

# Profiling and Debugging Games on Mobile Platforms

Lorenzo Dal Col  
Senior Software Engineer, Graphics Tools

*Gamelab 2013, Barcelona – 26<sup>th</sup> June 2013*



# Agenda

- Introduction to Performance Analysis with ARM<sup>®</sup> DS-5<sup>™</sup> and Streamline<sup>™</sup>
- Debugging with Mali<sup>™</sup> Graphics Debugger
  - Architecture
  - Features
- Working out Limiting Factor
  - CPU bound
  - Fragment Bound
  - Vertex Bound
  - Bandwidth Bound
- Identify the problem and find out how to fix it
- Vertex Buffer Objects
- Q & A

# ARM Mali Developer Center

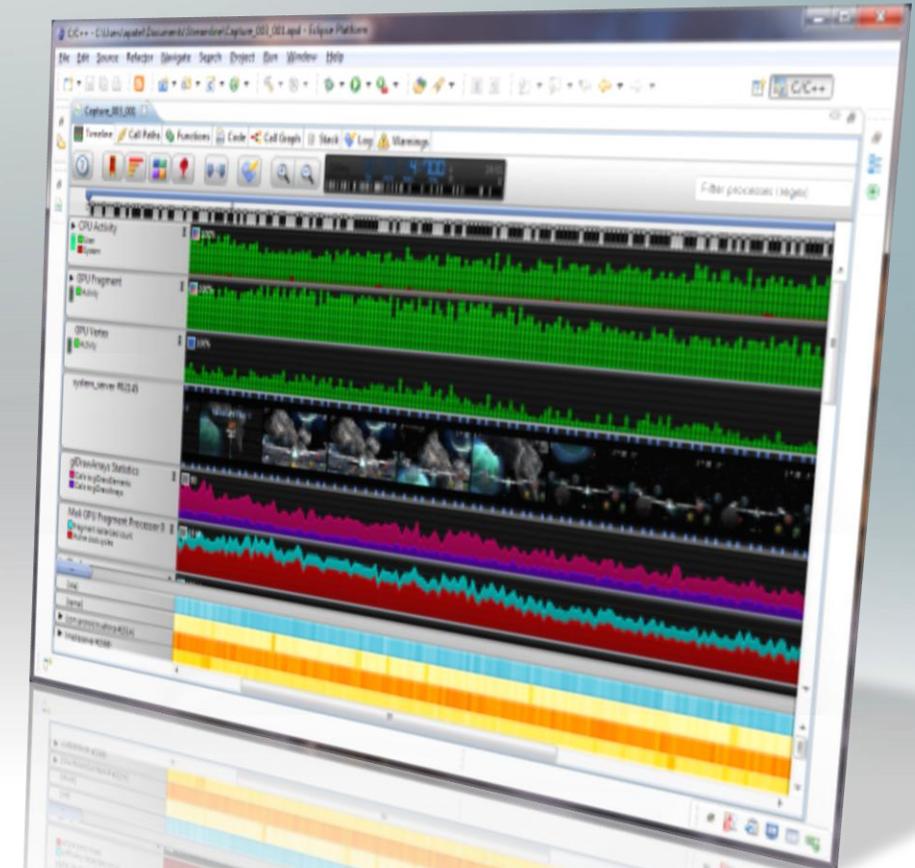
- One-stop-shop for developers developing on ARM Mali GPUs
- Rich set of development tools
  - Tools to enable system level approach for performance analysis and debug
  - Tools to enable pre-silicon software development
  - Tools to enable asset and content creation
- Direct access to ARM Mali experts
- Potential route to become an ARM partner and have demos showcased across ARM's channels into OEMs, Silicon companies and the market

[malideveloper.arm.com](http://malideveloper.arm.com)

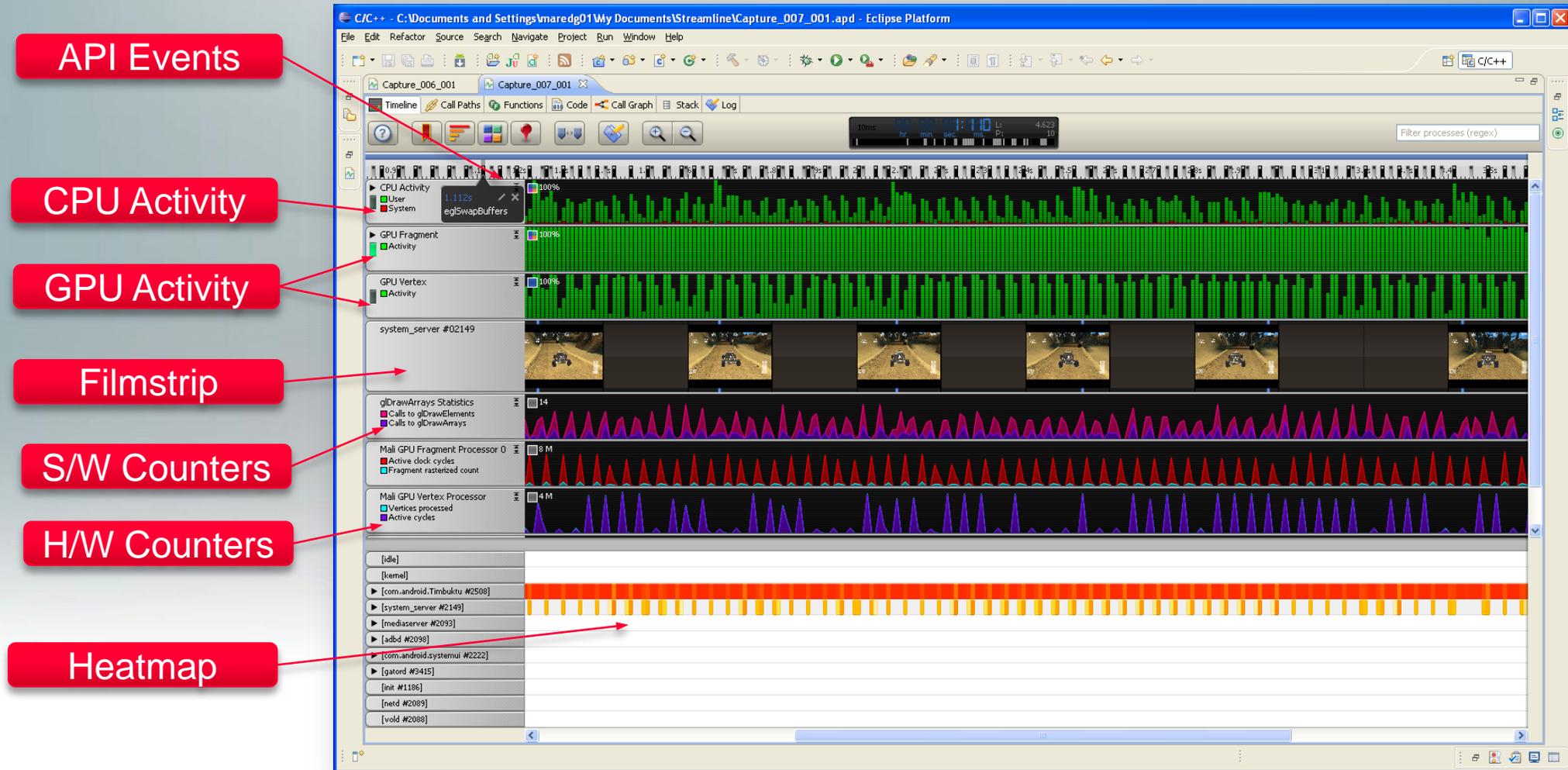


# Performance Analysis

- **ARM DS-5 toolchain** with support for ARM Mali GPUs
- **System wide performance analysis**  
Simultaneous visibility across ARM Cortex<sup>®</sup> processors & Mali GPUs
- **Optimize performance and power efficiency** of gaming applications across the system

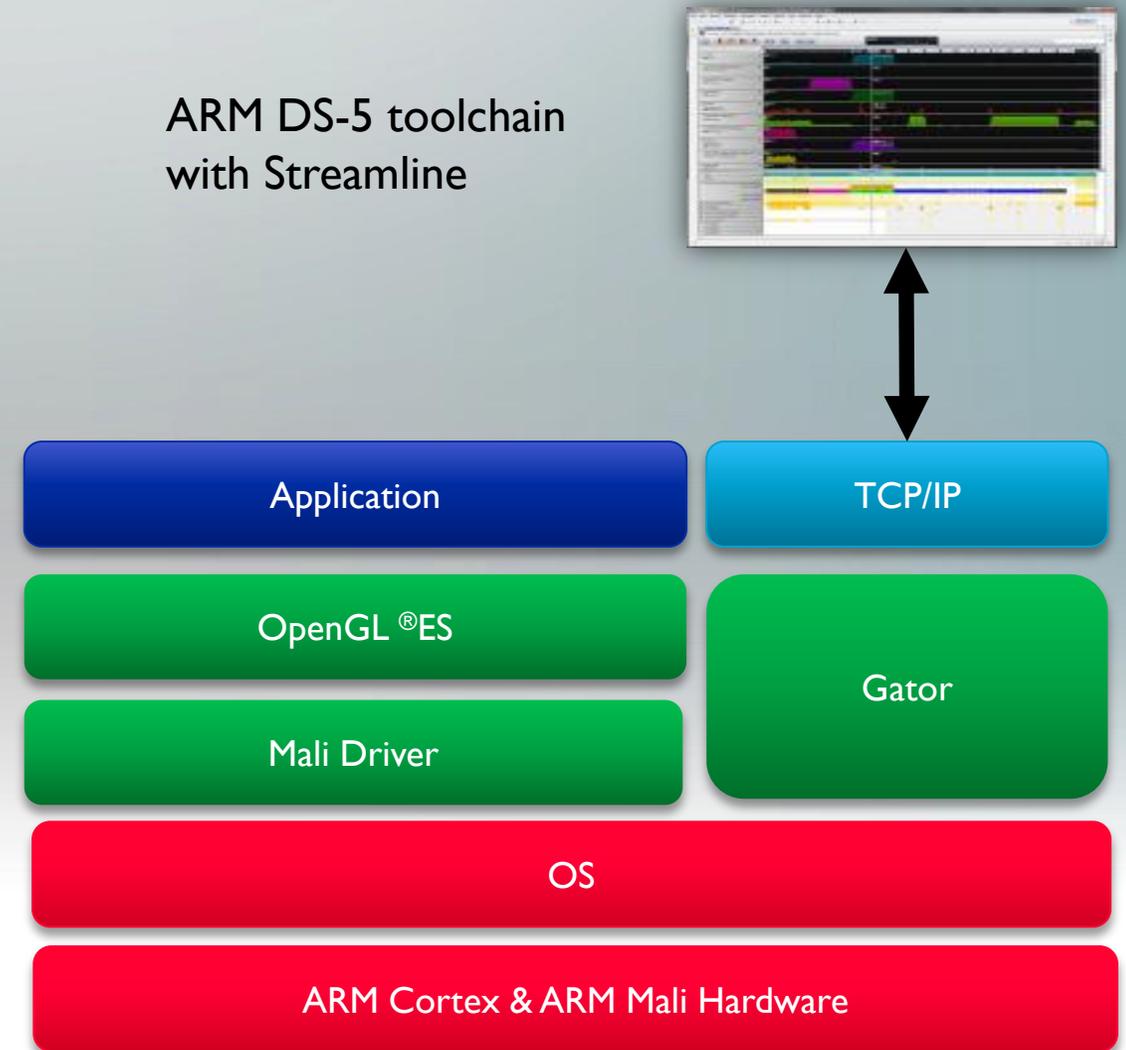


# Streamline Performance Analyzer

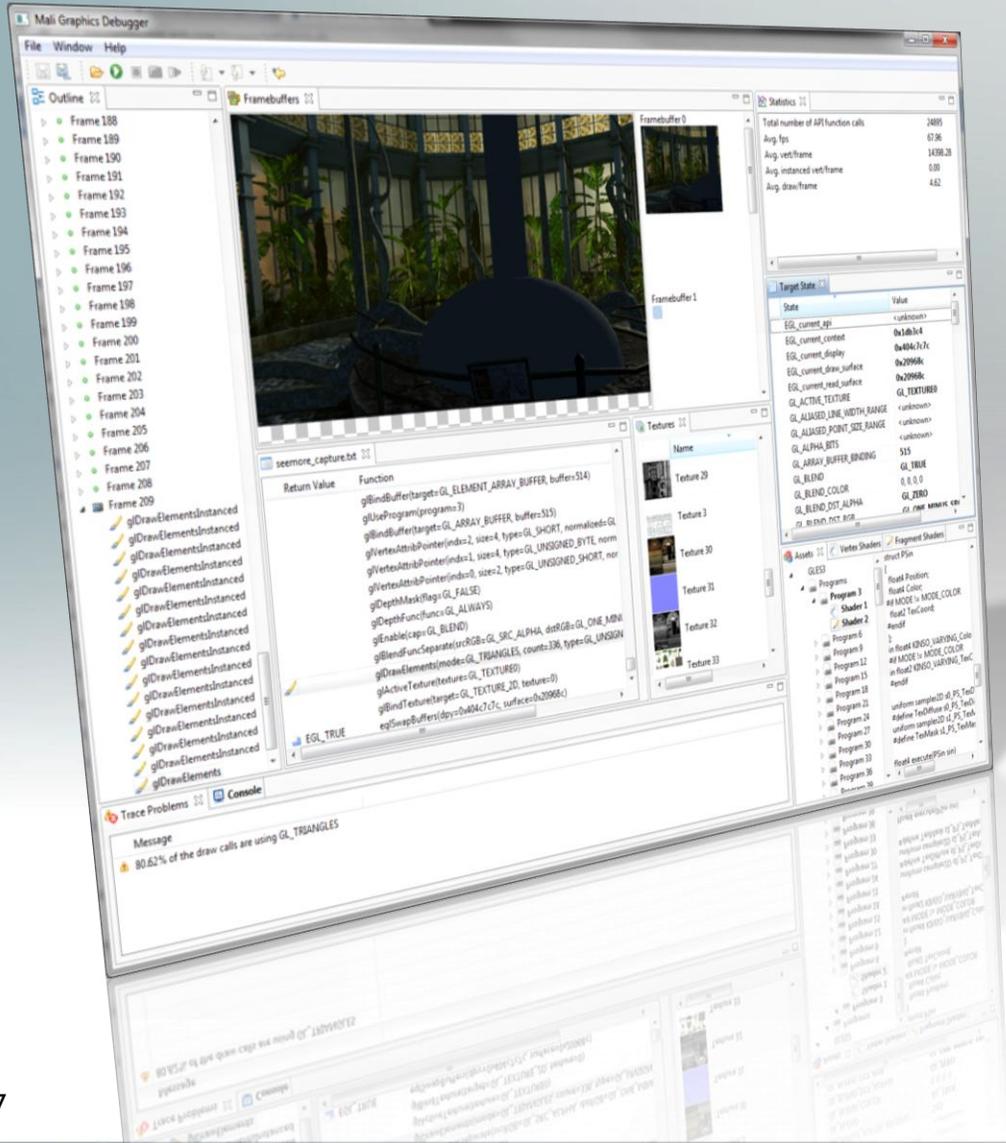


# Streamline Setup

- Gator interfaces with Kernel & Mali Drivers
  - Extracts H/W & S/W counters
  - Extract frame buffer
  - Passes through events & annotations
  - Transmit data over TCP/IP to DS-5 tools
- Transparent to user application
  - Option to add annotations to user application for more advanced debugging
- Negligible performance impact
  - Zero impact when profiling disabled
  - Minimal impact in performance when enabled



# Debug

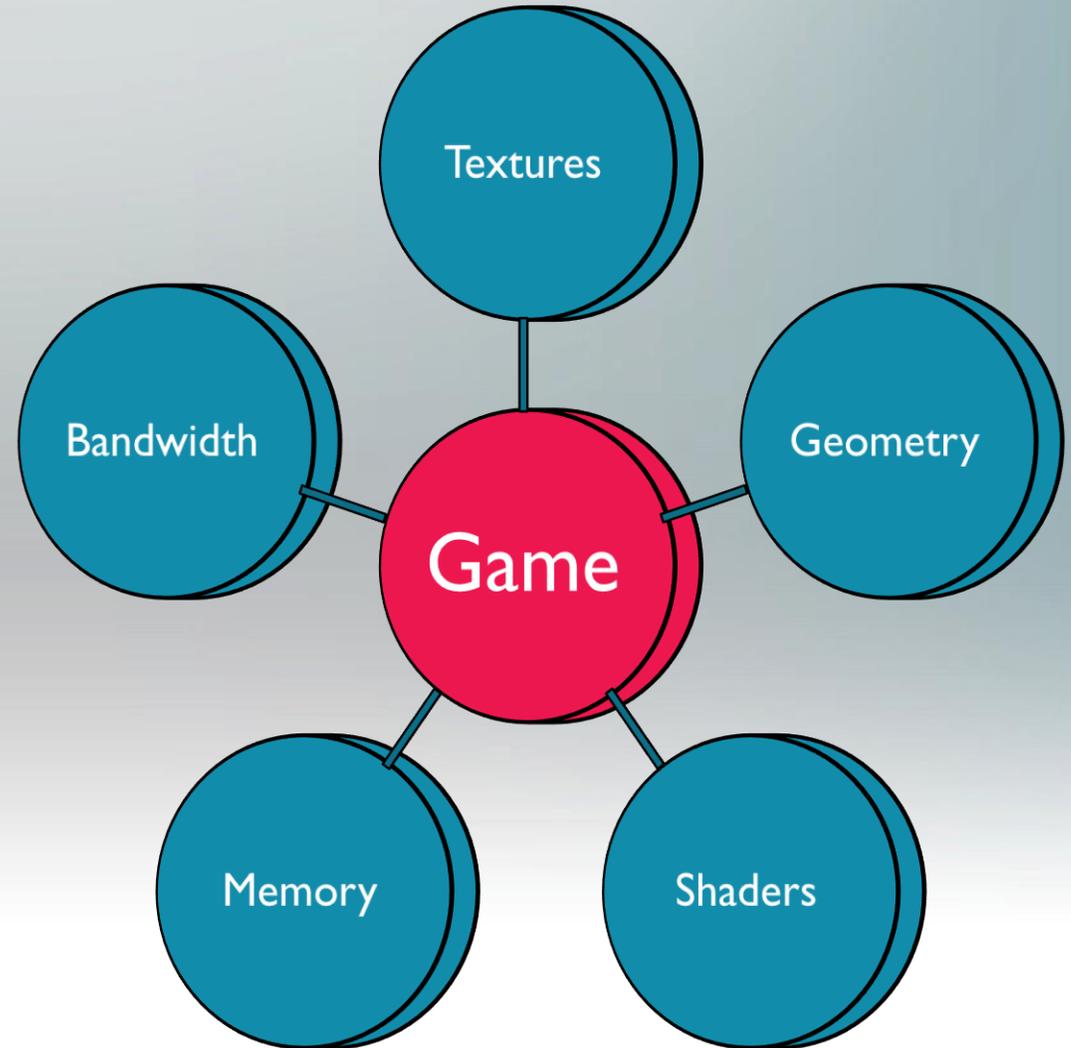


## Mali Graphics Debugger

- Graphics debugging for content developers
- API level tracing
- Understand issues and causes at frame level
- Support for OpenGL<sup>®</sup> ES 2.0, 3.0, EGL & OpenCL<sup>™</sup> I.1
- Complimentary to DS-5 Streamline

# API Level Analysis

- **GOAL: increase the performance of the application (frames per second)**
  - Analyse a 3D application running on a Mali GPU device
  - Find issues
  - Present potential solutions
- Improve the understanding of the graphics pipeline
  - Draw call stepping & profiling
  - State viewing



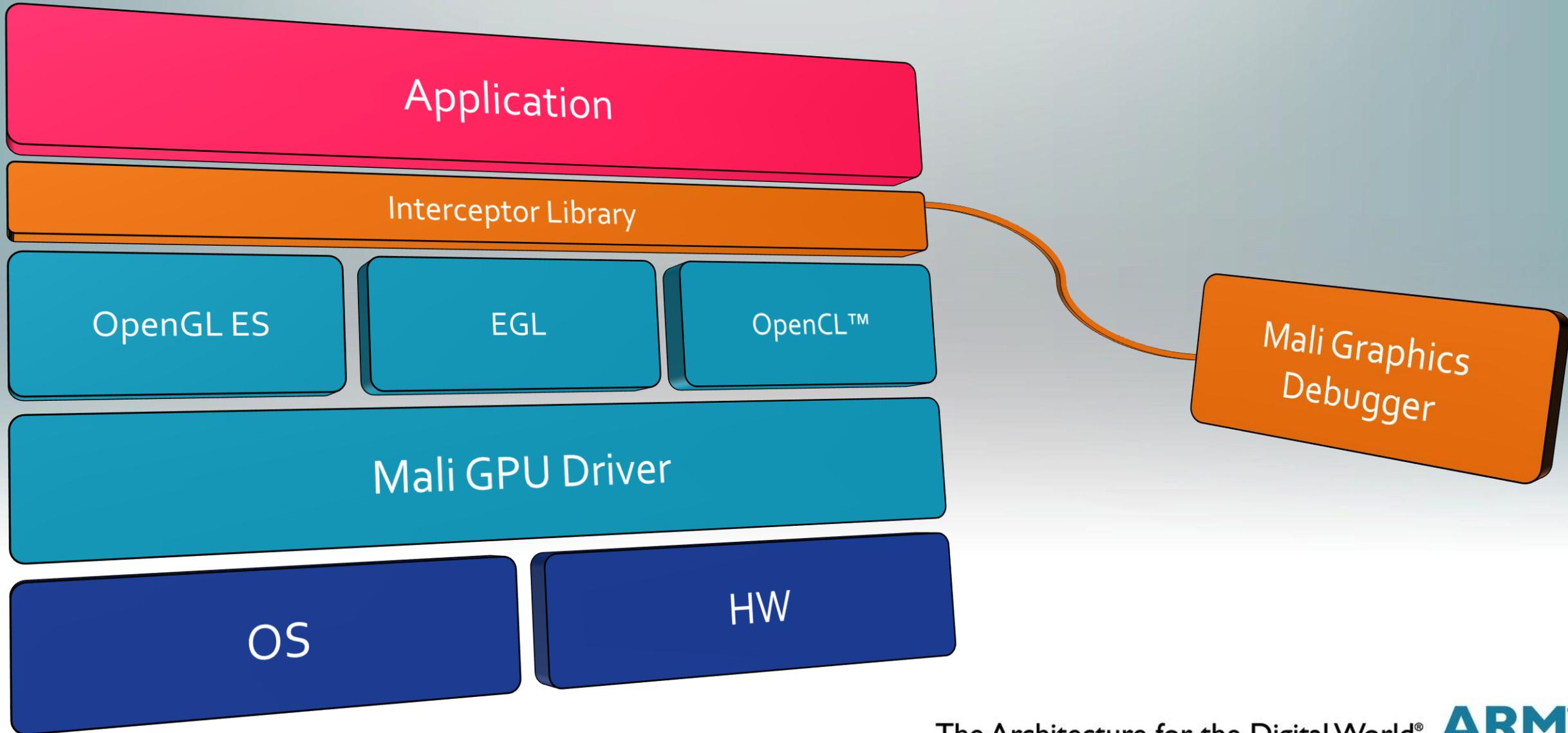
# Analyzing and Debugging on Device



TCP/IP

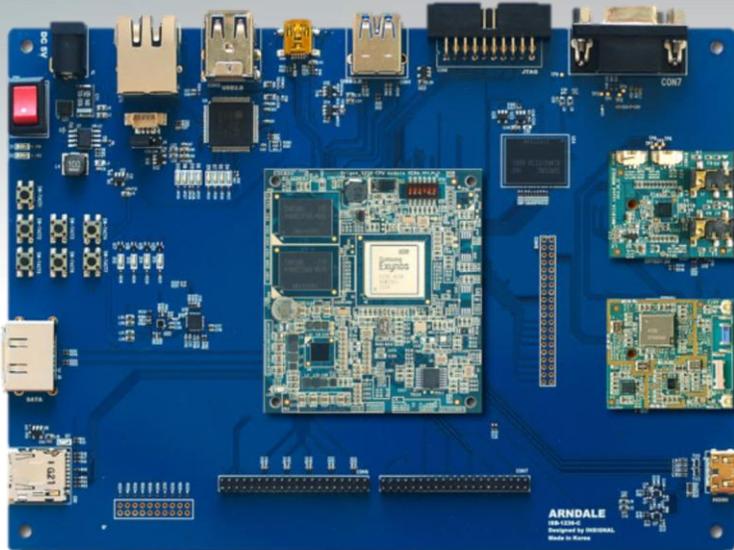


# Architecture of the Mali Graphics Debugger



# Out of the Box

- Nexus™ 10
  - Dual-core Cortex-A15 processor @1.7GHz
  - Mali-T604 @533MHz
  - 4 MP resolution (2560×1600)
  - Stock Android™ 4.2.1 with DS-5 support



- Arndale
  - First Samsung Exynos 5 development board
  - Targets both Linux and Android
  - Low cost development at around \$250

# Mali Graphics Debugger

Frame Outline

Framebuffer / Render Targets

Frame Stats

The screenshot displays the Mali Graphics Debugger interface with several key components:

- Frame Outline:** A tree view on the left showing a sequence of frames from 184 to 209.
- Framebuffer / Render Targets:** A central 3D scene view and a checkerboard overlay showing Framebuffer 0 and Framebuffer 1.
- Frame Stats:** A statistics panel on the right showing performance metrics:

Metric	Value
Total number of API function calls	24895
Avg. fps	57.96
Avg. vert/frame	14398.28
Avg. instanced vert/frame	0.00
Avg. draw/frame	4.62
- State View:** A panel on the right showing the current OpenGL state, including EGL context, texture, and blend states.
- API Trace:** A console window at the bottom left showing a list of OpenGL function calls and their return values.
- Textures:** A table at the bottom center listing textures with their names, sizes, and internal formats.

Name	Size	Int. format
Texture 29	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 3	256 x 256	GL_RGBA4
Texture 30	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 31	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 32	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
- Asset/Shader View:** A panel on the right showing a tree view of assets and shaders, including a GLSL shader program.
- Dynamic Help:** A message at the bottom left stating "80.62% of the draw calls are using GL\_TRIANGLES".

Dynamic Help

API Trace

Textures

Asset/Shader View

# API Calls Affect the State

## Application

```
while(true)
{
    angleX += 3;
    if(angleX >= 360) angleX -= 360;
    if(angleX < 0) angleX += 360;
    modelView = Matrix::createRotationX(angleX);
    modelView[8] -= 2.5;

    glUniformMatrix4fv(iLocMVP, 1, GL_FALSE,
        modelViewPerspective.getAsArray());

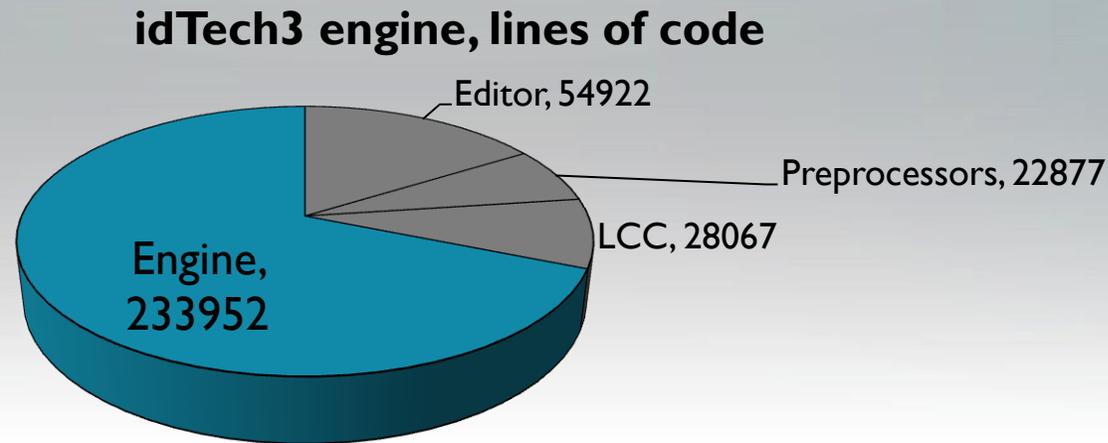
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
...
```

## Driver

```
glUniformMatrix4fv(5, 1, 0, [.36 .48 .8...]);
glClear(16384);
glDrawArrays(4, 0, 36);
glUniformMatrix4fv(5, 1, 0, [.31 .42 .5...]);
glClear(16384);
glDrawArrays(4, 0, 36);
glUniformMatrix4fv(5, 1, 0, [.26 .37 .2...]);
glClear(16384);
glDrawArrays(4, 0, 36);
glUniformMatrix4fv(5, 1, 0, [.21 .35 -.1...]);
glClear(16384);
glDrawArrays(4, 0, 36);
...
```

# Analyzing 3D Applications

- Games tend to have quite large code bases
  - Quake III engine (1999), ~ 300k lines of C code
  - Ogre3D (2012), ~1M lines of C++ code

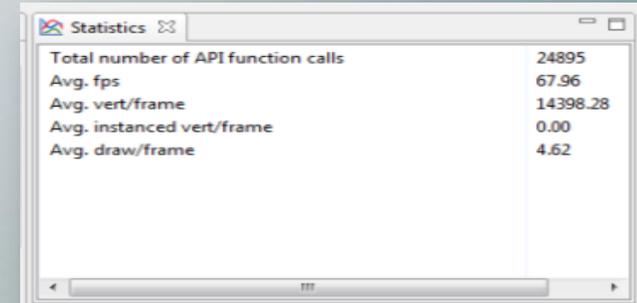


\* <http://fabiansanglard.net/quake3> Updated: Feb, 02, 2013.

# Identify Issues

## ■ Statistics View to identify :

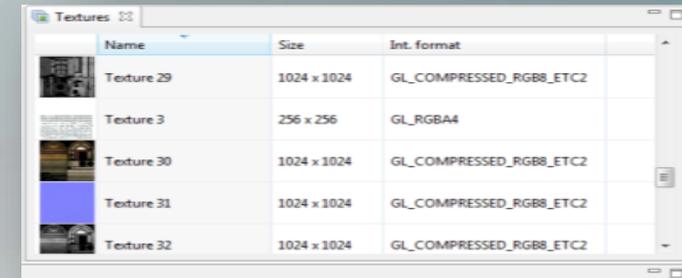
- High vertex count,
- Large number of draw calls per frame



Metric	Value
Total number of API function calls	24895
Avg. fps	67.96
Avg. vert/frame	14398.28
Avg. instanced vert/frame	0.00
Avg. draw/frame	4.62

## ■ Texture View to identify :

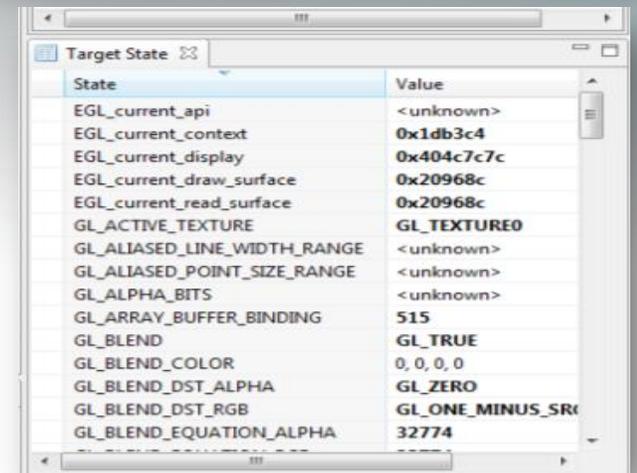
- Frames where compression is not being used
- Unnecessarily large texture dimensions
- Unnecessary pixel format being used



Name	Size	Int. format
Texture 29	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 3	256 x 256	GL_RGBA4
Texture 30	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 31	1024 x 1024	GL_COMPRESSED_RGBA_ETC2
Texture 32	1024 x 1024	GL_COMPRESSED_RGBA_ETC2

## ■ Target State View to identify :

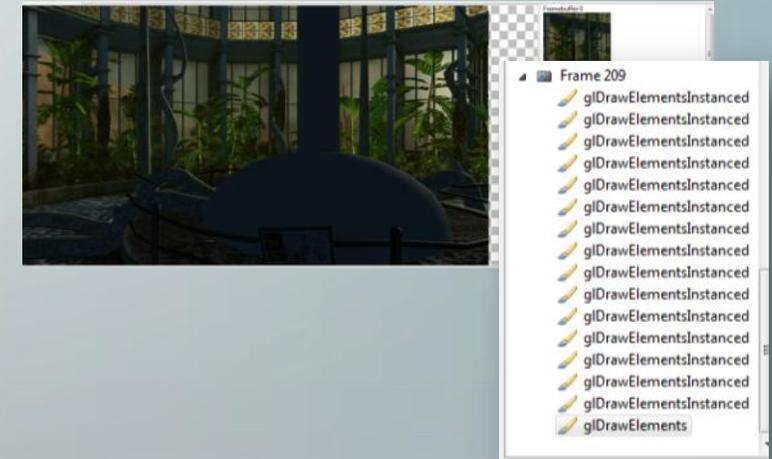
- Surprise changes to state
- Unnecessary changes



State	Value
EGL_current_api	<unknown>
EGL_current_context	0x1db3c4
EGL_current_display	0x404c7c7c
EGL_current_draw_surface	0x20968c
EGL_current_read_surface	0x20968c
GL_ACTIVE_TEXTURE	GL_TEXTURE0
GL_ALIASED_LINE_WIDTH_RANGE	<unknown>
GL_ALIASED_POINT_SIZE_RANGE	<unknown>
GL_ALPHA_BITS	<unknown>
GL_ARRAY_BUFFER_BINDING	515
GL_BLEND	GL_TRUE
GL_BLEND_COLOR	0, 0, 0, 0
GL_BLEND_DST_ALPHA	GL_ZERO
GL_BLEND_DST_RGB	GL_ONE_MINUS_SRC
GL_BLEND_EQUATION_ALPHA	32774

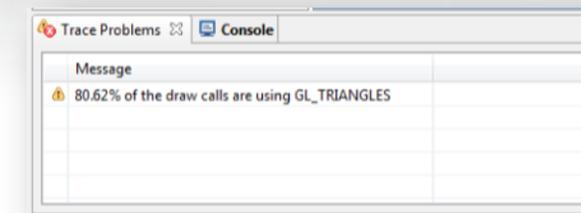
# Identify Issues

- **Draw-Call stepping to identify :**
  - Redundant draw-calls
  - Potential opportunity for batching
- **Shader Statistic View to identify :**
  - Expensive shaders by cycle count
- **Dynamic Help to highlight optimization opportunity**



A screenshot of a Shader Statistic View table. The table has four columns: 'Name', 'Instruc...', 'Shortes...', and 'Longes...'. The data is as follows:

Name	Instruc...	Shortes...	Longes...
Shader 2	47	47	47
Shader 11	34	34	34
Shader 14	33	33	33
Shader 5	23	23	23
Shader 8	23	23	23
Shader 16	23	23	23

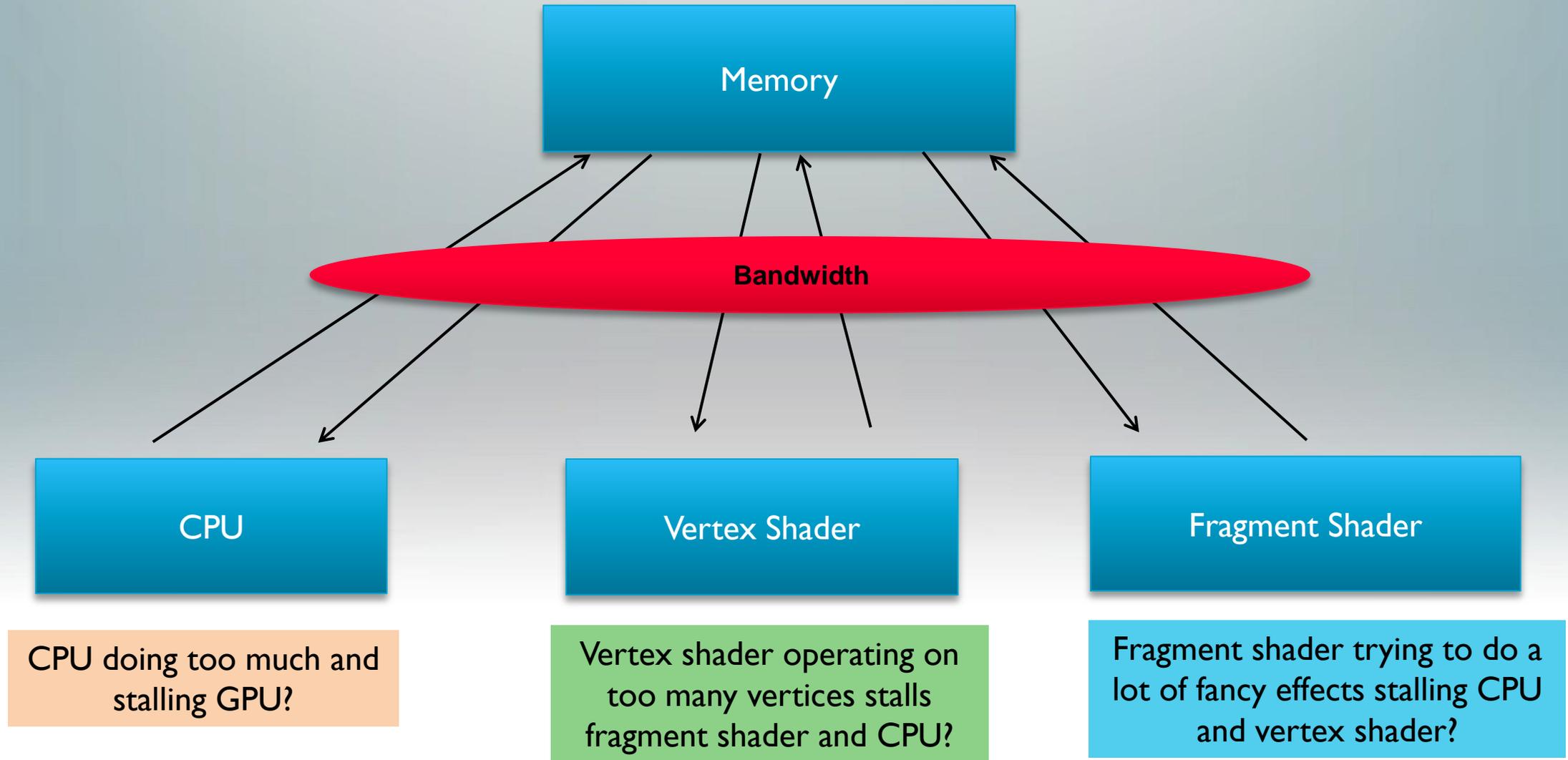


# Timbuktu Demo

The ARM logo is displayed in a bold, blue, sans-serif font, centered within a light gray rectangular area.

The Architecture for the Digital World®

# Bandwidth Vertex Fragment CPU

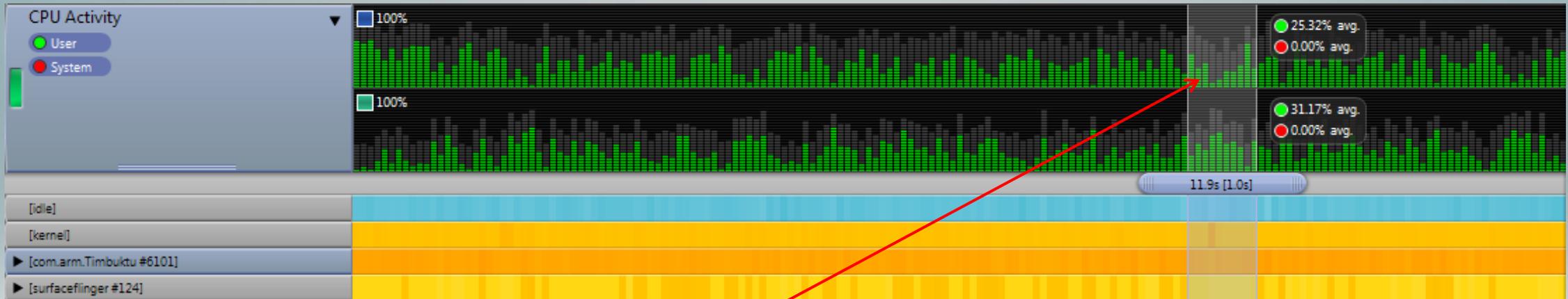


# CPU Bound

- Sometimes a slow frame rate can actually be a CPU issue and not a GPU one
  - In this case optimizing your graphics won't achieve anything
- Most mobile devices have more than one core these days Are you threading your application as much as possible?
  
- Mali GPU is a deferred architecture
  - Reduce the amount of draw calls you make
  - Try to combine your draw calls together
- Offload some of the work to the GPU
  - Even easier with Mali-T604 supporting OpenCL Full Profile

# CPU Bound Streamline

- Easy just look at the CPU Activity
  - Remember to look at all the cores.



Some of the area is greyed out due to Streamline's ability to present per App CPU activity

# Fragment Bound

- Overdraw
  - This is when you draw to each pixel on the screen more than once
  - Drawing your objects front to back instead of back to front reduces overdraw
  - Also limiting the amount of transparency in the scene can help
- Resolution too high or too many effects or cycles in shader
  - Every light and effect that you add will add to the number of cycles your shader will take
  - If you decide to run your app at native resolution be careful

# DS-5 Streamline: Fragment Bound

- Involves just 1 counter and the frequency of the GPU
  - Job Slot 0 Active

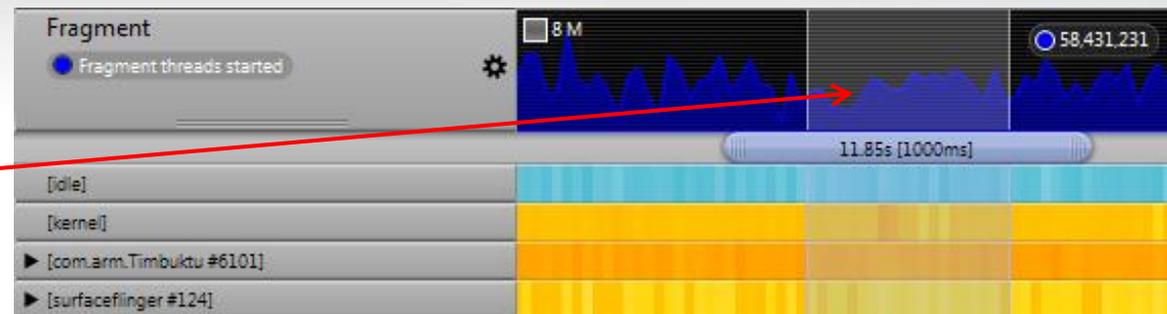


$$\text{Fragment Percentage} = (\text{Job Slot 0 active} / \text{Frequency}) * 100$$

Fragment Percentage = 84%

$$\text{Overdraw} = \text{Fragment Threads Started} * \text{Number of Cores} / \text{Resolution} * \text{FPS}$$

Overdraw = 3.9



# Mali Graphics Debugger: Shaders Statistics

- Find the top heavyweights
- Run the Mali Offline Shader Compiler
- Calculate statistics about the shaders they were compiled for
  - Number of instructions
  - Shortest and longest paths

```
ogre_trace.txt Shader 2 (GL_VERTEX_SHADER) X
```

```
uniform mat4   worldviewproj_matrix;
uniform mat4   texture_matrix;
uniform mat4   texture_matrix1;

attribute vec4  vertex;
attribute vec4  uv0;
varying vec2   oTexCoord2_0;
varying vec2   oTexCoord2_1;
void main() {
    vec4 outputPosition;
    vec4  lLocalParam_0;
    vec4  lLocalParam_1;

    FFP_Transform(worldviewproj_matrix, vertex, outputPosition);

    FFP_Construct(1.0, 1.0, 1.0, 1.0, lLocalParam_0);

    FFP_Construct(0.0, 0.0, 0.0, 0.0, lLocalParam_1);

    FFP_TransformTexCoord(texture_matrix, uv0.xy, oTexCoord2_0);

    FFP_TransformTexCoord(texture_matrix1, uv0.xy, oTexCoord2_1);

    gl_Position = outputPosition;
}
```

Resolution  
= 4,096,000 pixels  
GPU 533Mhz  
per pixel Approx.  
PS  
your shader

Instructions ▲	Shortest path	Longest path
47	47	47
34	34	34
33	33	33
23	23	23
23	23	23
23	23	23
3	3	3
3	3	3
2	2	2
2	2	2
1	1	1
1	1	1

# Example Shader Optimization

```
10 uniform float fresnelPower;  
11 uniform sampler2D noiseMap;  
12 uniform sampler2D ...  
13 uniform sampler2D ...
```

```
14  
15 varying vec3  
16 varying vec4  
17 varying vec3  
18 varying vec3
```

```
19  
20 void main()  
21 {
```

```
22     vec2 fin
```

```
23
```

```
24     vec3 noi
```

```
25     final +
```

```
26
```

```
27     float fr
```

```
28
```

```
29     vec4 ref
```

```
30     vec4 ref
```

```
31
```

```
32     gl_FragC
```

```
33 }  
34
```

Name	Instructions	Shortest path	Longest path
Shader 135	6	6	6
Shader 174	4	4	4
Shader 192	4	4	4
Shader 21	3	3	3
Shader 129	3	3	3
Shader 165	3	3	3
Shader 204	3	3	3
Shader 3	2	2	2
Shader 12	2	2	2
Shader 18	2	2	2
Shader 147	2	2	2
Shader 159	2	2	2

```
... * noiseScale;  
  
fresnelPower);
```

# Debug the draw calls

Outline

- ▶ Frame 15 : 50 draws
- ▶ Frame 16 : 36 draws
- ▶ Frame 17 : 63 draws
- ▶ Frame 18 : 63 draws
- ▼ Frame 19 : 63 draws
  - 0 glDrawElements : 10629 vert.
  - 1 glDrawElements : 3912 vert.
  - 2 glDrawElements : 10629 vert.
  - 3 glDrawElements : 3912 vert.
  - 4 glDrawElements : 3168 vert.
  - 5 glDrawElements : 1320 vert.
  - 6 glDrawElements : 804 vert.
  - 7 glDrawElements : 3036 vert.
  - 8 glDrawElements : 2220 vert.
  - 9 glDrawElements : 1338 vert.
  - 10 glDrawElements : 1080 vert.
  - 11 glDrawElements : 2718 vert.
  - 12 glDrawElements : 2133 vert.
  - 13 glDrawElements : 21483 vert.
  - 14 glDrawElements : 2352 vert.
  - 15 glDrawElements : 898 vert.
  - 16 glDrawElements : 2626 vert.



62

# Debug the draw calls

Outline

- ▶ Frame 15 : 50 draws
- ▶ Frame 16 : 36 draws
- ▶ Frame 17 : 63 draws
- ▶ Frame 18 : 63 draws
- ▼ Frame 19 : 63 draws
  - 0 glDrawElements : 10629 vert.
  - 1 glDrawElements : 3912 vert.
  - 2 glDrawElements : 10629 vert.
  - 3 glDrawElements : 3912 vert.
  - 4 glDrawElements : 3168 vert.
  - 5 glDrawElements : 1320 vert.
  - 6 glDrawElements : 804 vert.
  - 7 glDrawElements : 3036 vert.
  - 8 glDrawElements : 2220 vert.
  - 9 glDrawElements : 1338 vert.
  - 10 glDrawElements : 1080 vert.
  - 11 glDrawElements : 2718 vert.
  - 12 glDrawElements : 2133 vert.
  - 13 glDrawElements : 21483 vert.
  - 14 glDrawElements : 2352 vert.
  - 15 glDrawElements : 898 vert.
  - 16 glDrawElements : 2626 vert.



# Vertex Bound

- Too many vertices in geometry
  - Get your artist to remove unnecessary vertices
    - A lot of artists still generate content for high end desktop content
    - Impose some budgeting and limits
  - LOD Switching
    - Only objects near the camera need to be in high detail
    - Objects that are further away don't need the same level of detail
  - Use culling
- Too many cycles in the vertex shader
  - You only have a limited amount of cycles to do your vertex shading
  - The amount of cycles you can afford to spend on vertex shading is directly dependent on the number of vertices

# DS-5 Streamline: Vertex Bound

- Involves just I counter and the frequency of the GPU
  - Job Slot I Active

$$\text{Vertex Percentage} = (\text{Job Slot I active} / \text{Frequency}) * 100$$



$$\text{Vertex Percentage} = 13\%$$

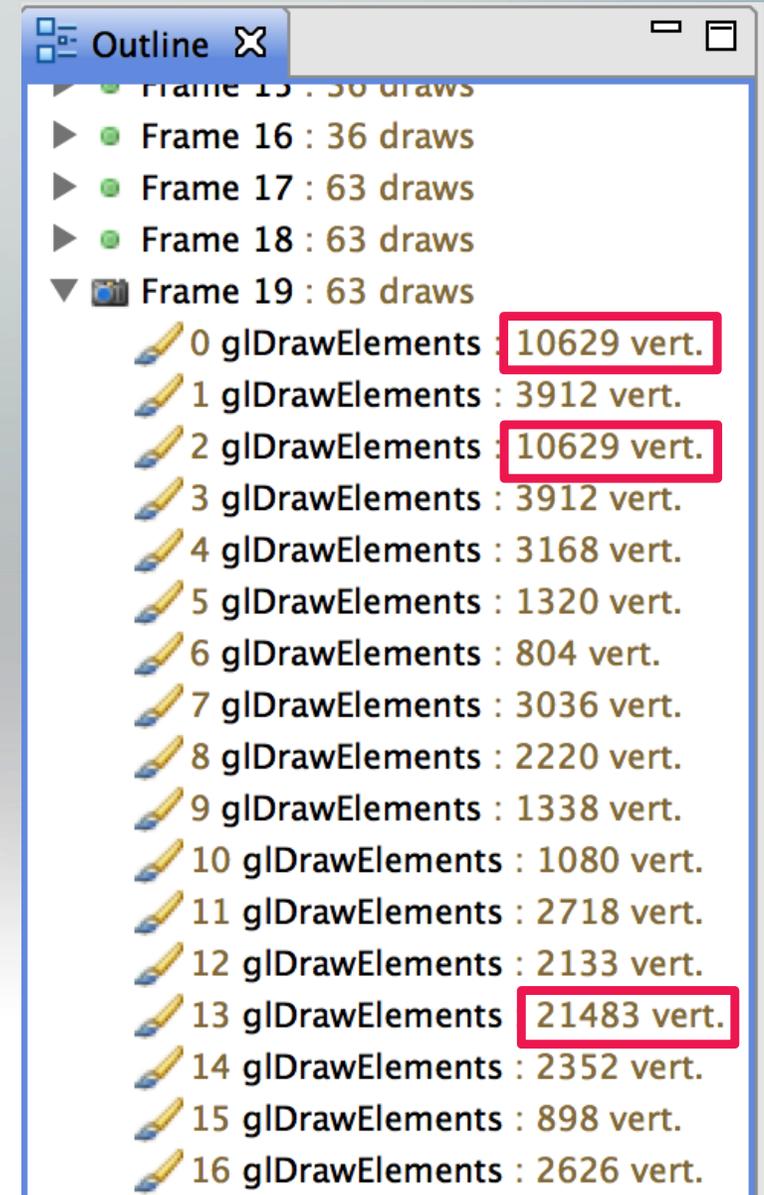
$$\text{Load Store CPI} = \text{Full Pipeline issues} / \text{Load Store Instruction Words Completed}$$

$$\text{Load Store CPI} = 2.02$$



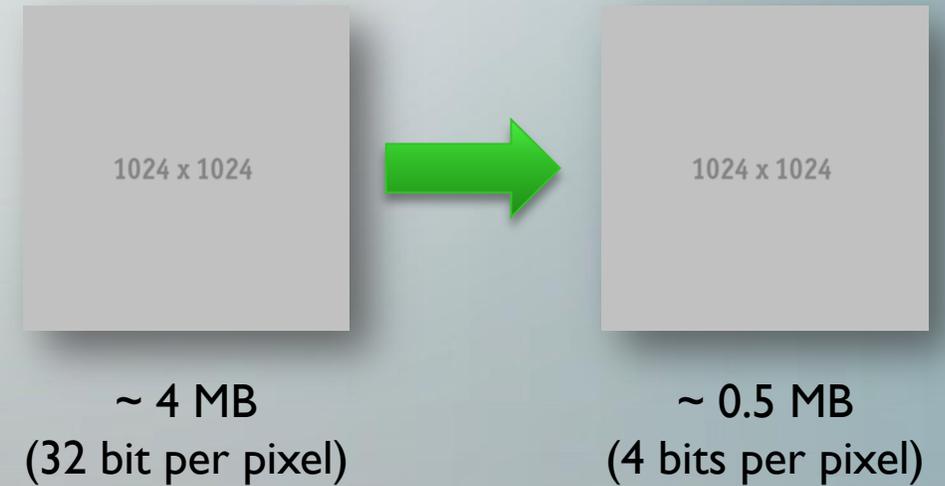
# Vertices Count

- Analyze the trace in Mali Graphics Debugger
- Find the draw calls with a high number of vertices
- Shader Statistics
  - Find the vertex shaders with a high number of instructions



# Bandwidth Bound

- When creating embedded graphics applications bandwidth is a scarce resource
  - A typical embedded device can handle  $\approx 5.0$  Gigabytes a second of bandwidth
  - A typical desktop GPU can do in excess of 100 Gigabytes a second
- Use texture compression
  - The main popular format is ETC Texture Compression
  - ASTC (Adaptive Scalable Texture Compression)  $< 1$  bit per pixel



<http://blogs.arm.com/multimedia/643-astc-texture-compression-arm-pushes-the-envelope-in-graphics-technology/>

# DS-5 Streamline: Bandwidth Counters

- Involves just 2 Streamline Counters
  - External Bus Read Beats
  - External Bus Write Beats

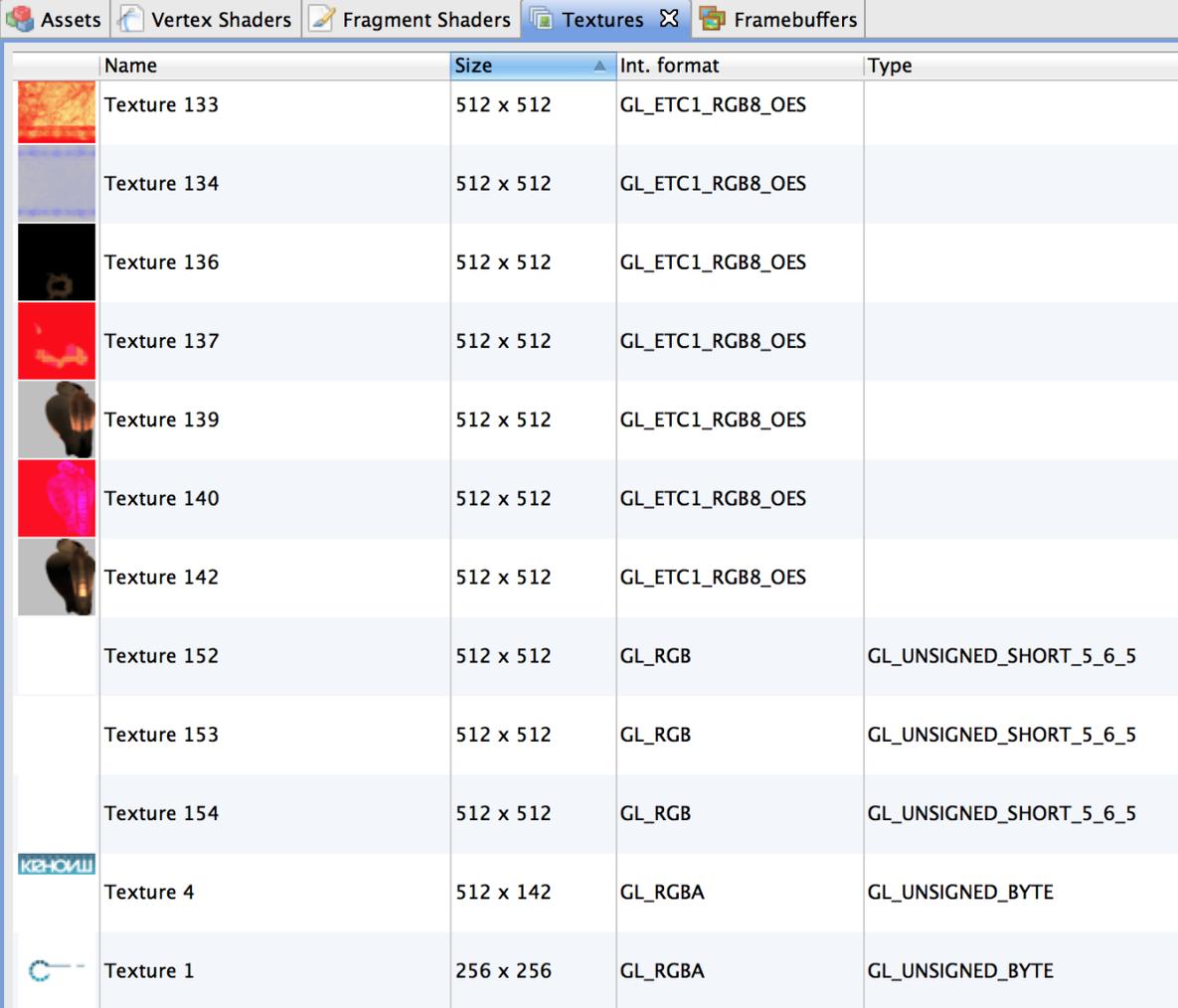
Bandwidth = 967 MB/S



**Bandwidth in Bytes = (External Bus Read Beats + External Bus Write Beats) \* Bus Width**

# Mali Graphics Debugger: Textures

- Show preview of the textures
- RGB, RGBA, Luminance Textures
  - Uncompressed
  - ETC1
  - ETC2
  - ASTC
- Sort by size, format, type
- Display full size



Name	Size	Int. format	Type
 Texture 133	512 x 512	GL_ETC1_RGB8_OES	
 Texture 134	512 x 512	GL_ETC1_RGB8_OES	
 Texture 136	512 x 512	GL_ETC1_RGB8_OES	
 Texture 137	512 x 512	GL_ETC1_RGB8_OES	
 Texture 139	512 x 512	GL_ETC1_RGB8_OES	
 Texture 140	512 x 512	GL_ETC1_RGB8_OES	
 Texture 142	512 x 512	GL_ETC1_RGB8_OES	
 Texture 152	512 x 512	GL_RGB	GL_UNSIGNED_SHORT_5_6_5
 Texture 153	512 x 512	GL_RGB	GL_UNSIGNED_SHORT_5_6_5
 Texture 154	512 x 512	GL_RGB	GL_UNSIGNED_SHORT_5_6_5
 Texture 4	512 x 142	GL_RGBA	GL_UNSIGNED_BYTE
 Texture 1	256 x 256	GL_RGBA	GL_UNSIGNED_BYTE

# Batching

- Try to combine as many of your drawcalls together as possible
- If objects use different textures try to combine the textures together in a texture atlas
  - This can be done automatically but often best done by artists
  - Update your texture coordinates accordingly

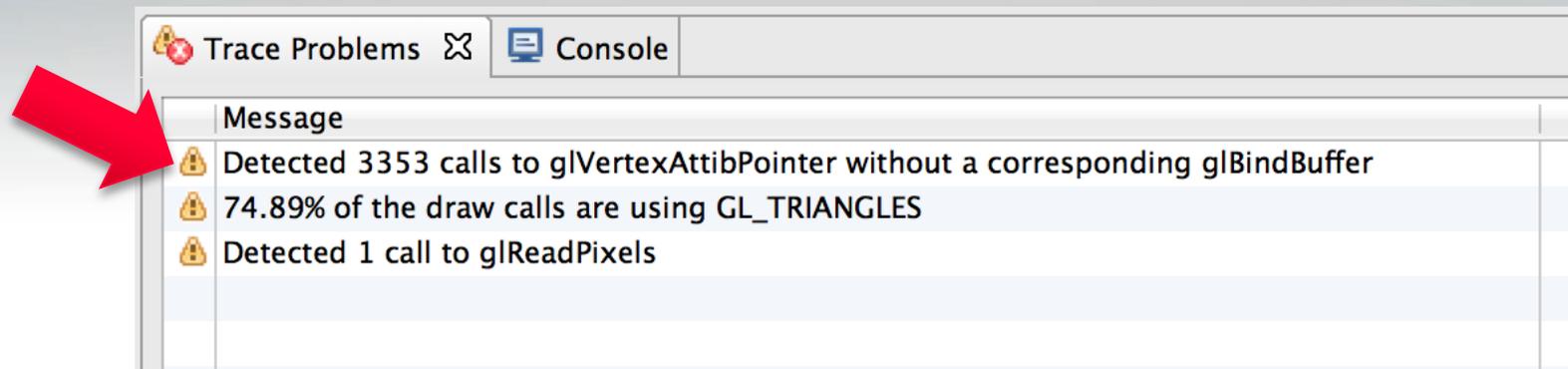


```
glBindTexture(<texture1>);  
glDrawElements(<someVertices>);  
glBindTexture(<texture2>);  
glDrawElements(<someVertices2>);  
glBindTexture(<texture3>);  
glDrawElements(<someVertices3>);  
glBindTexture(<texture4>);  
Etc....
```

# Vertex Buffer Objects

## ■ Vertex Buffer Objects

- Using Vertex Buffer Objects (VBO's) can save you a lot of time in overhead
- Every frame in your application all of your vertices and colour information will get sent to the GPU
- A lot of the time these won't change. So there is no need to keep sending them
- Would be a much better idea to cache the data in graphics memory
- This is where VBO's can be useful



# Mali Graphics Debugger v1.0.1

- Available on Mali Developer Center:

<http://malideveloper.arm.com>

- Being used by our vendors and partners
- Available free of charge

Stay in touch for the next release with vertex attributes, uniforms and VBOs.



The screenshot shows the Mali Developer Center website. At the top, there is a navigation bar with links for 'LEARN about Mali', 'DEVELOP for Mali', 'ENGAGE with Mali', and 'ARM'. A search bar is located in the top right corner. The main content area features a large banner for the Mali Graphics Debugger, including a 3D product box, a screenshot of the software interface, and a 'LATEST DOWNLOAD v1.0' button. Below the banner, there is a section titled 'ANALYSIS & DEBUG' with a sub-section for 'Mali Graphics Debugger'. This section contains text describing the tool as an 'OpenGL ES 1.1, 2.0, 3.0 and OpenCL™ 1.1 API Trace and Debug Tool' and lists its features and benefits, such as 'Draw-call by Draw-call stepping'. On the right side of the page, there is a 'DEVELOP for Mali' sidebar with a list of links: Tools, Asset Creation, Software Development, Analysis & Debug, SDKs, Drivers, Sample Code, Documentation, and Development Platforms.

Thank you

[malideveloper.arm.com](https://malideveloper.arm.com)