***Introduction to Graphics and Mobile Gaming***

**LAB 6**

**Streamline and the Graphics Analyzer**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc36134501)

[2 Requirements 1](#_Toc36134502)

[3 Streamline with Graphics Analyzer integration 1](#_Toc36134503)

[3.1 Simple Triangle 1](#_Toc36134504)

[4 ChessRoom 4](#_Toc36134505)

[4.1 Streamline 4](#_Toc36134506)

[4.2 Graphics Analyzer 6](#_Toc36134507)

[4.2.1 Frame buffer attachment 6](#_Toc36134508)

[4.2.2 Overdraw mode 8](#_Toc36134509)

[4.2.3 Render pass dependencies 9](#_Toc36134510)

[5 Optimization tips 9](#_Toc36134511)

[5.1 Geometry 9](#_Toc36134512)

[5.2 Texture 10](#_Toc36134513)

[5.3 Changes 10](#_Toc36134514)

[6 Conclusions 12](#_Toc36134515)

# Introduction

In this lab, we will explore functionalities of the tools in the Arm Mobile Studio. We will examine performance analysis with Streamline and best practices for graphics development in OpenGL ES using the Graphics Analyzer.

# Requirements

You are free to explore the apps that you have built in the previous labs of the course, but analysis trace files are provided in the LabResources folder. You can access the directory and import them in any of the tools.

To get any of the installed apps analyzed, please follow the instructions in the Getting started guide. Before you get any Unity app analyzed, make sure to go to ***Project Settings-> Player*** and select the graphics API to be ***OpenGL ES*** not Vulkan.

# Streamline with Graphics Analyzer integration

We will analyze some samples using the GA integration with Streamline.

## A screenshot of a computer Description automatically generatedSimple Triangle

Import the trace file for Simple Triangle. This is the most basic app that displays a red triangle on the screen.

We can see the full system image in Streamline.

As you can see, the view is too coarse to notice anything relevant on a low level, so you will want to zoom in and examine individual frames, which are usually 16-32 milliseconds in length. You can enable annotations for Streamline using a script inside the app build, which will display coloured markers at the end/beginning of every frame.

For OpenGL ES applications, you can display the annotations whenever the application calls eglSwapBuffers(). For more information on annotations, please refer to the User Guide **Chapter 9 Code Annotations.**

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedIt’s recommended to switch the template for the specific GPU on the device. Also, because we have integrated GA, we can change the HeatMap at the bottom of the screen to a Graphics Analyzer view that shows frames and render passes.

After zooming in, we can see a more detailed view.

* A screen shot of a computer

  Description automatically generatedThe first batch of charts displays **CPU Activity**. The first CPU doesn’t seem idle for too much, and the load is not big, while the second CPU is idle for most of the time. We’d expect not to see a lot of load from such a simple app.
* We can see one **draw call/frame** registered; we’re only drawing one triangle every time. If there’s a spike in draw calls, we should investigate that frame, maybe it’s drawing too many objects.
* A screenshot of a computer

  Description automatically generatedThe **framerate** is around 60 fps, the usually expected number. Frame rate is a very useful information to analyze. If we can see drops in frame rate, it means something might be wrong with our application and the workload it generates. Ideally, we’d like to optimize any game to run at 60 fps
* Further we can see the **Mali Job Manager Cycles** that displays the number of GPU cycles spent with running content. The **Mali Job Manager Utilization** shows the same data but normalized as a percentage against CPU active cycles.

If the content is GPU bound, the dominant work queue should be active all of the time, with other queues running in parallel to it.

Here, we can observe if an app if the app is not achieving good parallelism. In that case, one should check for API calls that drain the rendering pipeline, such as *glFinish()* or synchronous use of *glReadPixels().*

* The **IRQ active** is another good indicator to look at as it shows the number of cycles the GPU has an interrupt pending with the CPU. A normal IRQ pending rate is of ~2%, but a higher interrupt rate can indicate a queue of many small render passes or compute dispatches. It can also be a sign of a system integration issue. Unfortunately, it’s not often possible to fix high IRQ overhead using application modifications.
* Moving on to **Mali External Bus Bandwidth** charts. This shows the total bandwidth generated by the application. Reducing memory bandwidth is always an effective app optimization goal and can be monitored here.
* **A screenshot of a video game

  Description automatically generatedMali External Bus Stall Rate**: This shows how much back-pressure the GPU is getting from the external memory system. Normal rates are considered up to 5%, a higher rate indicates more traffic than the memory can handle.
* GPU geometry is another useful feature to look at; you can observe the amount of geometry being processed by the GPU and how much of it gets culled in the process.
* The ***Mali Primitive Culling*** charts represents the absolute number of primitives being processed and show how many of them get culled in which culling stage. For optimization, one should always aim at reducing the total number of primitives per frame.
* The ***Mali Primitive Culling Rate*** shows percentage of primitives entering each culling stage. The culling process is sequential and follows a series of stages:

A close up of a sign

Description automatically generated

* + In 3D scenes, we’d expect to have ~50% back-facing primitives, which get killed by the *facing culling* test. Here, we can’t see any of these because our app is 2D, drawing one triangle. We can, however, see this behaviour in the 3D Unity app, which we will discuss later.
  + Standard best practice indicates for an application to cull out-of-frustrum draw calls on the CPU, so *the culling by frustrum test* should be kept preferably under 10%.
  + *Culling by sample test* indicates percentage of primitives killed because they do not hit rasterization sample points usually they are too small. This happens in dense geometry like objects with too many vertices. Preferably, this percentage will be kept as close to 0% as possible. If that doesn’t happen, one needs to review the level of detail of objects in game.

A close up of a sign

Description automatically generated

* The ***Mali Pixels*** chart depicts the total number of pixels shaded by all shader cores, which gives a good indication of the number of pixels required to produce a frame.
* The ***Mali Overdraw*** chart shows the average number of fragments shaded per output pixel. Overdraw often brings issues in performance, so high levels of overdraw should be investigated further using the Graphics Analyzer.
* A picture containing screenshot

  Description automatically generatedIf we switch back to the *Default Template,* we can see two other useful charts generated by the Graphics Analyzer: ***Vertices*** and ***Vertices/Frame***. For our simple app, we can see that it maintains a constant 3 vertices/frame, just like we’d expect from a triangle. If we can detect frames with an excess of vertices, it’s worth investigating the meshes design and the level of detail on them.

# ChessRoom

We shall now take a look at the Unity Chess Room app using both Streamline and the Graphics Analyzer.

## Streamline

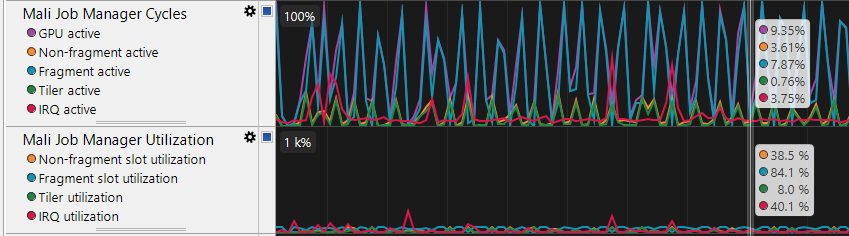
Import the file from the LabResources called “ChessRoomStreamline” into Streamline.

A picture containing electronics

Description automatically generatedAs you can see, the first ~6s of the trace are very different from the rest of the readings, that is, because if you launch the app, you will notice Unity displays its own logo in an animation in the beginning. Therefore, these 6s are irrelevant to us.

* Take a look at the ***CPU load***. We can see it doesn’t stay inactive for long period of the frame, and it doesn’t have a suspicious continuous increase either.
* A picture containing screenshot

  Description automatically generatedWe see an almost constant number of draw calls per frame, objects don’t get added or destroyed in the game, so there shouldn’t be more or less draw calls.
* The frame rate drop in the beginning is registered at the moment where the content is loaded and stabilizes afterward at around 60 fps.



* A screen shot of a computer

  Description automatically generatedNow let’s look at the GPU workload. The ***IRQ*** is higher than the ideal 2%, but we can’t be sure of a cause.
* The charts above are related to the ***GPU memory system***. We notice a higher number of read and writes in the early frames of the app. Stalls rate go up to around 2%, so they’re in normal limits.
* Mali GPUs are designed for an external memory latency for up to 170 cycles. The highest percentage of reads seems to be in the 0-127 range and very few in the slower bins here, which is a good indicator.
* As mentioned before, in culling tests, we’d expect to see around 50% of primitives culled by the facing test. We have a percentage of around 38% here. The frustrum calling percentage is a bit higher than we’d like. On the other hand, sample culling is really low, which indicates a good level of detail in objects.
* ***Mali Pixels***: We can see a considerable increase in pixel numbers from the basic SimpleTriangle app, which is expected.

A picture containing object

Description automatically generated

* ***A picture containing indoor

  Description automatically generatedWarp divergence rate*** displays the percentage of instructions executed when there is control flow divergence across the warp. Minimizing control flow divergence is always a good target as this can affect shader execution efficiency.
* From the block of Texture charts, the interesting one is ***the Mali Core Texture Usage Rate,*** which reports statistics about the type of textures being made:
  + Mipmapped texture percentage is the highest here, which is a good indicator. 3D scenes should always use mipmapped texture as they improve performance and image quality.
  + Compressed textures reports formats such as ASTC and ETC (ASTC preferred over ETC). Using block compressed textures reduces bandwidth.
  + Trilinear filtered access reports percentage of texture samples using trilinear filtering, which run at half the rate of bilinear accesses.

## Graphics Analyzer

Please follow the instructions for getting started with GA for Unity. Connect your phone and start tracing the app.

A close up of a logo

Description automatically generatedYou’ll need to pause it and capture frames individually to get more insight data. Capture one of each type of feature to see step by step through the rendering process. Select a feature to capture and then click the camera button.

In the Trace Outline window, filter the frames to Show Only Frames With Features Enabled.

### A close up of a logo Description automatically generatedFrame buffer attachment

A screenshot of a cell phone

Description automatically generatedGA displays all the render passes and draw calls in a frame. If you go throw the draw calls step by step, you’ll be able to see all elements that are being drawn in your mobile game.

A close up of a mans face

Description automatically generatedTake this RenderPass1 draw calls as example:

A screenshot of a computer

Description automatically generated

You can see the geometric object here, how many vertices and triangle it’s made of.



You can see the object being drawn in the scene here.



A screenshot of a computer

Description automatically generated

Depth and stencil maps of the image.

This draw call seems to have a lot of vertices, so we can investigate the statistic charts as well:

A screenshot of a cell phone

Description automatically generatedThis specific draw call accounts for almost half of the total vertices in the render pass (17952 vertices with 3673 unique indices that lead to around 6000 triangles in the mesh). We might want to reconsider if this mesh is worth taking up that much computational power. However, it is a big and central piece of the game, so it’s not responsible for micro-triangles.

### A close up of a logo Description automatically generatedOverdraw mode

The overdraw mode analyzes how many times each pixel is rendered to the framebuffer. You will notice that the fragment shader is replaced with a gray and almost transparent fragment shader. For each pixel, every time it’s rendered, the alpha value is increased. So, the areas that appear more white have more overdraw.A screenshot of a cell phone

Description automatically generated

You can go through draw calls step by step and analyze which ones cause the overdraw. You will immediately notice that when it renders the 3 chess pieces in the scene (each with 2004 vertices), the game renders the whole screen instead of just the local object. So, there might be something wrong to elements related to the chess piece, such as texture, shadows, shader.

Always check for UI elements such as text or buttons. It’s a good practice to have them all on the same Canvas. That way, there is only one render for the whole screen for that Canvas, not multiple full screen draw calls.

### A screenshot of a social media post Description automatically generatedRender pass dependencies

In a captured frame, you can also generate the render pass dependencies.

Reducing the number of render passes is always a good thing because it reduces memory accesses. If we take a closer look here at what our render passes do, we will notice that the last render pass of every frame doesn’t have anything to do with the rendering of objects in the game.

Unity uses a setting called Blit, which means that the final output gets rendered to a frame buffer and then gets copied (or blitted) to the screen of the device. You can manually set the ***Blit Type*** to **Never** from Unity, which means everything will get directly rendered to the screen and cause the last render pass in every frame to disappear. This might bring an increase in performance; however, direct rendering is not compatible with linear colorspace, MSAA (Multisampling anti-aliasing in HDRP) or non-native screen resolutions.

# Optimization tips

Our chess room application was built with these best practices in mind, but there are still some adjustments that we can do.

## Geometry

* Don’t use more triangles than needed. High detailed object seems tempting for the game quality, but often details might not even be visible on a phone screen.
* Use more triangles for objects closer to the camera and as little as possible for the objects far from the camera. Unfortunately, there isn’t an exact number for maximum triangle count per model. This can depend on many factors, such as how many objects are in your field of view. Having lots of high triangle count objects will impact negatively on performance. In our chess room, we have one very detailed object in the centre of the screen (the phoenix mesh) and 3 chess pieces. It’s affordable to have a quite high triangle count.
* Small triangle details on 3D objects are often not even visible in a game because of the device’s screen size of the position of the object in game. Details can be better achieved using Normal Maps and Textures to show details.
* Use LOD (Level Of Detail). This technique consists of reducing the number of triangles rendered for an object as it gets further away from the camera. It will help avoid micro-triangles and optimize the rendering process.
* Do not use LOD when the objects and the camera view are static.

Read more on geometry best practice here : <https://developer.arm.com/solutions/graphics/developer-guides/best-practices-for-mobile-game-art-assets-1/geometry-best-practices/single-page>

## Texture

* Use texture atlas: an image consisting of multiple smaller images packed together. Instead of using one texture for a single mesh, use an atlas shader between multiple meshes. Using atlases will reduce the total number of in-game textures and will enable batching on static objects that share this texture, which will reduce the number of draw calls.
* Use texture filtering: it will improve texture quality in game scene:
  + Nearest/Point filtering: up close, the texture will look blocky.
  + Bilinear: texture is blurry up close. Use this to achieve a balance between performance and visual quality.
  + Trilinear: same as bilinear, and it also adds blur between the mipmap level and provides a smooth transition. This will cost more memory bandwidth than bilinear.
  + Anisotropic: textures look better when viewed from different angles. This is recommended for ground-level textures.

A combination of Bilinear+ 2xAnisotropic will generally perform and look better than Trilinear+ 1xAnisotropic. It’s recommended to keep to Anisotropic level low (max 2) because it costs a lot of bandwidth and will affect battery life.

* Use mipmapping: controlling the level-of-detail of objects. Mipmaps are lower resolution copies of the original texture. Depending on how close/far an object is from the camera, a lower/higher resolution for texture is selected.
* Texture size: not all textures need to be the highest size.
* Texture colour space: diffuse textures should use sRGB colour space, while textures that are not processed as colours should not use sRGB (such as metallic, roughness, normal maps). Using sRGB on these textures will cause issues such as wrong reflections on the material.
* Use texture compression: in development, textures are exported as PNG or TNA formats, but in the final rendering process, these are slow to process, so we can use ASTC,ETC1,ETC2. The recommended format is ASTC. It takes longer to encode than ETC, but it achieves the same or better quality with less memory size.

For more texture best practices, please visit: <https://developer.arm.com/solutions/graphics/developer-guides/best-practices-for-mobile-game-art-assets-1/textures-best-practices/single-page>

## Changes

Throughout the labs, we introduced texture atlases, compression, and mipmapping into our application.

1. One thing we can still apply is texture compression ASTC. This can be done from ***Unity->Build Settings***.
2. In 4.3.2, we introduced the notion of Android Blit. Please navigate to ***Project Settings*** and set it to **Never**.
3. We can also turn off casting and receiving shadows on the game object because we have our own separate script to generate shadows. This should reduce draw calls.
4. Inspect the shaders in the game and make sure they’re set as Mobile shaders; this leads to less fragment work.

A screenshot of a computer

Description automatically generatedWe’ll now take a look at Streamline before and after we integrated the changes to see how performance changed (you can find the second trace file under the name).

Left: initial app Right: optimization changes

* There is no significant change in the CPU load, but the frequency increased.
* The number of draw calls is constantly lower than before (36 vs 41).
* Overall GPU activity decreased, along with fragment activity.
* IRQ values increased; this might have to do with disabling the Blit, which can sometimes lead to compatibility issues.
* A circuit board

  Description automatically generatedThere’s significantly less Bus Read latency and Outstanding Reads and Writes.
* Primitives culled by the facing test increased to ~50%, the number we would have liked to see on 3D applications.
* Because there are less visible primitives, the number of pixels decreased as well.
* Overall GPU activity is much lower, bandwidth decreased significantly, and the quality of the game did not get affected. This will eventually lead to much better battery life.

Take a look at the Graphics Analyzer and inspect the missing draw calls. Because we set the BlitType to Never, the 3rd render pass of every frame will have disappeared as well.

# Conclusions

The tools in the Arm Mobile Studio offer a very well detailed image of the application you’re trying to analyze. They don’t give indications on how to fix issues in your app, but they do offer enough information for you as a developer to figure that out. As seen, even just following the good practice tips can show a significant difference in GPU performance.

Lab content source: <https://community.arm.com/developer/tools-software/graphics/b/blog/posts/accelerating-mali-gpu-analysis-using-arm-mobile-studio?_ga=2.186417257.1548598520.1566203570-1867192701.1559896154>