove as many duplicate vertices as possible.

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.

4.6.4 Mesh memory layout

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

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

Ideal mesh bandwidth

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

4.7 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.

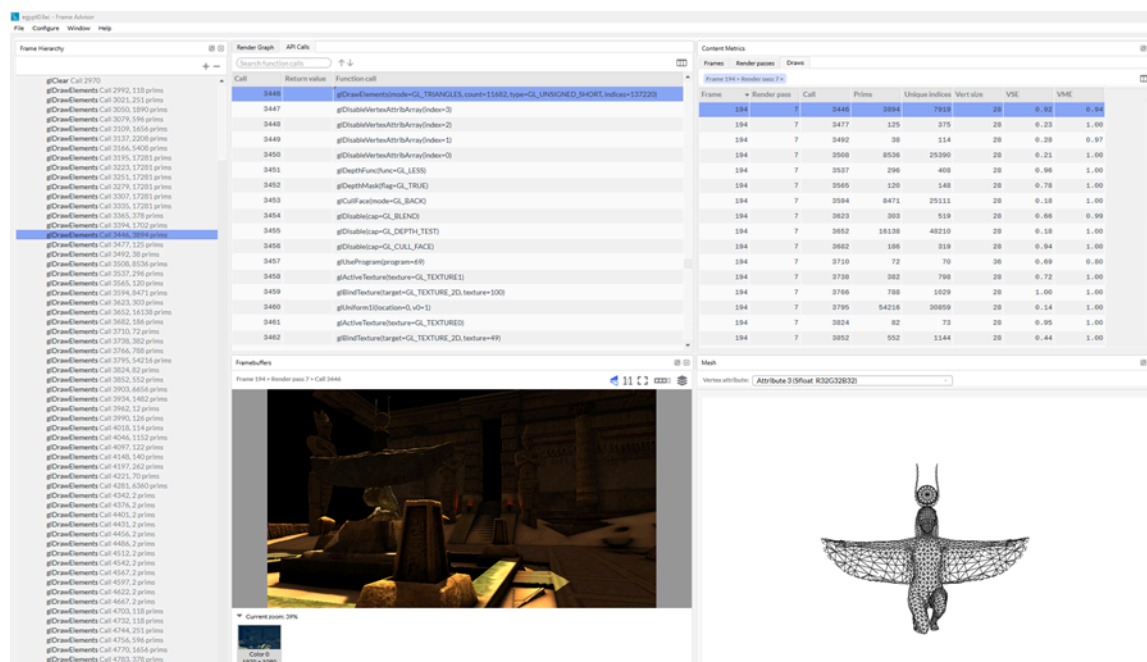
About this task

Here is an example of how you could use the **API Calls** view to identify where an issue is.

Procedure

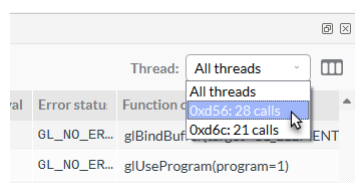
1. To see how the draw calls render in the **Framebuffers** view, step through the draw calls in the **Frame Hierarchy** view.
You can see that the draw call selected in the **Frame Hierarchy** view is also selected in the **API Calls** view.


Figure 4-21: Draw calls selected



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

Figure 4-22: See function calls for a specific thread



2. To add and remove columns showing different data for your function calls, click the **Edit columns** icon .
3. In the **API Calls** view, look for errors or other values that are not correct for that draw call. Alternatively, use the search box in the **API Calls** view to find inefficient uses of the API. For example, opaque draws that have blending enabled, or if back-face culling is disabled.

Next steps

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

5. How to get help

Arm® provides lots of resources for you to find more information about Arm® 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 -> Preferences** in the Frame Advisor menu.

6. Troubleshooting Frame Advisor

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

6.1 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.

6.2 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.

6.3 The Framebuffers view is slow to load images

When you click through the draw calls in your captured frame, the **Framebuffers** 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 **Framebuffers** view when they are accessed again.

Solution

1. To open the **Preferences** dialog, click **Configure**, then **Preferences**.
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 **Framebuffers** view load faster.

6.4 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. Click **Configure**, then **Preferences**.
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](#).

6.5 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 -> Preferences** in the Frame Advisor menu.

6.6 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

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