Arm® Frame Advisor

Version 1.0

User Guide

Non-Confidential

Issue 00
102693_0100_00_en

Arm® Frame Advisor
User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	21 November 2023	Non-Confidential	New document for v1.0

Proprietary Notice

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

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
italic	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: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Caution

Recommendations. Not following these recommendations might lead to system failure or damage.

Warning

Requirements for the system. Not following these requirements might result in system failure or damage.

Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	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
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

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

1.3 Other information

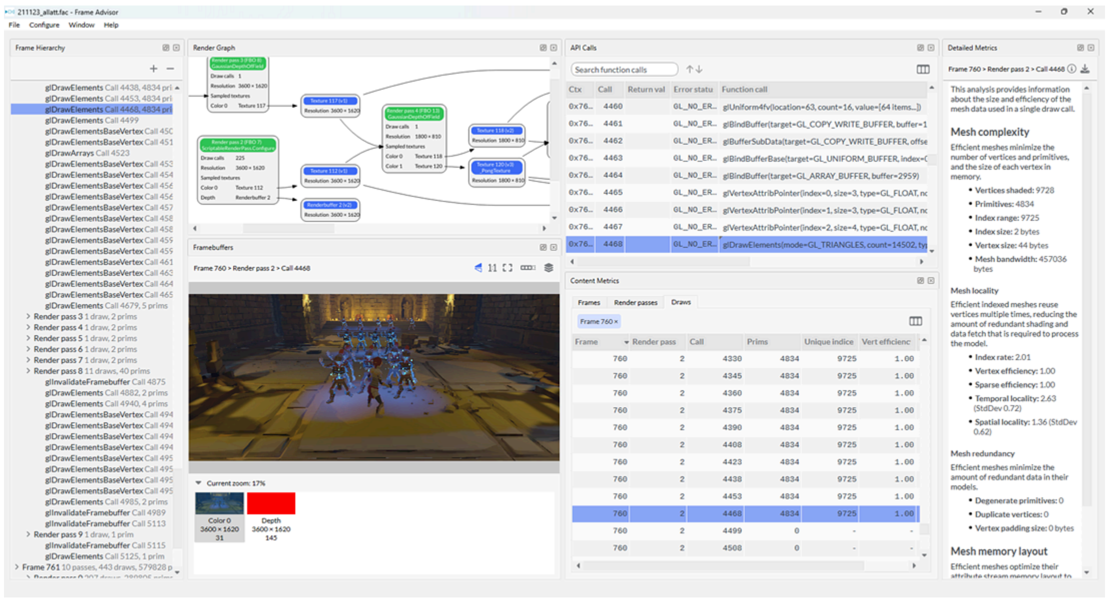
See the Arm website for other relevant information.

- 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, and see where you can improve your application.

Figure 2-1: The Frame Advisor Analysis screen



Frame Hierarchy:

glDrawElements Call 4438, 4834 pri
glDrawElements Call 4453, 4834 pri
glDrawElements Call 4468, 4834 pri
glDrawElements Call 4499
glDrawElementsBaseVertex Call 450
glDrawElementsBaseVertex Call 451
glDrawArrays Call 4523
glDrawElementsBaseVertex Call 453
glDrawElementsBaseVertex Call 454
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 457
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 461
glDrawElementsBaseVertex Call 463
glDrawElementsBaseVertex Call 464
glDrawElementsBaseVertex Call 465
glDrawElements Call 4679, 5 prims
> Render pass 3 1 draw, 2 prims
> Render pass 4 1 draw, 2 prims
> Render pass 5 1 draw, 2 prims
> Render pass 6 1 draw, 2 prims
> Render pass 7 1 draw, 2 prims
> Render pass 8 11 draws, 40 prims
glInvalidateFramebuffer Call 4875
glDrawElements Call 4882, 2 prims
glDrawElements Call 4940, 4 prims
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElements Call 4985, 2 prims
glInvalidateFramebuffer Call 4989
glInvalidateFramebuffer Call 5113
> Render pass 9 1 draw, 1 prim
glInvalidateFramebuffer Call 5115
glDrawElements Call 5125, 1 prim
> Frame 761 10 passes, 443 draws, 579828 p

API Calls:

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=(
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, typ

Content Metrics:

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	-	-

Detailed Metrics:

This analysis provides information about the size and efficiency of the mesh data used in a single draw call.

Mesh complexity

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

- **Vertices shaded:** 9728
- **Primitives:** 4834
- **Index range:** 9725
- **Index size:** 2 bytes
- **Vertex size:** 44 bytes
- **Mesh bandwidth:** 457036 bytes

Mesh locality

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

- **Index rate:** 2.01
- **Vertex efficiency:** 1.00
- **Sparse efficiency:** 1.00
- **Temporal locality:** 2.63 (StdDev 0.72)
- **Spatial locality:** 1.36 (StdDev 0.62)

Mesh redundancy

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

- **Degenerate primitives:** 0
- **Duplicate vertices:** 0
- **Vertex padding size:** 0 bytes

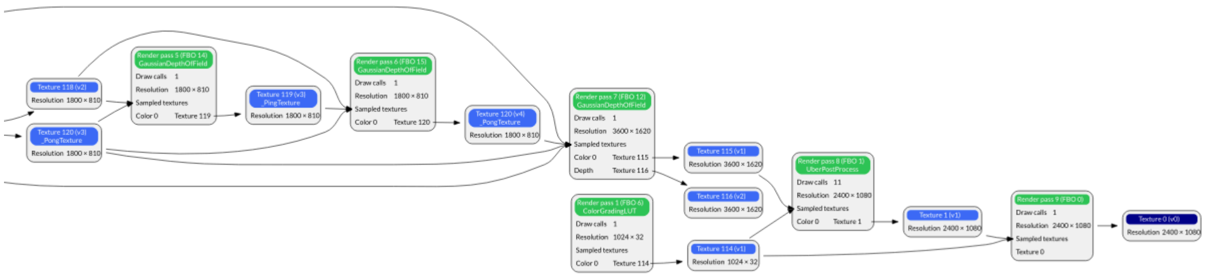
Mesh memory layout

Efficient meshes optimize their

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.

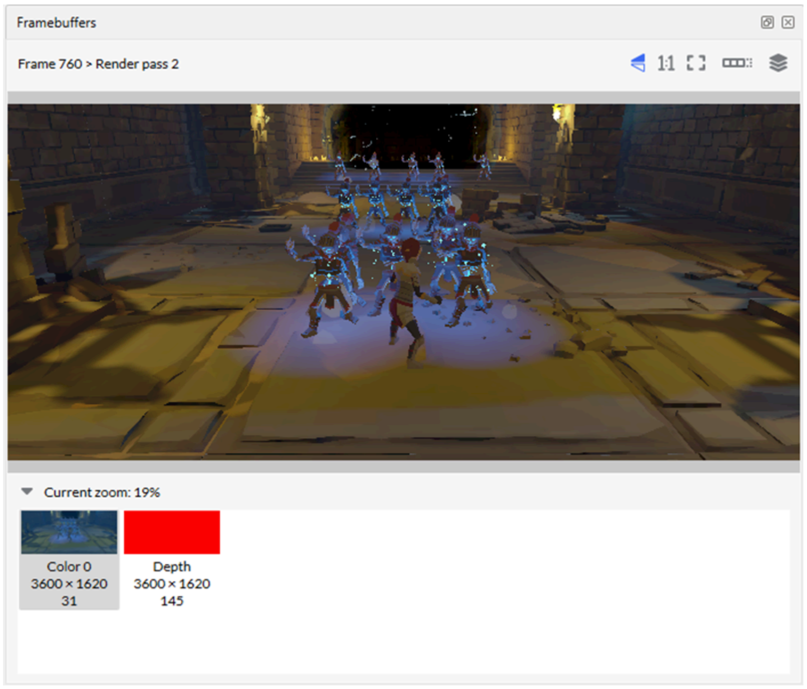
Figure 2-2: Render Graph



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 any draws that render in a back-to-front order, or 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



Evaluate your model geometry

Use the table of 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 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

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.

Note

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 mobilestudio@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 Mobile Studio and follow the installation instructions in the Arm Mobile 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.

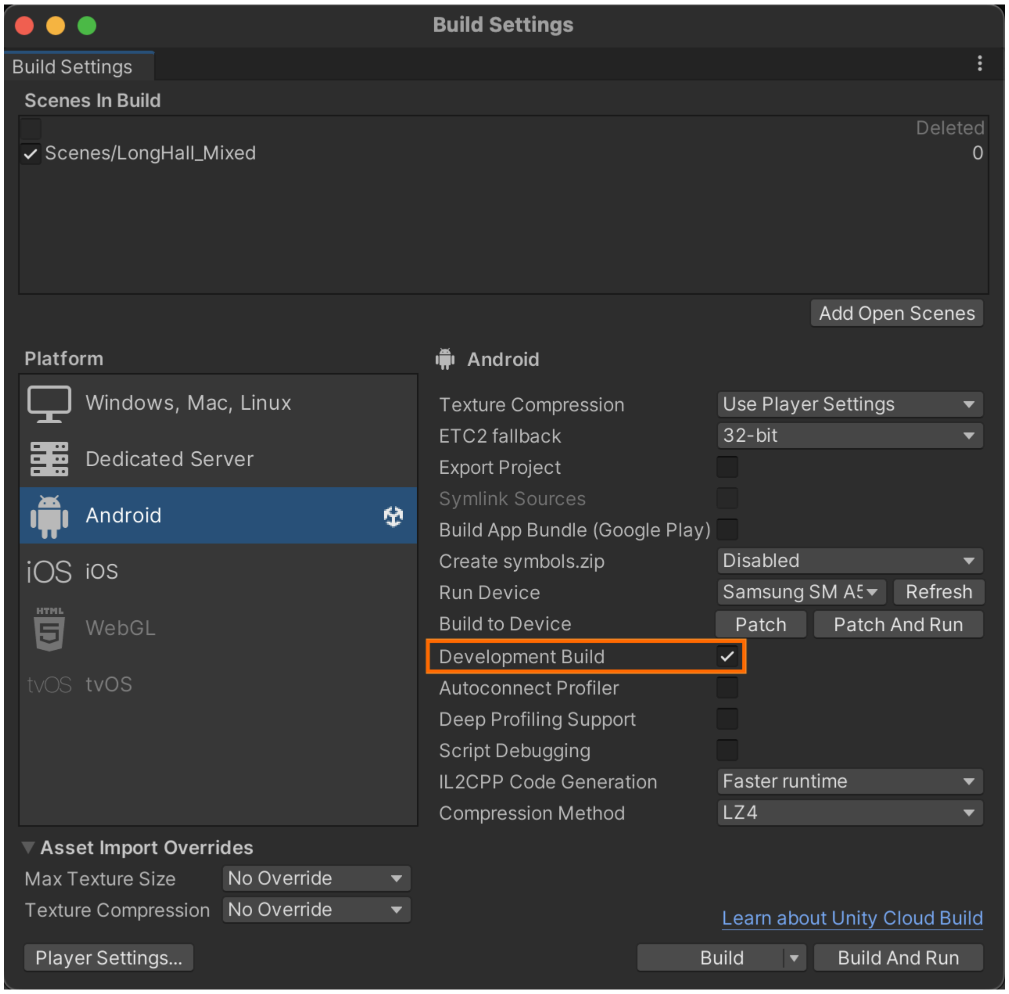
```
adb devices
List of devices attached
ce12345abcdf1a1234       device
```

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



✓ Scenes/LongHall_Mixed 0

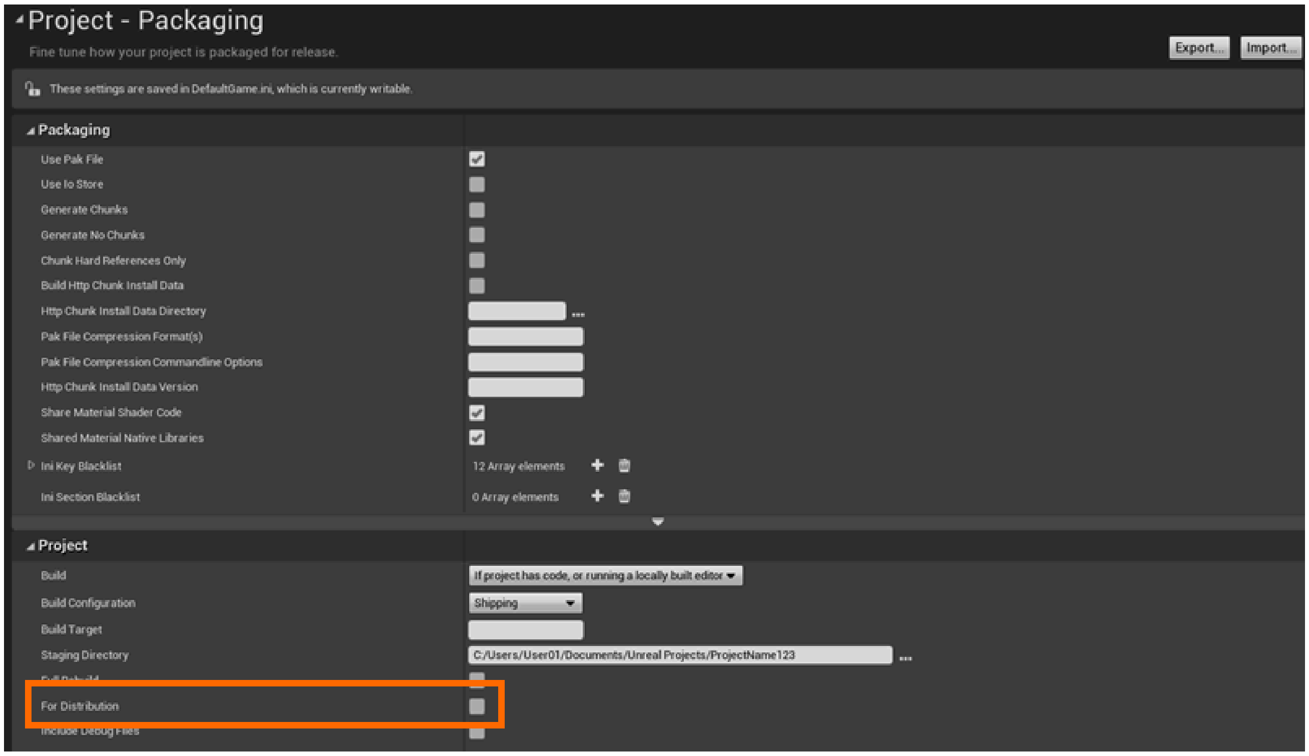
Texture Compression	Use Player Settings
ETC2 fallback	32-bit
Export Project	
Symlink Sources	
Build App Bundle (Google Play)	
Create symbols.zip	Disabled
Run Device	Samsung SM A5
Build to Device	
Development Build	✓
Autoconnect Profiler	
Deep Profiling Support	
Script Debugging	
IL2CPP Code Generation	Faster runtime
Compression Method	LZ4

Asset Import Overrides

Max Texture Size	No Override
Texture Compression	No Override

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



Project - Packaging

Fine tune how your project is packaged for release.

These settings are saved in DefaultGame.ini, which is currently writable.

Packaging	
Use Pak File	✓
Use Io Store	☐
Generate Chunks	☐
Generate No Chunks	☐
Chunk Hard References Only	☐
Build Http Chunk Install Data	☐
Http Chunk Install Data Directory	
Pak File Compression Format(s)	
Pak File Compression Commandline Options	
Http Chunk Install Data Version	
Share Material Shader Code	✓
Shared Material Native Libraries	✓
Ini Key Blacklist	12 Array elements
Ini Section Blacklist	0 Array elements
Project	
Build	If project has code, or running a locally built editor
Build Configuration	Shipping
Build Target	
Staging Directory	C:/Users/User01/Documents/Unreal Projects/ProjectName123
Full Rebuild	☐
For Distribution	☐
Include Debug Files	☐

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.

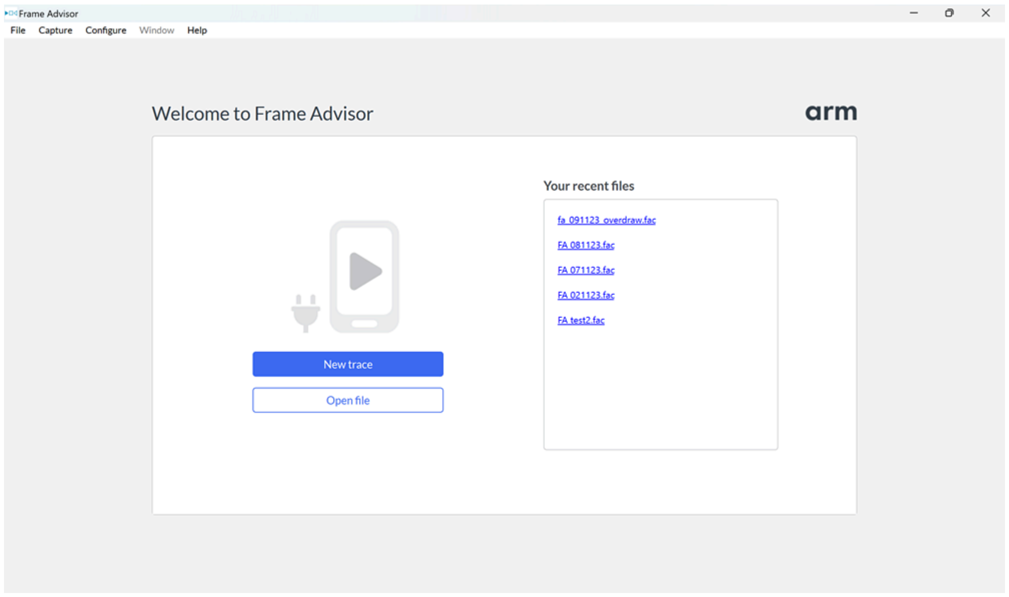
Procedure

1. Open Frame Advisor:
    - On Windows, from the **Start** menu, navigate to **Arm MS <version>** and select **Arm MS Frame Advisor <version>**.
    - On macOS, navigate to the `<install_directory>/frame_advisor` folder, and double-click the `Frame_Advisor.app` file.
    - On Linux, navigate to the `<install_directory>/frame_advisor` directory in a terminal, and run the `frame_advisor` file:

    ```
    cd <install_directory>/frame_advisor
    ./frame_advisor
    ```

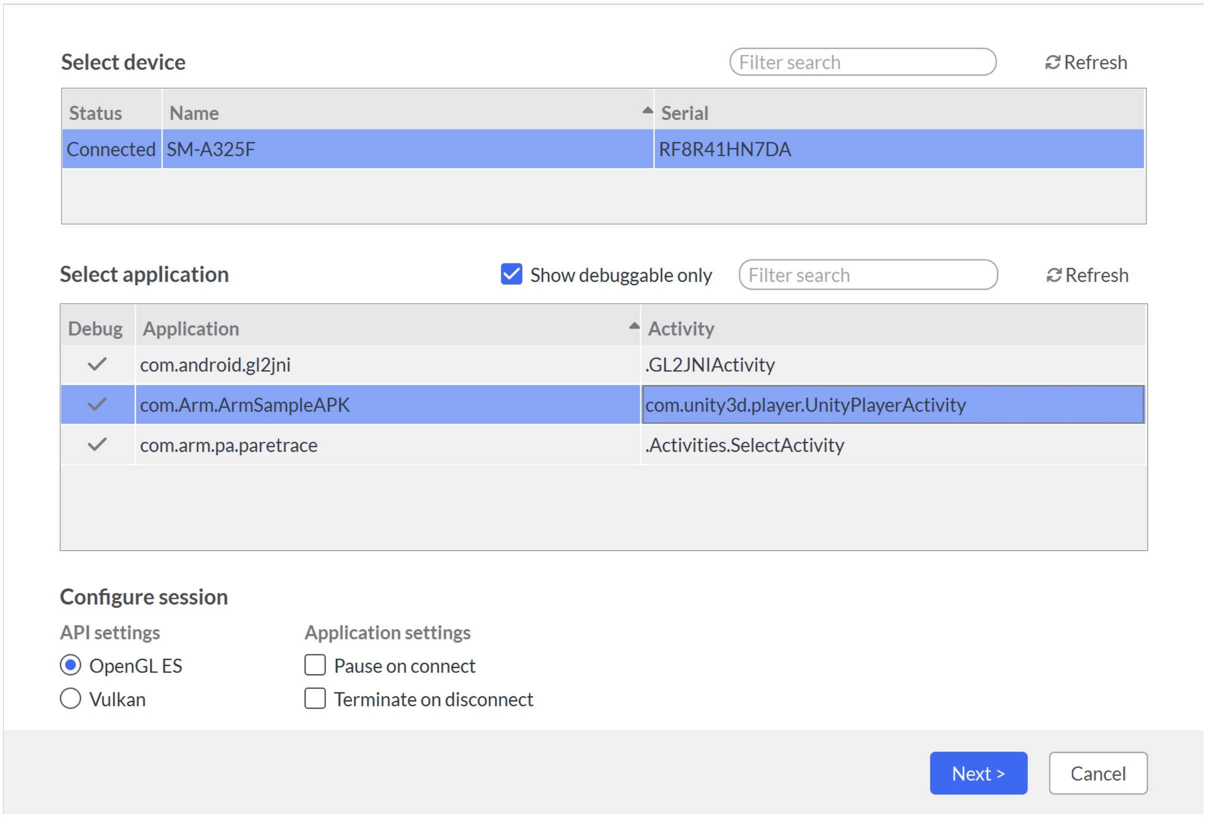
2. When Frame Advisor launches, you can either start a new trace or open an existing one.

   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

Status	Name	Serial
Connected	SM-A325F	RF8R41HN7DA

Select application

[✓] Show debuggable only

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

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**.

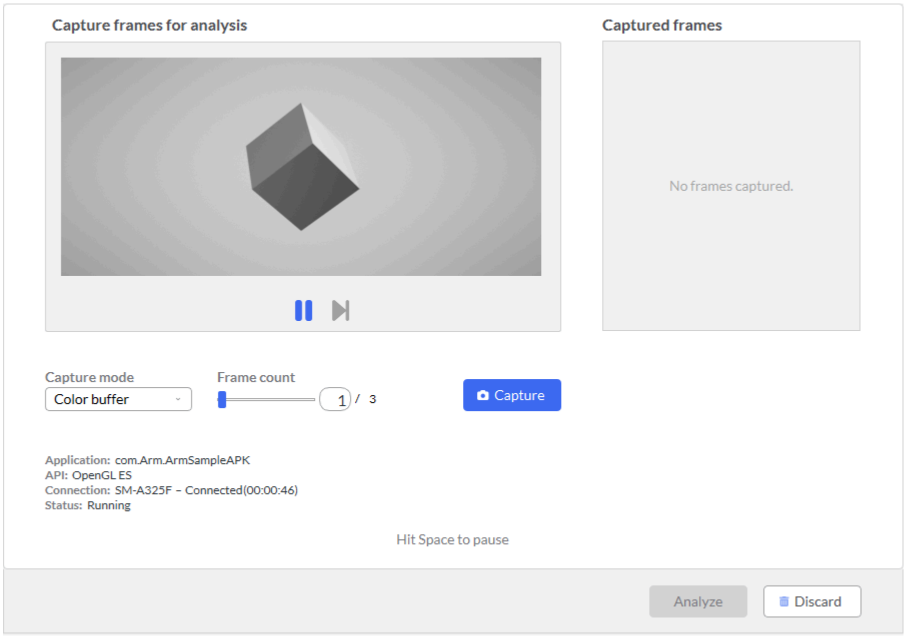
Optionally, you can use the **Application settings** to pause the application when Frame Advisor connects to it, and to stop the application when the capture is complete.

Click **Next >** to continue.

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

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

Figure 3-5: Frame Advisor capture screen



Application: com.Arm.ArmSampleAPK
API: OpenGL ES
Connection: SM-A325F - Connected(00:00:46)
Status: Running

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

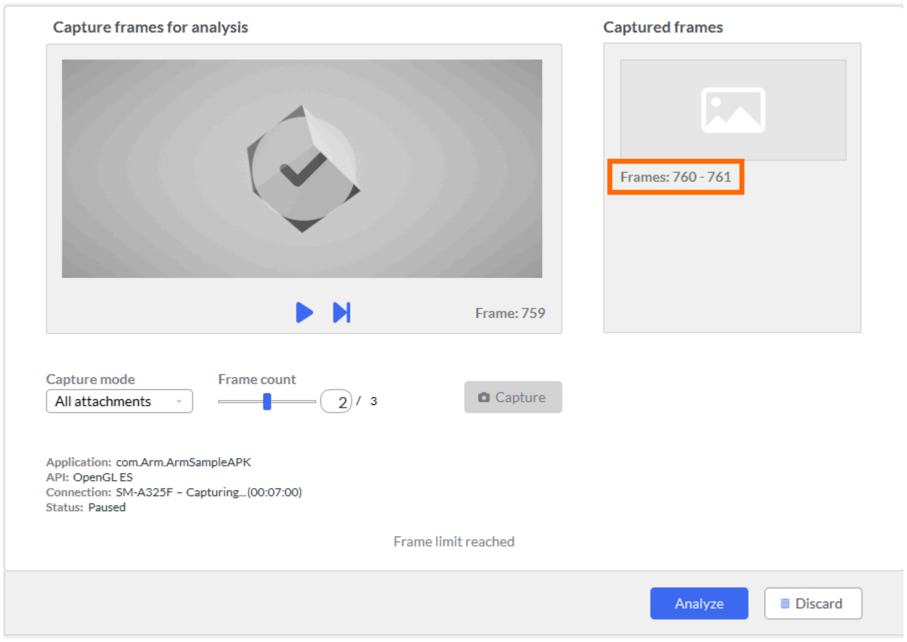
You can capture up to 3 frames. The larger the frame count, the longer the time it takes for Frame Advisor to capture the frames.

---

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



Application: com.Arm.ArmSampleAPK
API: OpenGL ES
Connection: SM-A325F – Capturing...(00:07:00)
Status: Paused

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

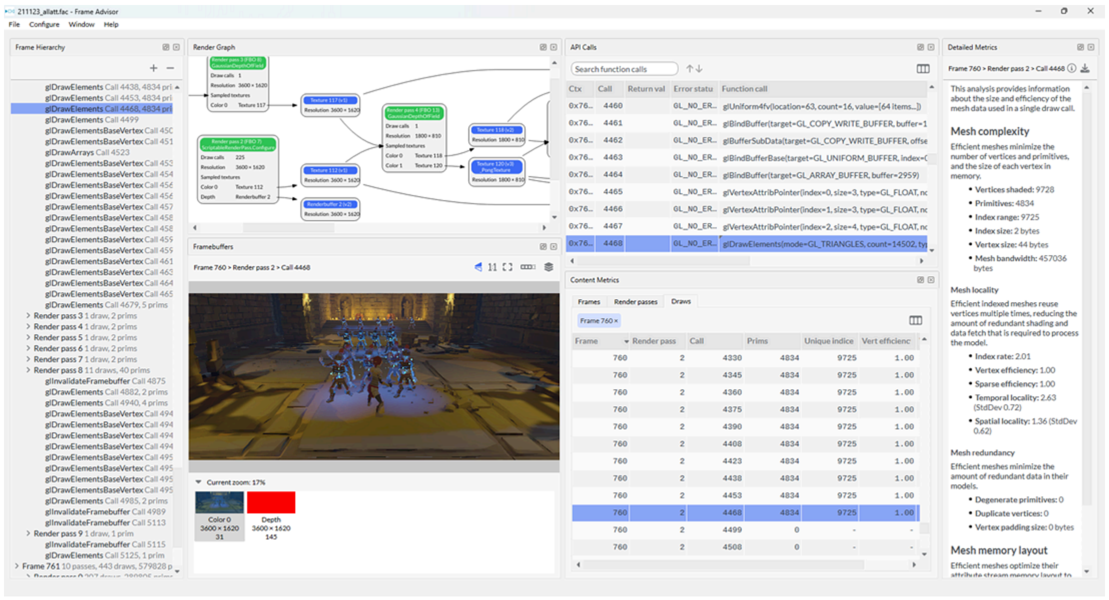
Next steps

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



glDrawElements Call 4438, 4834 pri
glDrawElements Call 4453, 4834 pri
glDrawElements Call 4468, 4834 pri
glDrawElements Call 4499
glDrawElementsBaseVertex Call 450
glDrawElementsBaseVertex Call 451
glDrawArrays Call 4523
glDrawElementsBaseVertex Call 453
glDrawElementsBaseVertex Call 454
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 457
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 461
glDrawElementsBaseVertex Call 463
glDrawElementsBaseVertex Call 464
glDrawElementsBaseVertex Call 465
glDrawElements Call 4679, 5 prims
> Render pass 3 1 draw, 2 prims
> Render pass 4 1 draw, 2 prims
> Render pass 5 1 draw, 2 prims
> Render pass 6 1 draw, 2 prims
> Render pass 7 1 draw, 2 prims
> Render pass 8 11 draws, 40 prims
glInvalidateFramebuffer Call 4875
glDrawElements Call 4882, 2 prims
glDrawElements Call 4940, 4 prims
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 494
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElementsBaseVertex Call 495
glDrawElements Call 4985, 2 prims
glInvalidateFramebuffer Call 4989
glInvalidateFramebuffer Call 5113
> Render pass 9 1 draw, 1 prim
glInvalidateFramebuffer Call 5115
glDrawElements Call 5125, 1 prim
> Frame 761 10 passes, 443 draws, 579828 p

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=(
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

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	-	-

This analysis provides information about the size and efficiency of the mesh data used in a single draw call.

Mesh complexity

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

- **Vertices shaded:** 9728
- **Primitives:** 4834
- **Index range:** 9725
- **Index size:** 2 bytes
- **Vertex size:** 44 bytes
- **Mesh bandwidth:** 457036 bytes

Mesh locality

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

- **Index rate:** 2.01
- **Vertex efficiency:** 1.00
- **Sparse efficiency:** 1.00
- **Temporal locality:** 2.63 (StdDev 0.72)
- **Spatial locality:** 1.36 (StdDev 0.62)

Mesh redundancy

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

- **Degenerate primitives:** 0
- **Duplicate vertices:** 0
- **Vertex padding size:** 0 bytes

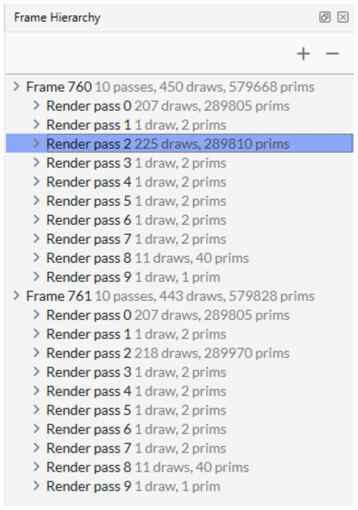
Mesh memory layout

Efficient meshes optimize their

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



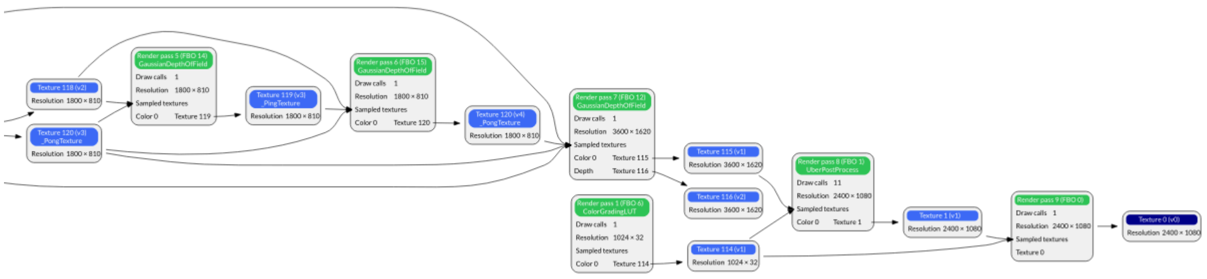
> Frame 760 10 passes, 450 draws, 579668 prims
> Render pass 0 207 draws, 289805 prims
> Render pass 1 1 draw, 2 prims
> Render pass 2 225 draws, 289810 prims
> Render pass 3 1 draw, 2 prims
> Render pass 4 1 draw, 2 prims
> Render pass 5 1 draw, 2 prims
> Render pass 6 1 draw, 2 prims
> Render pass 7 1 draw, 2 prims
> Render pass 8 11 draws, 40 prims
> Render pass 9 1 draw, 1 prim
> Frame 761 10 passes, 443 draws, 579828 prims
> Render pass 0 207 draws, 289805 prims
> Render pass 1 1 draw, 2 prims
> Render pass 2 218 draws, 289970 prims
> Render pass 3 1 draw, 2 prims
> Render pass 4 1 draw, 2 prims
> Render pass 5 1 draw, 2 prims
> Render pass 6 1 draw, 2 prims
> Render pass 7 1 draw, 2 prims
> Render pass 8 11 draws, 40 prims
> Render pass 9 1 draw, 1 prim

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

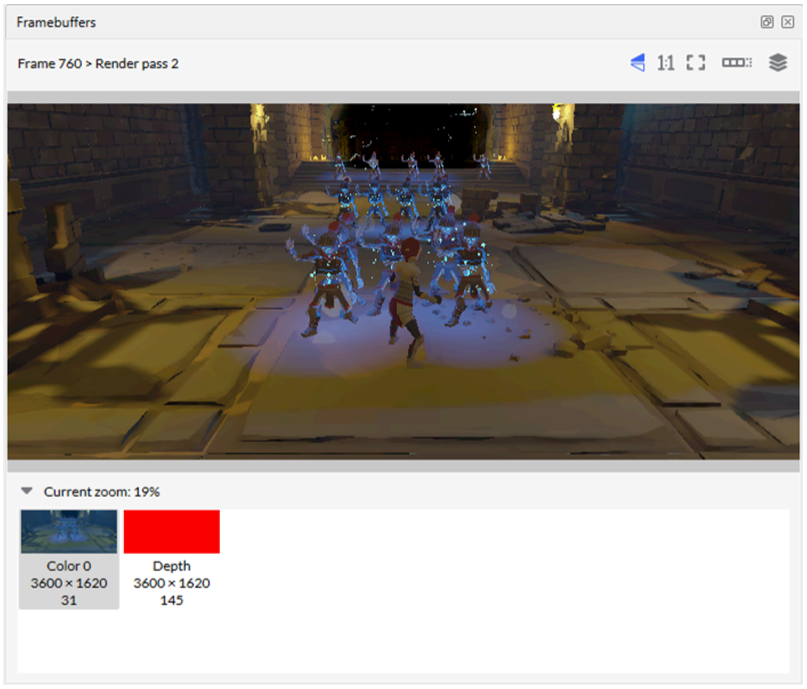


The dark blue node at the end of the process shows what is displayed on the screen of your device. Use the **Render Graph** to identify expensive parts of a frame that process workloads inefficiently, and find opportunities to optimize workload performance. See Analyze workloads using the render graph.

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 3-10: Framebuffers view

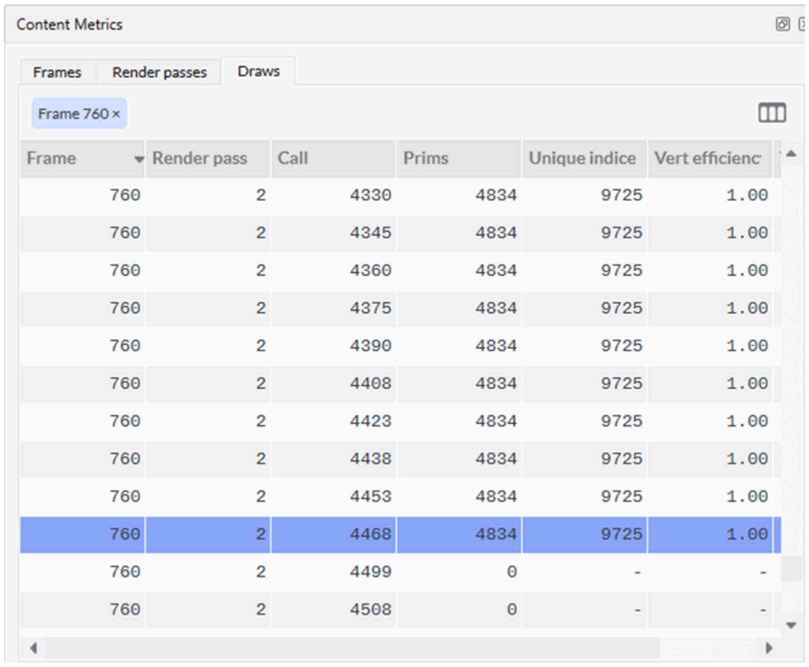


To help you analyze rendering efficiency, use the **Framebuffers** view to check object ordering, and levels of overdraw and shading. See Analyze object rendering.

Content Metrics view

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

Figure 3-11: Content Metrics view



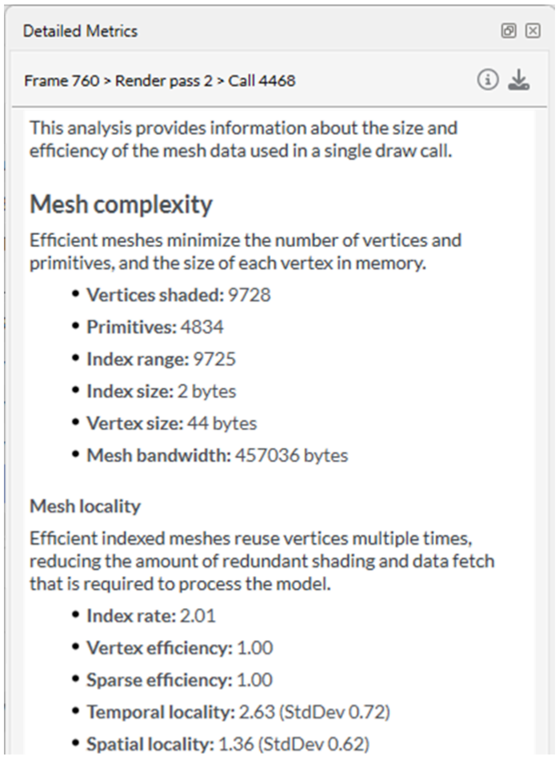
Frame	Render pass	Call	Prims	Unique indice	Vert efficienc
760	2	4330	4834	9725	1.00
760	2	4345	4834	9725	1.00
760	2	4360	4834	9725	1.00
760	2	4375	4834	9725	1.00
760	2	4390	4834	9725	1.00
760	2	4408	4834	9725	1.00
760	2	4423	4834	9725	1.00
760	2	4438	4834	9725	1.00
760	2	4453	4834	9725	1.00
760	2	4468	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 3-12: Detailed Metrics view



Detailed Metrics

Frame 760 > Render pass 2 > Call 4468

This analysis provides information about the size and efficiency of the mesh data used in a single draw call.

Mesh complexity

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

- **Vertices shaded:** 9728
- **Primitives:** 4834
- **Index range:** 9725
- **Index size:** 2 bytes
- **Vertex size:** 44 bytes
- **Mesh bandwidth:** 457036 bytes

Mesh locality

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

- **Index rate:** 2.01
- **Vertex efficiency:** 1.00
- **Sparse efficiency:** 1.00
- **Temporal locality:** 2.63 (StdDev 0.72)
- **Spatial locality:** 1.36 (StdDev 0.62)

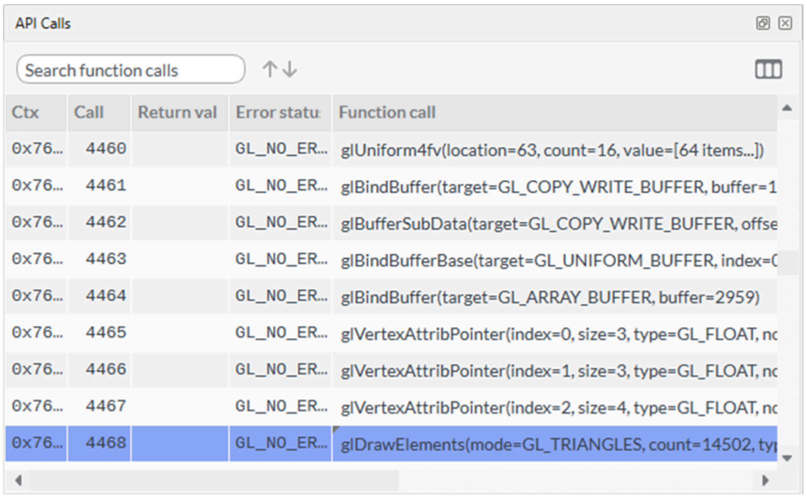
Complicated meshes containing many triangles are expensive for the GPU to process and, depending on the object's position and size on screen, may get no benefit from the added complexity. Reducing the number of triangles in a mesh dramatically reduces shader cost. If the object's silhouette is accurately 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 3-13: 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=(
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

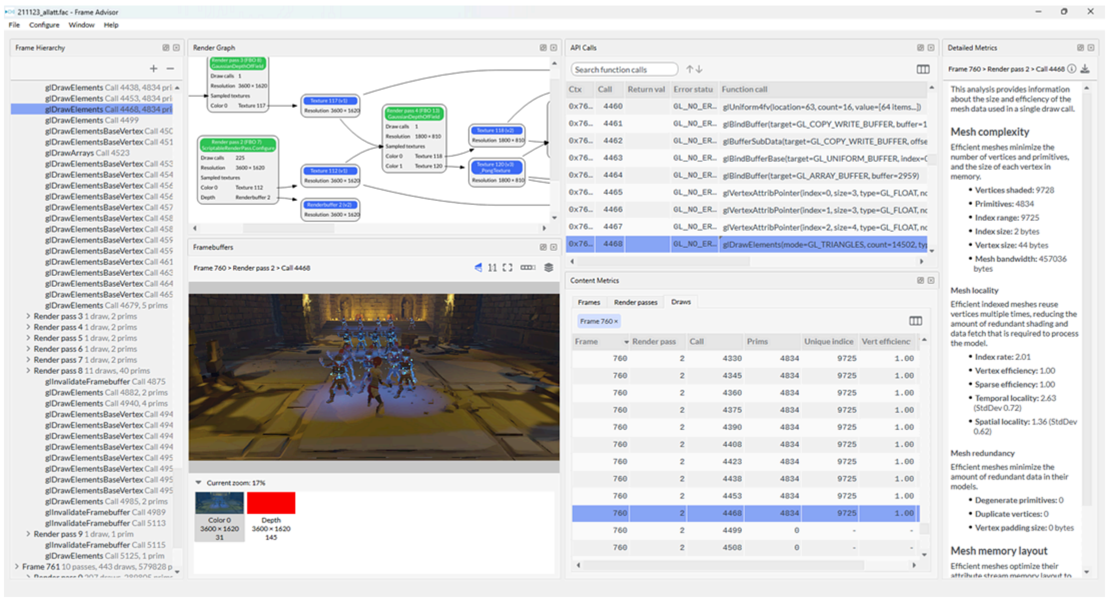
- Analyzing your frames
- How to get help

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



Frame Hierarchy:

```
glDrawElements Call 4438, 4834 pri
glDrawElements Call 4453, 4834 pri
glDrawElements Call 4468, 4834 pri
glDrawElements Call 4499
glDrawElementsBaseVertex Call 450
glDrawElementsBaseVertex Call 451
glDrawArrays Call 4523
glDrawElementsBaseVertex Call 453
glDrawElementsBaseVertex Call 454
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 456
glDrawElementsBaseVertex Call 457
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 458
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 459
glDrawElementsBaseVertex Call 461
glDrawElementsBaseVertex Call 463
glDrawElementsBaseVertex Call 464
glDrawElementsBaseVertex Call 465
glDrawElements Call 4679, 5 prims
> Render pass 3 1 draw, 2 prims
> Render pass 4 1 draw, 2 prims
> Render pass 5 1 draw, 2 prims
> Render pass 6 1 draw, 2 prims
> Render pass 7 1 draw, 2 prims
> Render pass 8 11 draws, 40 prims
    glInvalidateFramebuffer Call 4875
    glDrawElements Call 4882, 2 prims
    glDrawElements Call 4940, 4 prims
    glDrawElementsBaseVertex Call 494
    glDrawElementsBaseVertex Call 494
    glDrawElementsBaseVertex Call 494
    glDrawElementsBaseVertex Call 494
    glDrawElementsBaseVertex Call 495
    glDrawElementsBaseVertex Call 495
    glDrawElementsBaseVertex Call 495
    glDrawElementsBaseVertex Call 495
    glDrawElements Call 4985, 2 prims
    glInvalidateFramebuffer Call 4989
    glInvalidateFramebuffer Call 5113
> Render pass 9 1 draw, 1 prim
    glInvalidateFramebuffer Call 5115
    glDrawElements Call 5125, 1 prim
> Frame 761 10 passes, 443 draws, 579828 p
```

API Calls:

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=(
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, typ

Content Metrics (Draws):

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	-	-

Detailed Metrics:

This analysis provides information about the size and efficiency of the mesh data used in a single draw call.

Mesh complexity

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

- **Vertices shaded:** 9728
- **Primitives:** 4834
- **Index range:** 9725
- **Index size:** 2 bytes
- **Vertex size:** 44 bytes
- **Mesh bandwidth:** 457036 bytes

Mesh locality

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

- **Index rate:** 2.01
- **Vertex efficiency:** 1.00
- **Sparse efficiency:** 1.00
- **Temporal locality:** 2.63 (StdDev 0.72)
- **Spatial locality:** 1.36 (StdDev 0.62)

Mesh redundancy

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

- **Degenerate primitives:** 0
- **Duplicate vertices:** 0
- **Vertex padding size:** 0 bytes

Mesh memory layout

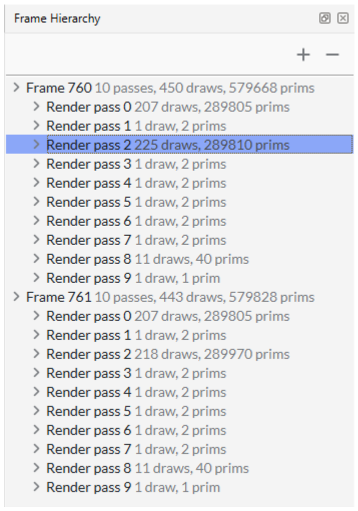
Efficient meshes optimize their

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.

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

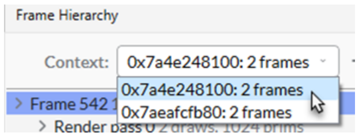


Frame Hierarchy

```
> Frame 760 10 passes, 450 draws, 579668 prims
  > Render pass 0 207 draws, 289805 prims
  > Render pass 1 1 draw, 2 prims
  > Render pass 2 225 draws, 289810 prims
  > Render pass 3 1 draw, 2 prims
  > Render pass 4 1 draw, 2 prims
  > Render pass 5 1 draw, 2 prims
  > Render pass 6 1 draw, 2 prims
  > Render pass 7 1 draw, 2 prims
  > Render pass 8 11 draws, 40 prims
  > Render pass 9 1 draw, 1 prim
> Frame 761 10 passes, 443 draws, 579828 prims
  > Render pass 0 207 draws, 289805 prims
  > Render pass 1 1 draw, 2 prims
  > Render pass 2 218 draws, 289970 prims
  > Render pass 3 1 draw, 2 prims
  > Render pass 4 1 draw, 2 prims
  > Render pass 5 1 draw, 2 prims
  > Render pass 6 1 draw, 2 prims
  > Render pass 7 1 draw, 2 prims
  > Render pass 8 11 draws, 40 prims
  > Render pass 9 1 draw, 1 prim
```

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



Frame Hierarchy

Context: 0x7a4e248100: 2 frames

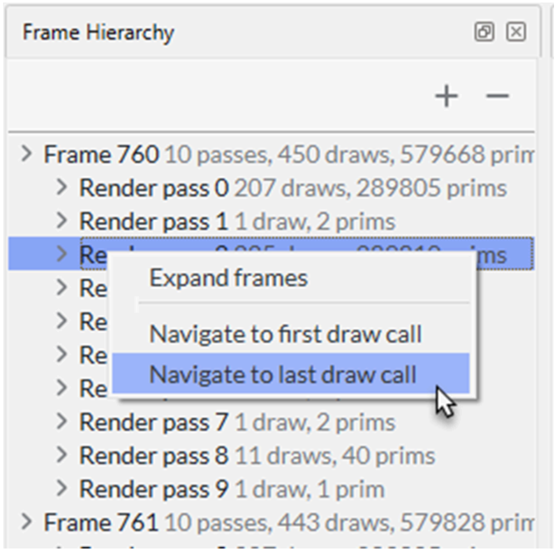
0x7a4e248100: 2 frames
0x7aeafcfb80: 2 frames

> Frame 542
> Render pass 0 2 draws, 1024 prims

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


Frame Hierarchy
Frame 760 10 passes, 450 draws, 579668 prim
Render pass 0 207 draws, 289805 prims
Render pass 1 1 draw, 2 prims
Expand frames
Navigate to first draw call
Navigate to last draw call
Render pass 7 1 draw, 2 prims
Render pass 8 11 draws, 40 prims
Render pass 9 1 draw, 1 prim
Frame 761 10 passes, 443 draws, 579828 prim

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 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

---

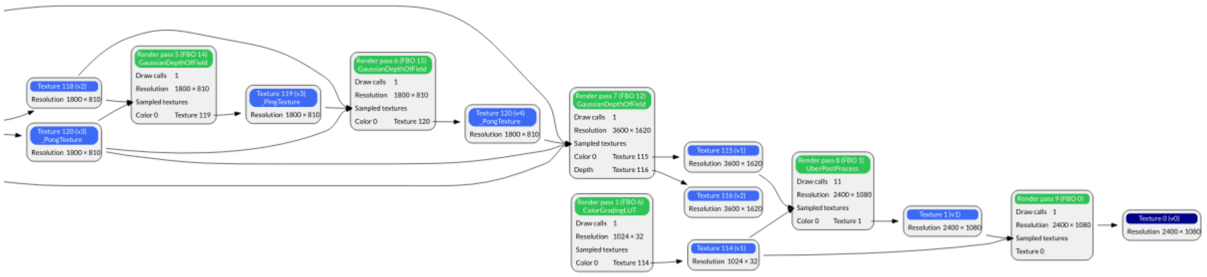
4.2 Analyze workloads using the Render Graph

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 find opportunities to optimize 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.

Tip

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:

a. Index of the render pass, and the framebuffer object that it relates to.

b. Number of draw calls in the render pass.

c. Rendered resolution

d. Input and output attachments

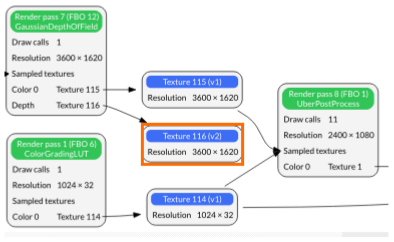
3. Click a render pass to navigate to it and update information shown in the other views of the **Analysis** screen.
4. Look for render passes that are consuming bandwidth and accessing memory unnecessarily, such as:

    - 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 memory 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 at the start of a render pass in the **Frame Hierarchy** view.

- Render passes that do not invalidate output attachments at the end of the render pass. Writing these output attachments to memory 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 can be 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. These attachments can be discarded at the end of the render pass so that they do not consume memory bandwidth.

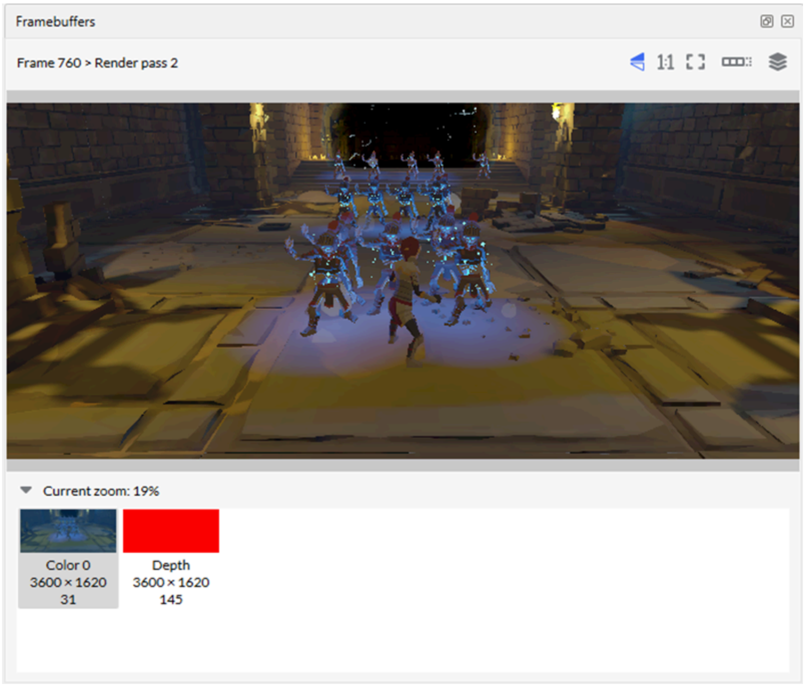
4.3 Analyze object rendering

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 find opportunities to optimize their performance.

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.

Figure 4-7: Framebuffers view



Tip

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

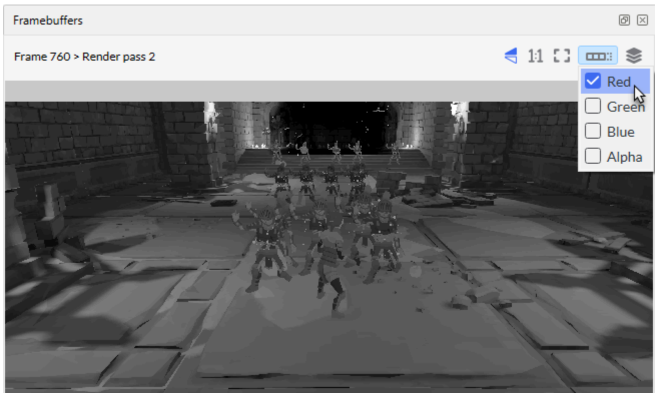
2. Expand a render pass in the **Frame Hierarchy** view, then use the arrow keys or mouse to step through each draw call in sequence.
   See how your frame is composed by watching how the framebuffer changes as you step through the draw calls. Any color, depth and overdraw attachments used in the framebuffer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

Tip

You can zoom and pan around the visualization to see more detail. Use the rescale buttons to get back to the original size image, or fit the image to the window.

3. 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 which channels that you want to show.

Figure 4-8: 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.

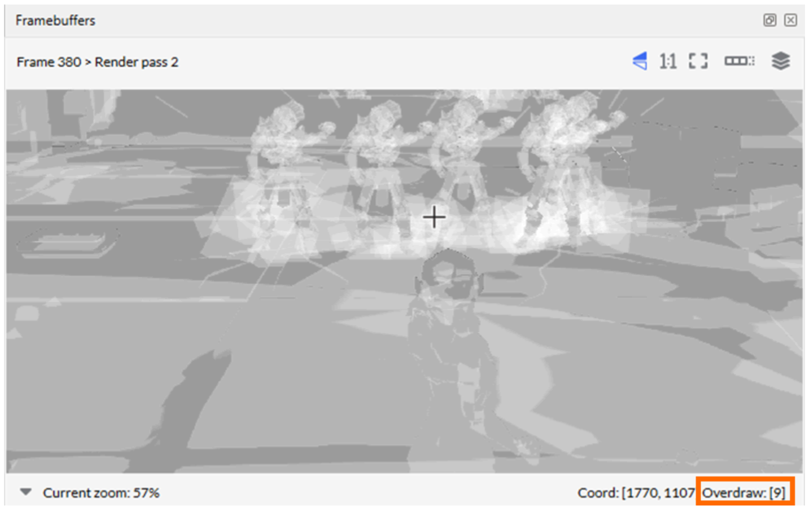
4. Check the ordering of opaque objects. To avoid high levels of overdraw, ensure the objects are rendered in order from front-to-back, starting with objects closest to camera first. This order enables the GPU to use early ZS testing to prevent shading of occluded objects, and eliminating unnecessary work before fragment shading.

Tip

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

If you captured frames using **Overdraw mode**, click the overdraw thumbnail to show it in the visualization area, then move the cursor around to see the levels of overdraw. The more times that a pixel is rendered, the higher the overdraw value and the whiter it appears in the final image.

Figure 4-9: Framebuffers overdraw view



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

5. 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.
6. 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.
   Processing individual objects is costly. To help improve performance, merge the identical objects into a single draw call to reduce the draw call count.
7. As you step through the draw calls, check if the complexity of objects is appropriate for their size on screen.
   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. 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 for more information.

Next steps

Investigate the efficiency of your mesh further in the **Detailed Metrics** view. See Content metrics in detail for more information.

4.4 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 function call has created the draw.

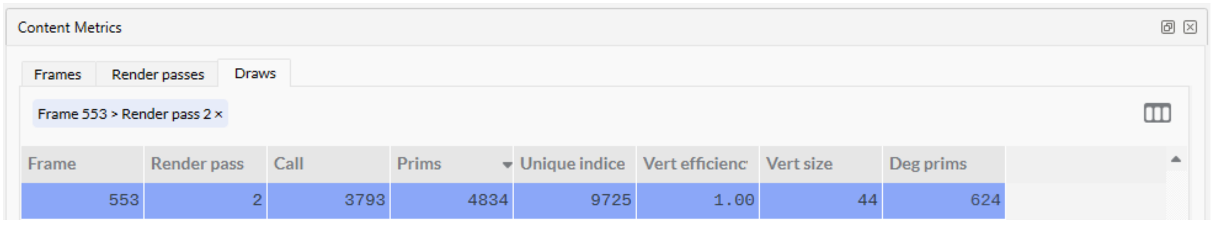
About this task

Here is an example of how you could use the **Content Metrics** view to identify degenerate primitives.

Procedure

1. To add and remove columns of data in the **Content Metrics** view, navigate to the **Draws** tab, then click the **Edit columns** icon ▥. In this example, we are adding a column to show degenerate primitives.
2. To sort the data and arrange the draw calls containing the highest number of degenerate primitives at the top of the table, click the **Degenerate primitives** column heading.
3. Select a draw call that has information you want to investigate.

Figure 4-10: 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 see data for all the frames in your trace, click the filters to clear them.

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

4.5 Content metrics in detail

The **Detailed Metrics** view helps you to understand the cost of meshes, identify inefficiencies, and find optimization opportunities. The recommended mesh memory layout splits the input attributes into two streams. One for the attributes needed to compute position, and one for the remaining attributes. It 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**.

4.5.1 Mesh complexity

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

Shaded vertices
: 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.

4.5.2 Mesh locality

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

Index rate
: The number of unique indices per primitive.

The Index rate is a measure of how vertices are shared across multiple primitives. To reduce your Index rate, reuse vertices as much as possible.

Vertex efficiency
: The ratio of vertex shader invocations to the number of unique indices in the index buffer.

Arm® GPUs always shade indices in groups of 4 consecutive indices, even if those indices are not used in the model. Values less than 1 indicate that the model is overshading because of unused indices in these groups of 4. Minimize overshading by tightly packing meshes to ensure all vertices between the minimum and maximum index value are used.

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 consecutive indices, even if those indices are not used in the model. Values less than 1 indicate that the model is overshading because of unused indices in these groups of 4. Minimize overshading by tightly packing meshes to ensure all vertices 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.

Reducing the average temporal locality improves cache efficiency of the post-transform cache during vertex shading. To lower your average temporal locality, reduce the number of indices between reuse of an index value.

Arm® Frame Advisor models a 1024-byte entry post-transform cache for computing vertex reshading. Aim to keep temporal locality under 1024 bytes. Under 512 bytes is ideal.

Average spatial locality

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

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

4.5.3 Mesh redundancy

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

Instance degenerate primitives

The number of primitives that have zero area.

A significant percentage of degenerate primitives may indicate a mesh encoding issue. If you are using degenerate primitives for encoding spatial jumps in the mesh, try using primitive restart instead.

Duplicate vertices

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

Duplicate vertices are caused by model reuse when only some attributes are used. To prevent duplicate vertices wasting bandwidth, remove as many duplicate vertices as possible. Also try specializing meshes to see if the bandwidth saving is worth the increased memory footprint.

Vertex padding size

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

Padding is caused by model reuse, when only some attributes are used. Padding in the same cache line as used data is still fetched from memory, so aim to make vertex padding size as as small as possible. Also try specializing meshes to see if the bandwidth saving is worth the increased memory footprint.

4.5.4 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 for the attributes needed to compute position, and one for the remaining attributes. 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.

4.6 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.

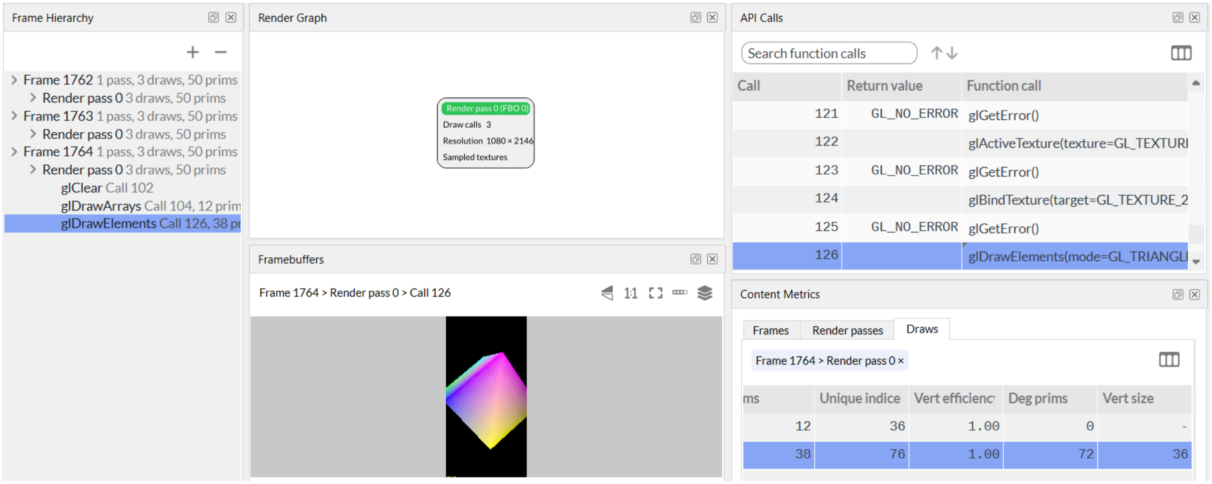
About this task

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 4-11: Draw calls selected



Frame 1762 1 pass, 3 draws, 50 prims
Render pass 0 3 draws, 50 prims
Frame 1763 1 pass, 3 draws, 50 prims
Render pass 0 3 draws, 50 prims
Frame 1764 1 pass, 3 draws, 50 prims
Render pass 0 3 draws, 50 prims
glClear Call 102
glDrawArrays Call 104, 12 prim
glDrawElements Call 126, 38 pr

Render pass 0 (FBO 0)
Draw calls 3
Resolution 1080 × 2146
Sampled textures

Call	Return value	Function call
121	GL_NO_ERROR	glGetError()
122		glActiveTexture(texture=GL_TEXTURI
123	GL_NO_ERROR	glGetError()
124		glBindTexture(target=GL_TEXTURE_2
125	GL_NO_ERROR	glGetError()
126		glDrawElements(mode=GL_TRIANGLI

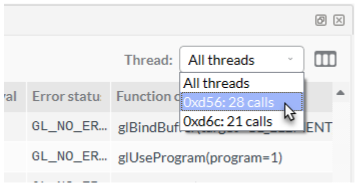
Frame 1764 > Render pass 0 > Call 126

Frame 1764 > Render pass 0 ×

ms	Unique indice	Vert efficienc	Deg prims	Vert size
12	36	1.00	0	-
38	76	1.00	72	36

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 4-12: 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® Mobile 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® Mobile Studio on the Developer website.
- To ask a question directly, you can email the Arm® Mobile Studio team at mobilestudio@arm.com.

  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 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.