

# Optimize Your Mobile Games With Practical Case Studies



Stephen Barton, Senior Software Engineer, ARM  
Stacy Smith, Senior Software Engineer, ARM

GDC  
March 2016

# Agenda

- Part I
  - Discuss the importance of analysis and tools
  - Overview of Mali Tools software suite
  - Brief look at Mali Graphics Debugger and Streamline
  - Look at Ice Cave through the debugging microscope
- Part II
  - Optimization techniques “The Sensible Six”
  - Optimizations in Ice Cave

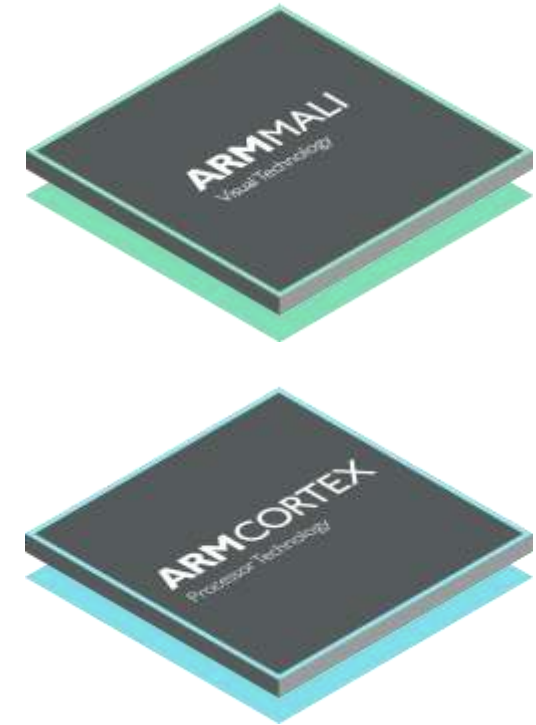
# Importance of Analysis & Debug

## Mobile Platforms

- Expectation of amazing, console-like graphics and playing experience
- Screen resolution beyond HD
- Limited power budget

## Solution

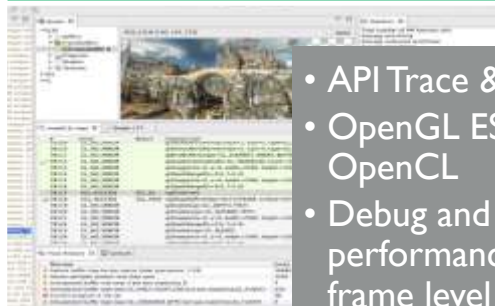
- ARM Cortex CPUs and Mali GPUs are designed for low power whilst providing innovative features to keep up performance
- Software developers can be “smart” when developing apps
- Good tools help you identify performance bottlenecks in your code



# Mali GPU Software Tools

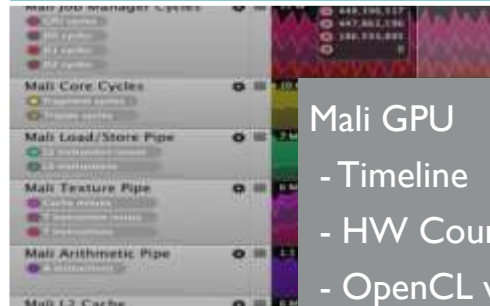
## Performance Analysis, Debug and Software Development

### Mali Graphics Debugger



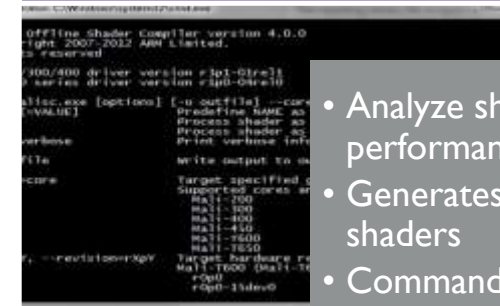
- API Trace & Debug
- OpenGL ES, OpenCL
- Debug and improve performance at frame level

### ARM DS-5 Streamline



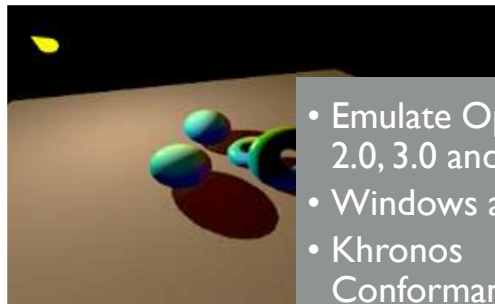
- Mali GPU
  - Timeline
  - HW Counters
  - OpenCL visualizer

### Mali Offline Compiler



- Analyze shader performance
- Generates binary shaders
- Command line tool

### OpenGL ES Emulator



- Emulate OpenGL ES 2.0, 3.0 and 3.1
- Windows and Linux
- Khronos Conformant

### Texture Compression Tool



- Command line and GUI
- ETC, ETC2, ASTC
- 3D textures

### Third Party Tools

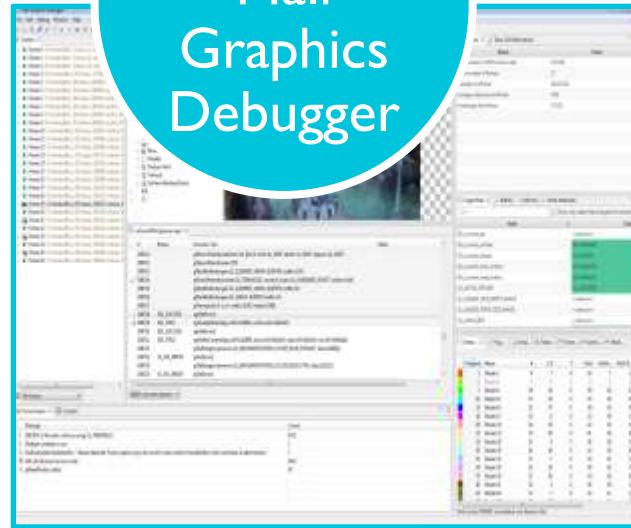


- Integration with partners' tools

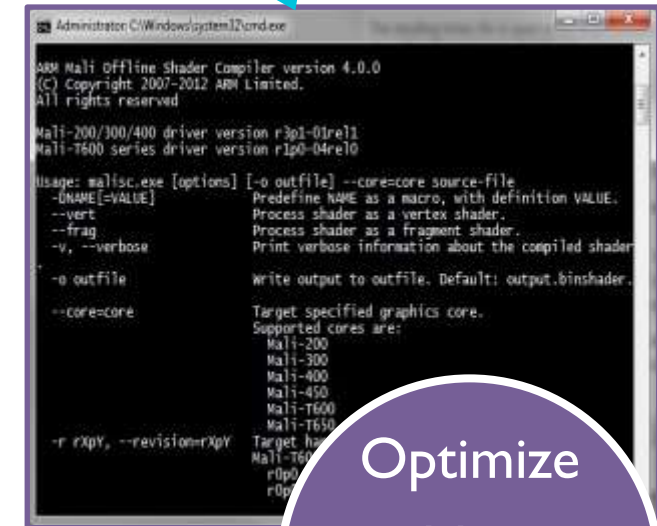
# Performance Analysis and Debug With Mali GPUs



Analyze  
DS-5  
Streamline



Debug  
Mali  
Graphics  
Debugger



```
Administrator C:\Windows\system32\cmd.exe

ARM Mali Offline Shader Compiler version 4.0.0
(c) Copyright 2007-2012 ARM Limited.
All rights reserved.

Mali-200/300/400 driver version r3p1-01rel1
Mali-T600 series driver version r1p0-04rel0

Usage: malisc.exe [options] [-o outfile] --core=core source-file
--NAME[=VALUE]      Predefine NAME as a macro, with definition VALUE.
--vert              Process shader as a vertex shader.
--frag              Process shader as a fragment shader.
-v, --verbose        Print verbose information about the compiled shader.
-o outfile           Write output to outfile. Default: output.binshader.
--core=core          Target specified graphics core.
Supported cores are:
Mali-200
Mali-300
Mali-400
Mali-450
Mali-T600
Mali-T650
Target hardware:
Mali-T600
r0p0
r0p0
```

Optimize  
Mali  
Offline  
Compiler

# Investigation With the ARM Mali Graphics Debugger

The screenshot displays the ARM Mali Graphics Debugger interface, which is divided into several functional panels. On the left, a list of frames (Frame 0 to Frame 25) is shown, with Frame 18 selected. The central panel displays a 3D scene rendered on the ARM Mali GPU. To the right, a 'Statistics' panel provides summary data: Total number of API function calls (187728), Total number of frames (27), Average vertices/frame (250472.22), Average instanced vertices/frame (0.00), and Average draw/frame (175.22). Below this, a 'States' panel shows the current state of various GPU components, including Buffers, Uniforms, Vertex Attributes, and Buffers. The 'API Trace' panel at the bottom left shows a list of function calls, with a specific call highlighted. The 'Dynamic Help' panel at the bottom right provides context-sensitive information. The 'Assets View' panel at the top left shows a tree of assets, including GLSL shaders and textures. The 'Frame Outline' panel at the top right shows a list of framebuffers. The 'Frame Capture: Framebuffers' panel at the bottom center shows a list of captured framebuffers. The 'Textures Shaders' panel at the bottom right shows a list of textures and shaders. The 'API Trace' panel at the bottom left shows a list of function calls, with a specific call highlighted. The 'Dynamic Help' panel at the bottom right provides context-sensitive information.

Assets View

Frame Outline

API Trace

Dynamic Help

Frame Capture: Framebuffers

Frame Statistics

States  
Uniforms  
Vertex  
Attributes  
Buffers

Textures  
Shaders



# Mali Graphics Debugger

## Outline view

The screenshot displays the Outline view of the Mali Graphics Debugger. It shows a hierarchical tree structure of graphics events. The top level lists frames from Frame 9 to Frame 18. Frame 13 is expanded, showing its sub-entries: Framebuffer 11, Framebuffer 12, Framebuffer 9, Framebuffer 10, Framebuffer 1, Framebuffer 2, and Framebuffer 0. Framebuffer 11 is further expanded, showing a list of 10 draw calls. The third draw call is selected and highlighted in blue. To the right of the tree, a vertical timeline shows the sequence of draw calls with their corresponding addresses (e.g., 97936, 97937, etc.). The selected draw call at address 97954 is highlighted in the timeline. To the right of the timeline, the OpenGL ES commands for the selected draw call are listed, including `glDisableVertexAttribArray`, `glBindBuffer`, `glVertexAttribPointer`, `glUniformMatrix4fv`, `glBindBufferBase`, `glBindBuffer`, `glDrawElements`, and `glBindBuffer`.

- ▶ Frame 9 : 7 framebuffers, 215 draws, 293805 vertices
- ▶ Frame 10 : 7 framebuffers, 192 draws, 292431 vertices
- ▶ Frame 11 : 7 framebuffers, 192 draws, 292623 vertices
- ▶ Frame 12 : 7 framebuffers, 197 draws, 292665 vertices
- ▲ Frame 13 : 7 framebuffers, 193 draws, 292713 vertices
  - ▶ Framebuffer 11 : 31 draws, 70722 vertices, 14709 vertices
  - ▶ Framebuffer 12 : 1 draw, 0 vertices, 0 unique indices
  - ▶ Framebuffer 9 : 41 draws, 72372 vertices, 15771 unique indices
  - ▶ Framebuffer 10 : 1 draw, 0 vertices, 0 unique indices
  - ▶ Framebuffer 1 : 117 draws, 149613 vertices, 33241 unique indices
  - ▶ Framebuffer 2 : 1 draw, 0 vertices, 0 unique indices
  - ▶ Framebuffer 0 : 1 draw, 6 vertices, 4 unique indices
- ▶ Frame 14 : 7 framebuffers, 192 draws, 292653 vertices
- ▶ Frame 15 : 7 framebuffers, 192 draws, 290829 vertices
- ▶ Frame 16 : 7 framebuffers, 191 draws, 291099 vertices
- ▶ Frame 17 : 7 framebuffers, 190 draws, 291135 vertices
- ▲ Frame 18 : 7 framebuffers, 189 draws, 291177 vertices
  - ▶ Framebuffer 11 : 31 draws, 70722 vertices, 14709 vertices
    - 1 glDrawElements : 444 vertices 163 unique indices
    - 2 glDrawElements : 444 vertices 160 unique indices
    - 3 glDrawElements : 444 vertices 163 unique indices
    - 4 glDrawElements : 444 vertices 163 unique indices
    - 5 glDrawElements : 66 vertices 20 unique indices
    - 6 glDrawElements : 66 vertices 20 unique indices
    - 7 glDrawElements : 66 vertices 20 unique indices
    - 8 glDrawElements : 66 vertices 20 unique indices
    - 9 glDrawElements : 444 vertices 163 unique indices
    - 10 glDrawElements : 444 vertices 160 unique indices

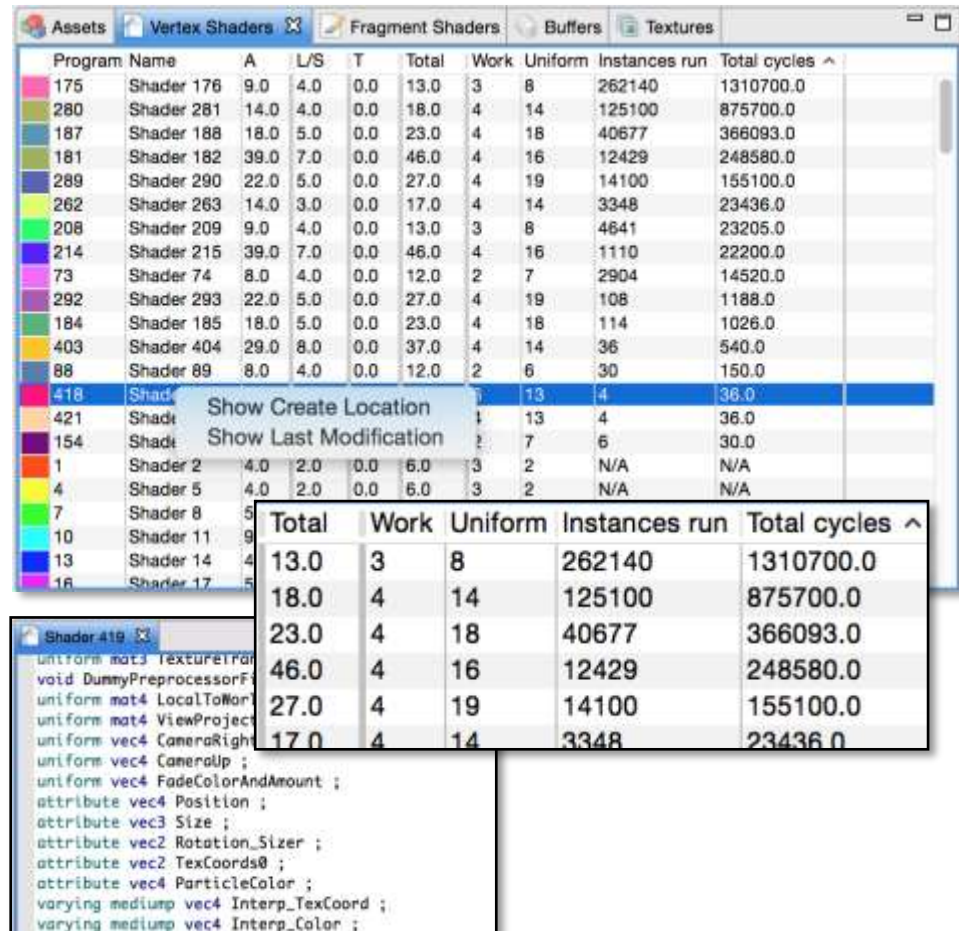
97936  
97937  
97938  
97939  
97940  
97941  
97942  
97943  
97944  
97945  
97946  
97947  
97948  
97949  
97950  
97951  
97952  
97953  
97954  
97955  
97956  
97957  
97958  
97959  
97960  
97961  
97962

`glDisableVertexAttribArray`  
`glBindBuffer(target, buffer)`  
`glVertexAttribPointer(index, size, type, normalized, stride, offset)`  
`glUniformMatrix4fv(location, count, type, values)`  
`glBindBufferBase(target, index, buffer)`  
`glBindBuffer(target, buffer)`  
`glDrawElements(mode, count, type, offset)`  
`glBindBuffer(target, buffer)`  
`glVertexAttribPointer(index, size, type, normalized, stride, offset)`  
`glUniformMatrix4fv(location, count, type, values)`  
`glBindBufferBase(target, index, buffer)`  
`glBindBuffer(target, buffer)`  
`glDrawElements(mode, count, type, offset)`  
`glBindBuffer(target, buffer)`  
`glDrawElements(mode, count, type, offset)`  
`glVertexAttribPointer(index, size, type, normalized, stride, offset)`  
`glUniformMatrix4fv(location, count, type, values)`

- Quick access to key graphics events
  - Frames
  - Render targets
  - Draw calls
- Investigate at frame level
  - Find which draw calls have higher geometry impact
- Jump between the outline view and the trace view seamlessly

# Mali Graphics Debugger

## Shaders reports and statistics



Program Name	A	L/S	T	Total	Work	Uniform	Instances run	Total cycles
175 Shader 176	9.0	4.0	0.0	13.0	3	8	262140	1310700.0
280 Shader 281	14.0	4.0	0.0	18.0	4	14	125100	875700.0
187 Shader 188	18.0	5.0	0.0	23.0	4	18	40677	366093.0
181 Shader 182	39.0	7.0	0.0	46.0	4	16	12429	248580.0
289 Shader 290	22.0	5.0	0.0	27.0	4	19	14100	155100.0
262 Shader 263	14.0	3.0	0.0	17.0	4	14	3348	23436.0
208 Shader 209	9.0	4.0	0.0	13.0	3	8	4641	23205.0
214 Shader 215	39.0	7.0	0.0	46.0	4	16	1110	22200.0
73 Shader 74	8.0	4.0	0.0	12.0	2	7	2904	14520.0
292 Shader 293	22.0	5.0	0.0	27.0	4	19	108	1188.0
184 Shader 185	18.0	5.0	0.0	23.0	4	18	114	1026.0
403 Shader 404	29.0	8.0	0.0	37.0	4	14	36	540.0
88 Shader 89	8.0	4.0	0.0	12.0	2	6	30	150.0
418 Shader 419					13	4	36.0	
421 Shader 422					13	4	36.0	
154 Shader 155					7	6	30.0	
1 Shader 2	4.0	2.0	0.0	6.0	3	2	N/A	N/A
4 Shader 5	4.0	2.0	0.0	6.0	3	2	N/A	N/A
7 Shader 8								
10 Shader 11								
13 Shader 14								
16 Shader 17								

Total	Work	Uniform	Instances run	Total cycles
13.0	3	8	262140	1310700.0
18.0	4	14	125100	875700.0
23.0	4	18	40677	366093.0
46.0	4	16	12429	248580.0
27.0	4	19	14100	155100.0
17.0	4	14	3348	23436.0

```
Shader 419
uniform mat3 TextureIR;
void DummyPreprocessorF...
uniform mat4 LocalToWorld;
uniform mat4 ViewProject...
uniform vec4 CameraRight;
uniform vec4 CameraUp;
uniform vec4 FadeColorAndAmount;
attribute vec4 Position;
attribute vec3 Size;
attribute vec2 Rotation_Sizer;
attribute vec2 TexCoords0;
attribute vec4 ParticleColor;
varying mediump vec4 Interp_TexCoord;
varying mediump vec4 Interp_Color;
```

All the shaders being used by the application are reported

### Shader statistics

Each shader is compiled with the Mali Offline Compiler and is statically analyzed to display:

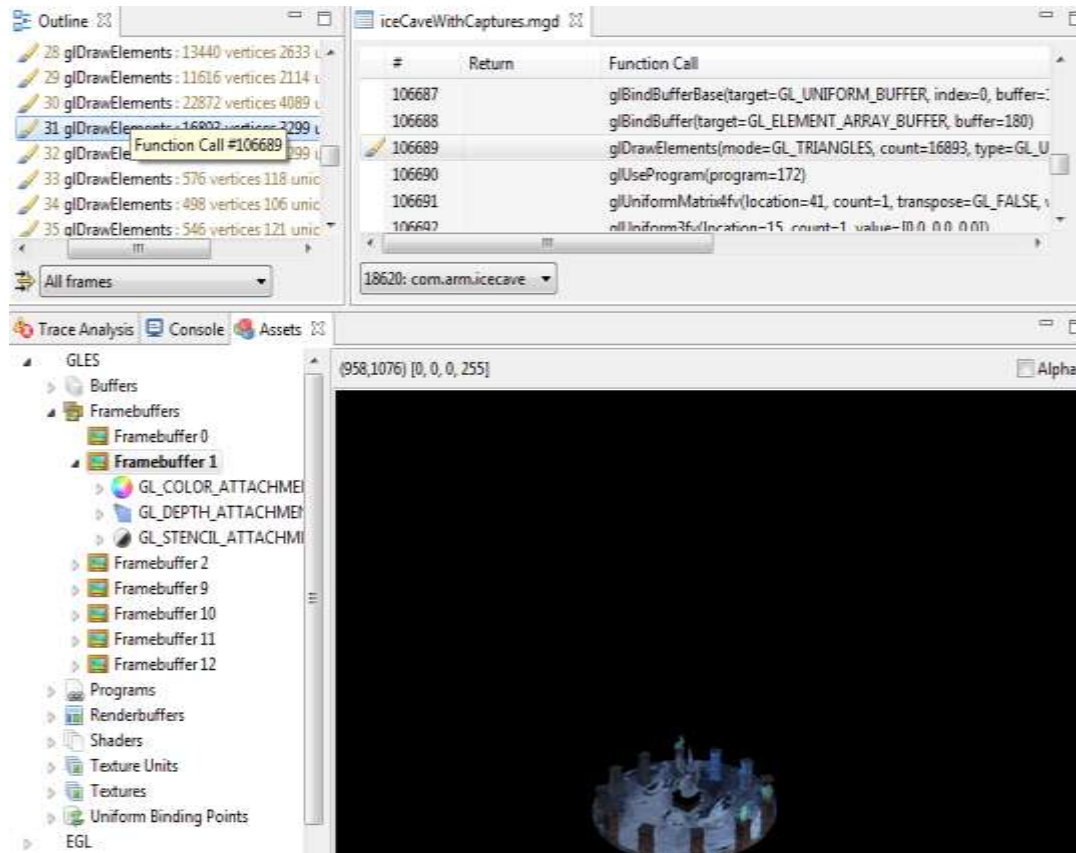
- number of instructions for each GPU pipeline
- number of work registers and uniform registers

Additionally, for each draw call MGD knows how many times that shader has been executed (i.e. the number of vertices) and overall statistics are calculated



# Mali Graphics Debugger

## Frame capture and analysis



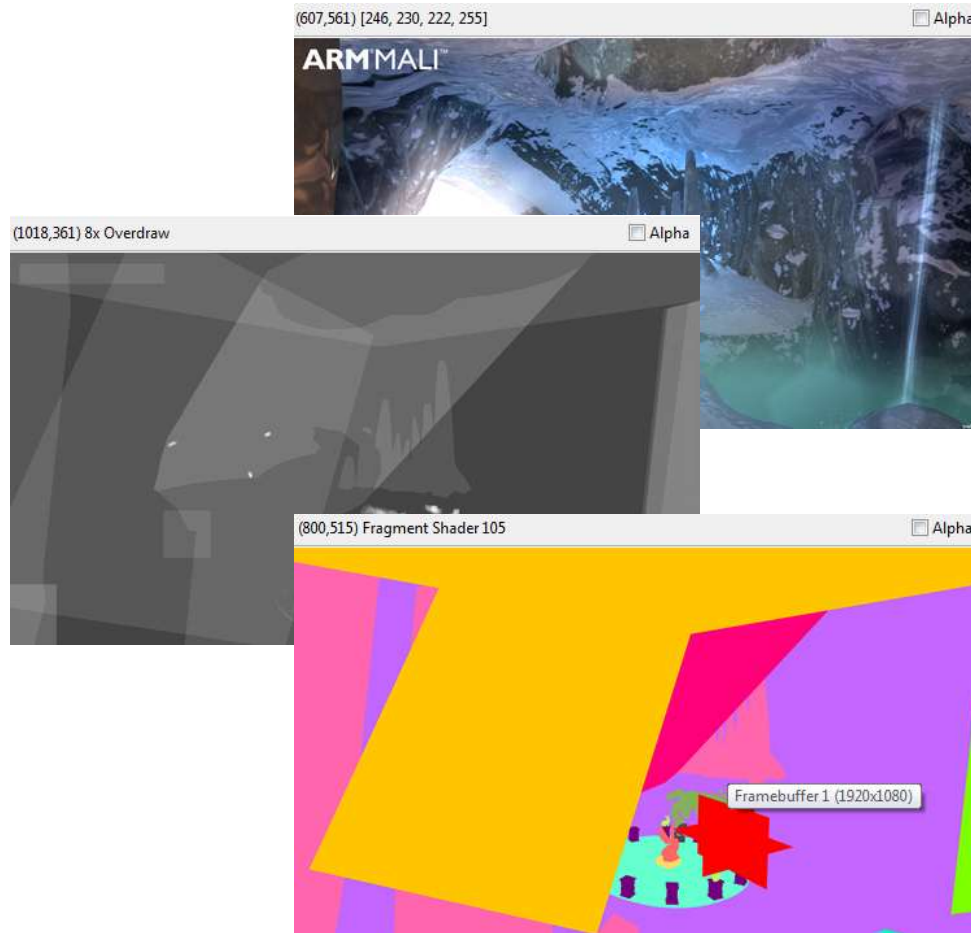
Frames can be fully captured to analyze the effect of each draw call

- A native resolution snapshot of each framebuffer is captured after every draw call
- The capture happens on target, so target dependent bugs or precision issues can be investigated

All the images can be exported and analyzed separately.

# Mali Graphics Debugger

## Alternative drawing modes



Different drawing modes can be forced and used for both live rendering and frame captures

### Native mode

- Frames are rendered with the original shaders

### Overdraw mode

- Highlights where overdraw happens (ie. objects are drawn on top of each other)

### Shader map mode

- Native shaders are replaced with different solid colors

### Fragment count mode

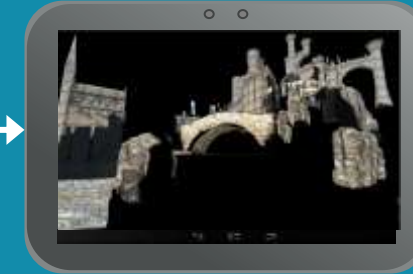
- All the fragments that are processed by each frame are counted

# Analyzing and Debugging on Device

**Host**  
Windows  
Linux  
Mac  
32-bit and 64-bit

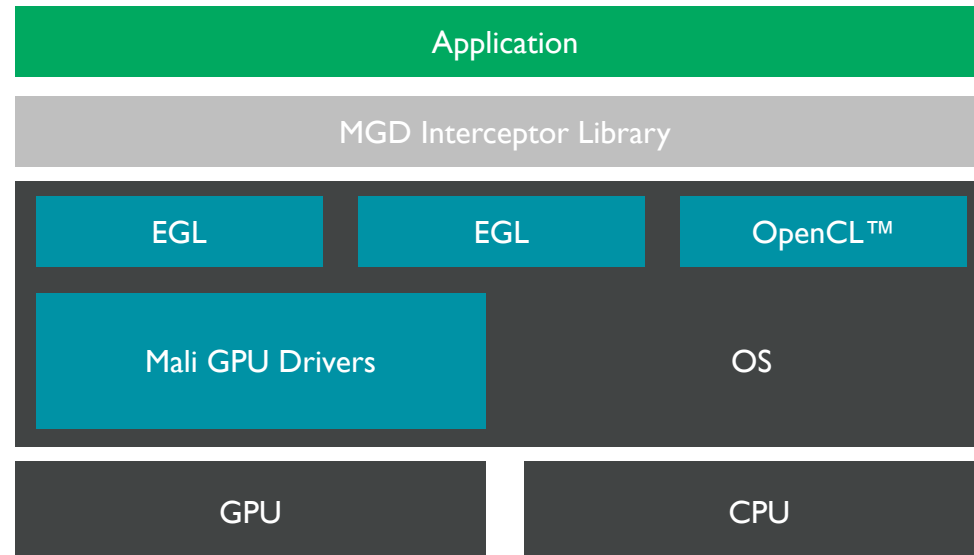


TCP/IP  
Ethernet  
USB



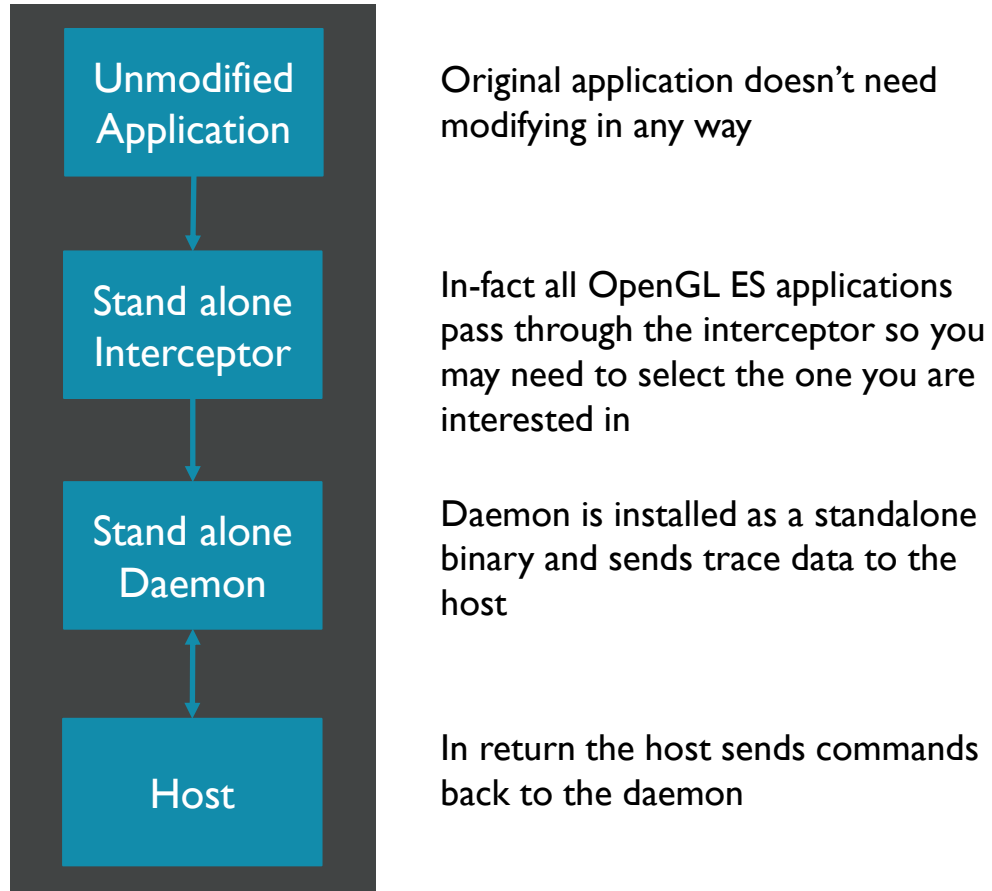
**Target**  
Android  
Linux  
32-bit and 64-bit

Under the hood of the  
Mali Graphics Debugger

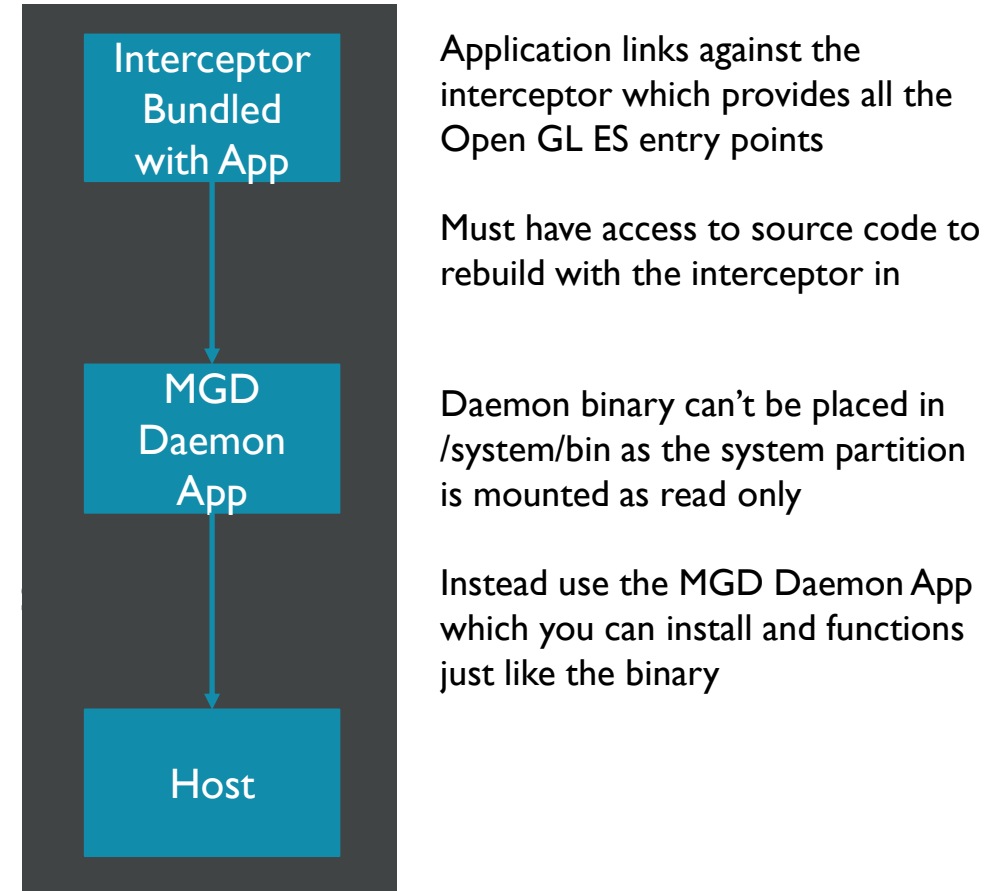


# How to get MGD to Work

- On a rooted system



- On a non-rooted system

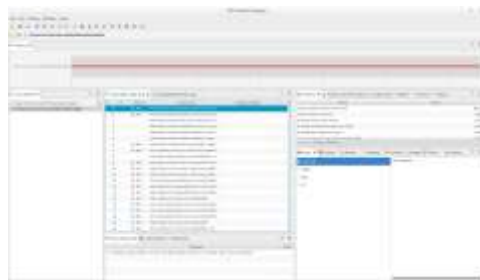


We will show both scenarios live at our lecture theatre at 1.30pm @ ARM booth #1624

# New Features in the Mali Graphics Debugger

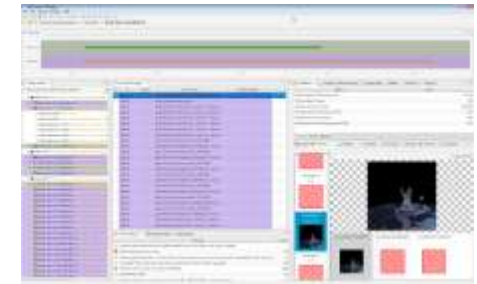
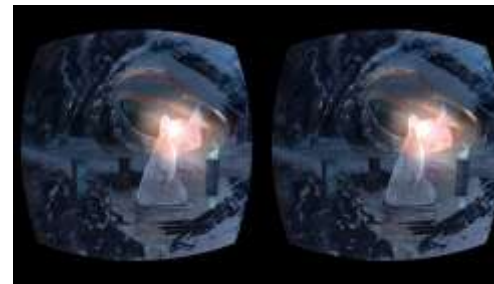
## ■ Vulkan Support

- Trace all the function calls in the spec
- Allows you to see exactly what calls compose your application
- Contact the ARM Mali forums and we would love to get you setup
- <https://community.arm.com/groups/arm-mali-graphics>



## ■ VR Support

- Frames are now registered through glFlush commands
- All standard MGD features work
- Frame Capture will now complete all render passes active when capture started





# New Features in the Mali Graphics Debugger

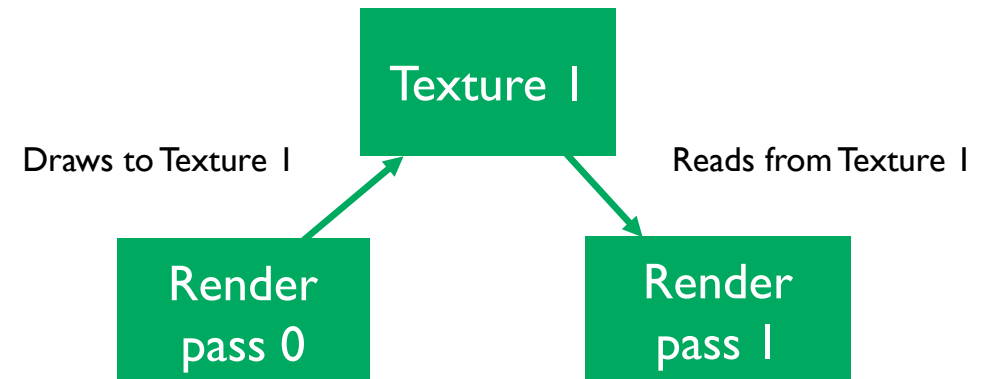
- Geometry Viewer

- Visualize all of your geometry per draw call
- Zero overhead to the application so you get this feature for free
- Open all your old traces and they will now have this support



- Render pass Dependencies

- MGD can now show you the dependencies on your scene
- Can do this for textures, render buffers and blits of the Framebuffer



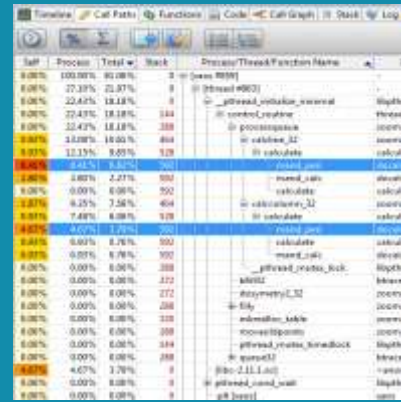
Render pass 1 has a dependency on render pass 0 due to texture 1

# Streamline



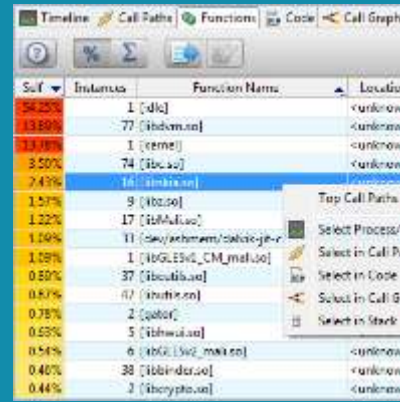
## Timeline

Visualization of system performance metrics, software profile and system events over time



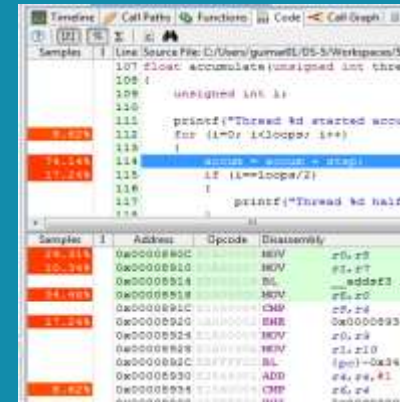
## Call Paths

Hierarchical profile table, aggregating samples per process, thread, and function call chain



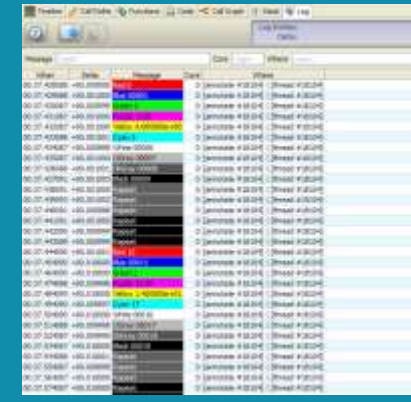
## Functions

Flat software profile table, listing shared libraries and function hotspots



## Code

Source and instruction level profile. Color coded source code lines for easy identification

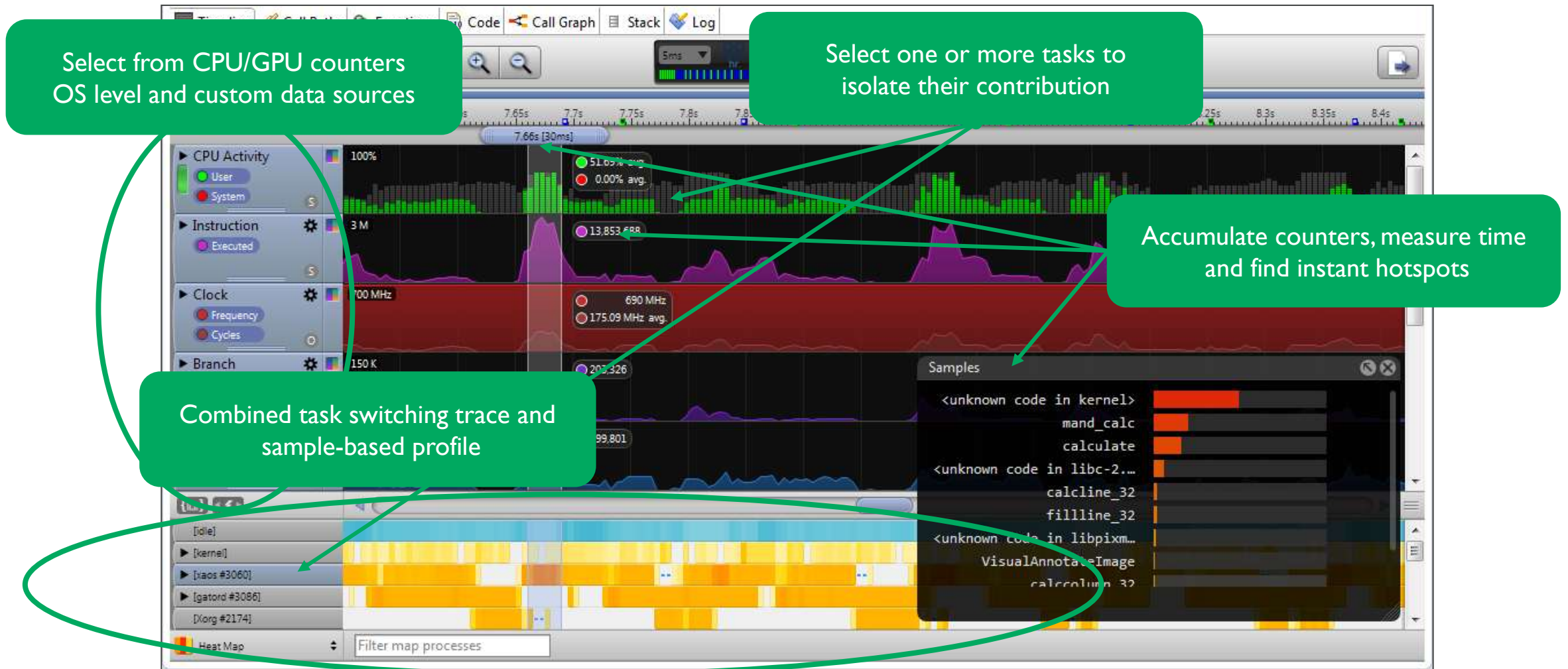


## Log

Chronologic list of text and graphic user annotations sent to Streamline, offering flexible filtering

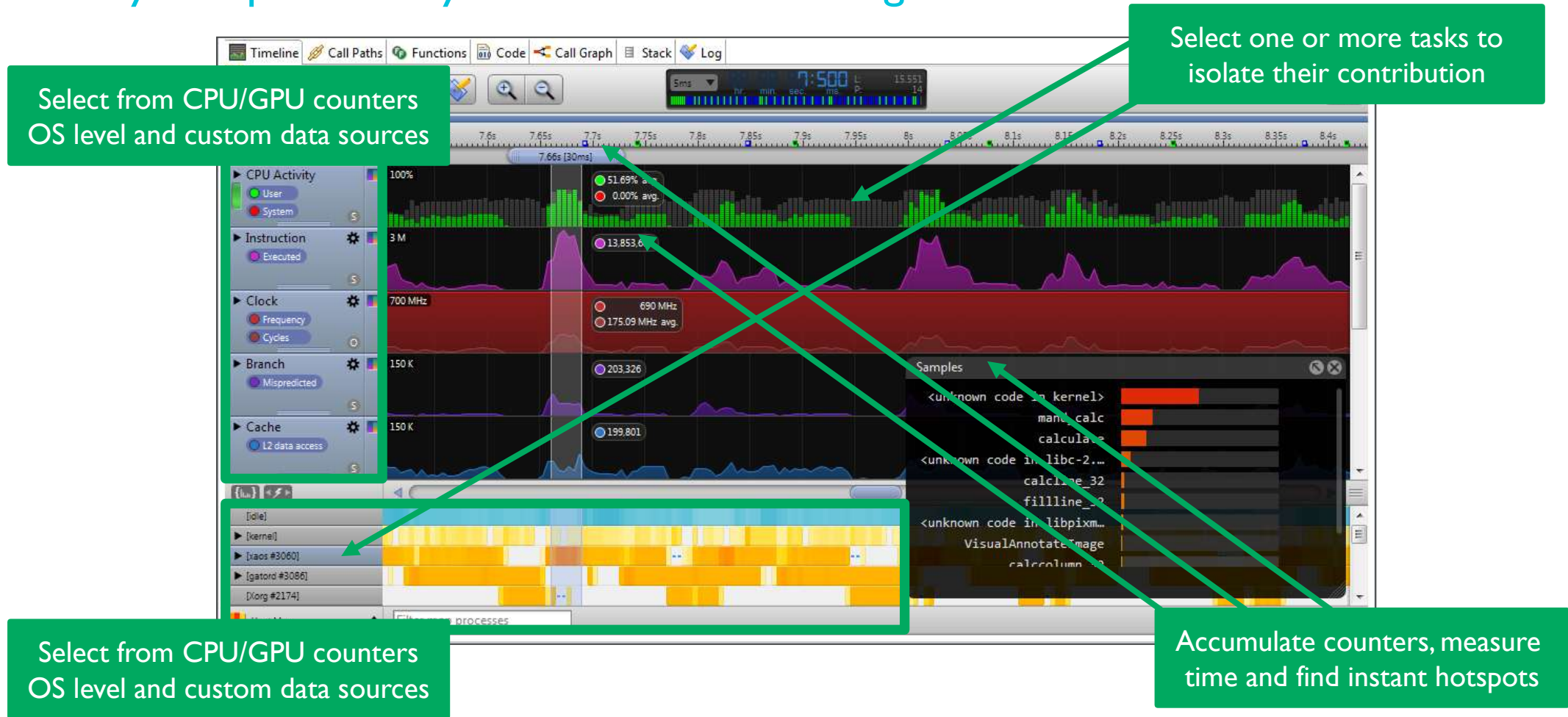
# Timeline: Heat Map

Identify hotspots and system bottlenecks at a glance



# Timeline: Heat Map

Identify hotspots and system bottlenecks at a glance





# Profiling Reports

## Analysis of call paths, source code and generated assembly

The image displays a profiling tool interface with three main panels. The left panel shows a call path tree, the middle panel shows the source code, and the right panel shows the disassembled assembly.

**Call Paths Panel:**

Self	Process	Total	Stack	Process/Thread/Function Name	Location
0.00%	100.00%	38.24%	0	[idle]	-
0.00%	100.00%	30.37%	0	[kernel]	-
0.00%	100.00%	22.45%	0	[threads]	-
0.00%	100.00%	6.75%	0	[xaos]	-
0.00%	100.00%	6.75%	0	[thread 1]	-
0.04%	60.58%	4.09%	176	main	ui.c:1074
0.04%	60.13%	4.06%	304	uih_do_fractal	ui_helper.c:912
10.56%	60.09%	4.05%	480	do_fractal	zoom.c:1538
17.31%	17.31%	1.17%	624	T.212	zoom.c:1023
3.72%	14.74%	0.99%	656	calcline_32	zoomd.c:31
5.57%	5.57%	0.38%	832	calcline_32	zoomd.c:31
0.04%	5.36%	0.36%	720	calculate	calculate.h:5
0.04%	0.04%	< 0.01%	720	mand_calc	docalc.c:133
0.04%	0.04%	< 0.01%	720	mand_peri	docalc.c:468
2.99%	10.15%	0.68%	656	calccolumn_32	zoomd.c:137
0.20%	3.72%	0.25%	720	calculate	calculate.h:5
3.19%	3.19%	0.22%	832	calccolumn_32	zoomd.c:137
0.16%	0.16%	0.01%	720	mand_calc	docalc.c:133
0.08%	0.08%	< 0.01%	720	mand_peri	docalc.c:468
5.94%	5.94%	0.40%	672	mkrealloc_table	zoom.c:444

**Call Paths Panel (Call Paths 1):**

Samples	Instances	Function Name	Location
28.81%	1	T.212	zoom.c:1023
17.57%	1	do_fractal	zoom.c:1538
15.46%	2	calcline_32	zoomd.c:31
15.05%	4	calculate	calculate.h:5
10.29%	2	calccolumn_32	zoomd.c:137
9.88%	1	mkrealloc_table	zoom.c:444
0.89%	3	addprices	zoom.c:1210
0.61%	1	mkfilltable	zoom.c:964

**Source Code Panel:**

```
107 float accumulate(unsigned int thread, float accum, unsigned int loops)
108 {
109     unsigned int i;
110
111     printf("Thread %d started accumulating\n", thread);
112     for (i=0; i<loops; i++)
113     {
114         accum = accum + step;
115         if (i==loops/2)
116         {
117             printf("Thread %d half way through accumulation\n", thread);
118         }
119     }
120 }
```

**Disassembly Panel:**

Samples	I	Address	Opcode	Disassembly
29.31%		0x0000890C	E1A00005	MOV r0, r5
10.34%		0x00008910	E1A01007	MOV r1, r7
		0x00008914	EB000116	BL __addsf3 ; 0x00008D74
34.48%		0x00008918	E1A05000	MOV r5, r0
		0x0000891C	E1580004	CMP r8, r4
17.24%		0x00008920	1A000002	BNE 0x00008930 ; accumulate + 0x5c
		0x00008924	E1A00009	MOV r0, r9
		0x00008928	E1A0100A	MOV r1, r10
		0x0000892C	EBFFFF2C	BL {pc}-0x348 ; 0x85e4
		0x00008930	E2844001	ADD r4, r4, #1
8.62%		0x00008934	E1560004	CMP r6, r4
		0x00008938	8AFFFFFF	BHT 0x0000890C ; accumulate + 0x38

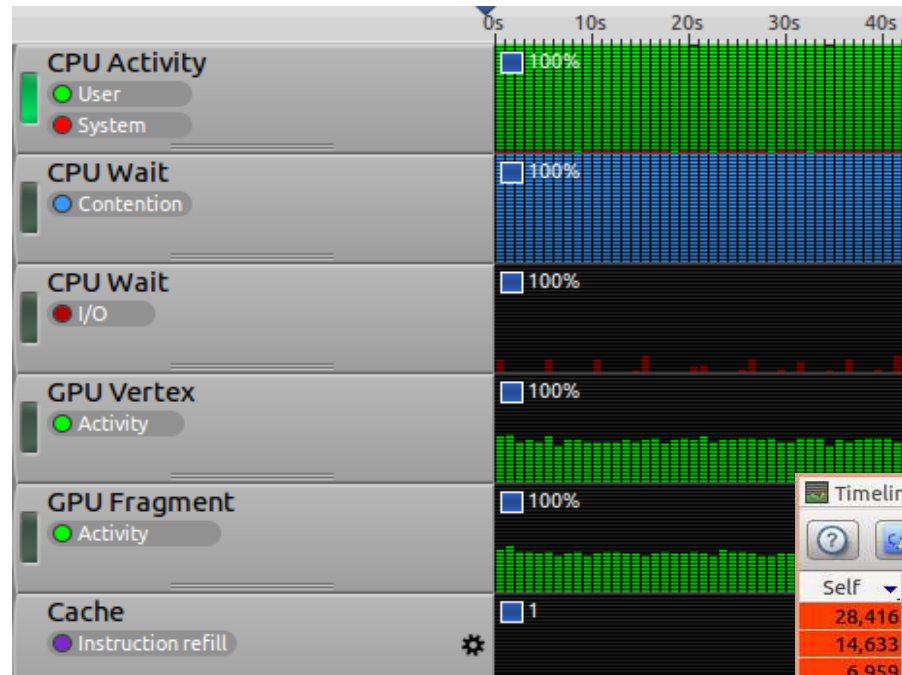
**Summary:**

Total: 43 (5.73%)

Find:



# Looking for the Bottleneck



As expected, CPU is maxed out by the benchmark

Simple profiling shows **memcpy** as hotspot

Timeline Call Paths Functions Code Call Graph Stack Log Warnings				
Functions: 1 Samples (Self): 28,416 (44.15%)				
Self	% Self	Instances	Function Name	Location
28,416	44.15%	16	memcpy	memcpy.S:43
14,633	22.74%	1	sc8810_deep_sleep	pm_sc8810.c:377
6,959	10.81%	4	cocos2d::CCSprite::updateTransform()	CCSprite.cpp:481
2,718	4.22%	1	IterateSpriteSheetCArray::update(float)	PerformanceNodeChildrenTest.cpp:268
2,298	3.57%	1	cocos2d::CCSprite::setVisible(bool)	CCSprite.cpp:887
1,347	2.09%	1	cocos2d::CCSpriteBatchNode::draw()	CCSpriteBatchNode.cpp:392
734	1.14%	3	cocos2d::CCTextureAtlas::updateQuad(cocos2d::_ccV3F...	CCTextureAtlas.cpp:316
374	0.58%	42	[gator]	<anonymous>
274	0.43%	41	_kprobes_text_end	entry-armv.S:900
215	0.33%	37	dalvik_inst	InterpAsm-armv7-a-neon.S:389
198	0.31%	6	scanObject(const Object*, GcMarkContext*)	MarkSweep.cpp:455
193	0.30%	8	[mali]	<anonymous>

# Optimizing

Optimization work led to reduced time in memcpy

Self	% Self	Instances	Function Name	Self	% Self	Instances	Function Name
5,966	54.62%	6	memcpy	3,807	35.01%	3	cocos2d::CCSprite::updateTransform()
1,678	15.36%	1	cocos2d::CCSprite::updateTransform()	1,030	9.47%	1	cocos2d::CCNode::updateTransform()
628	5.75%	1	IterateSpriteSheetCArray::update(float)	994	9.14%	1	IterateSpriteSheetCArray::update(float)
525	4.81%	1	cocos2d::CCSprite::setVisible(bool)	990	9.10%	1	memcpy
280	2.56%	1	cocos2d::CCSpriteBatchNode::draw()	899	8.27%	2	cocos2d::CCSprite::isQuadColorsDirty()
164	1.50%	1	cocos2d::CCTextureAtlas::updateQuad(co	795	7.31%	1	cocos2d::CCSprite::setVisible(bool)
87	0.80%	15	[gator]	161	1.48%	1	cocos2d::CCSpriteBatchNode::draw()
57	0.52%	1	sc8810_idle	161	1.48%	1	cocos2d::CCTextureAtlas::updateQuad(co
51	0.47%	1	cocos2d::CCNode::updateTransform()	121	1.11%	17	[gator]
50	0.46%	12	_schedule	75	0.69%	15	_schedule
41	0.38%	1	scanObject(const Object*, GcMarkContex				
32	0.29%	1	_delay				

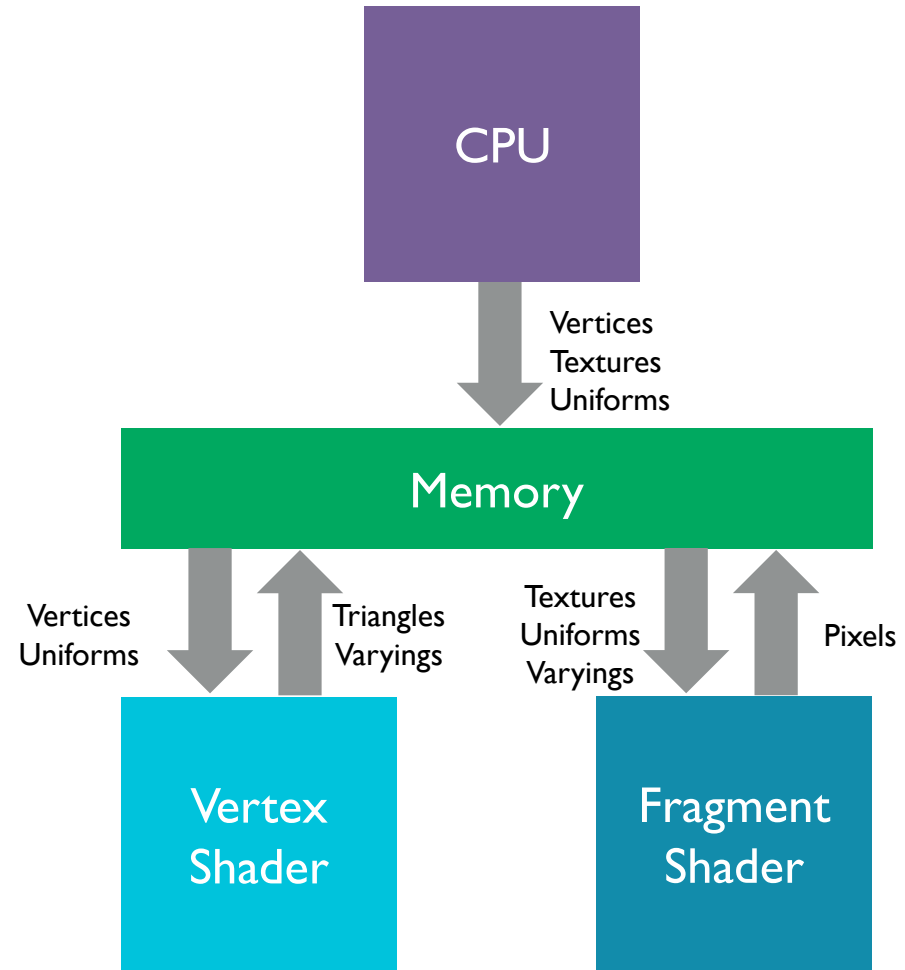
Real world optimization of 6FPS from memcpy alone



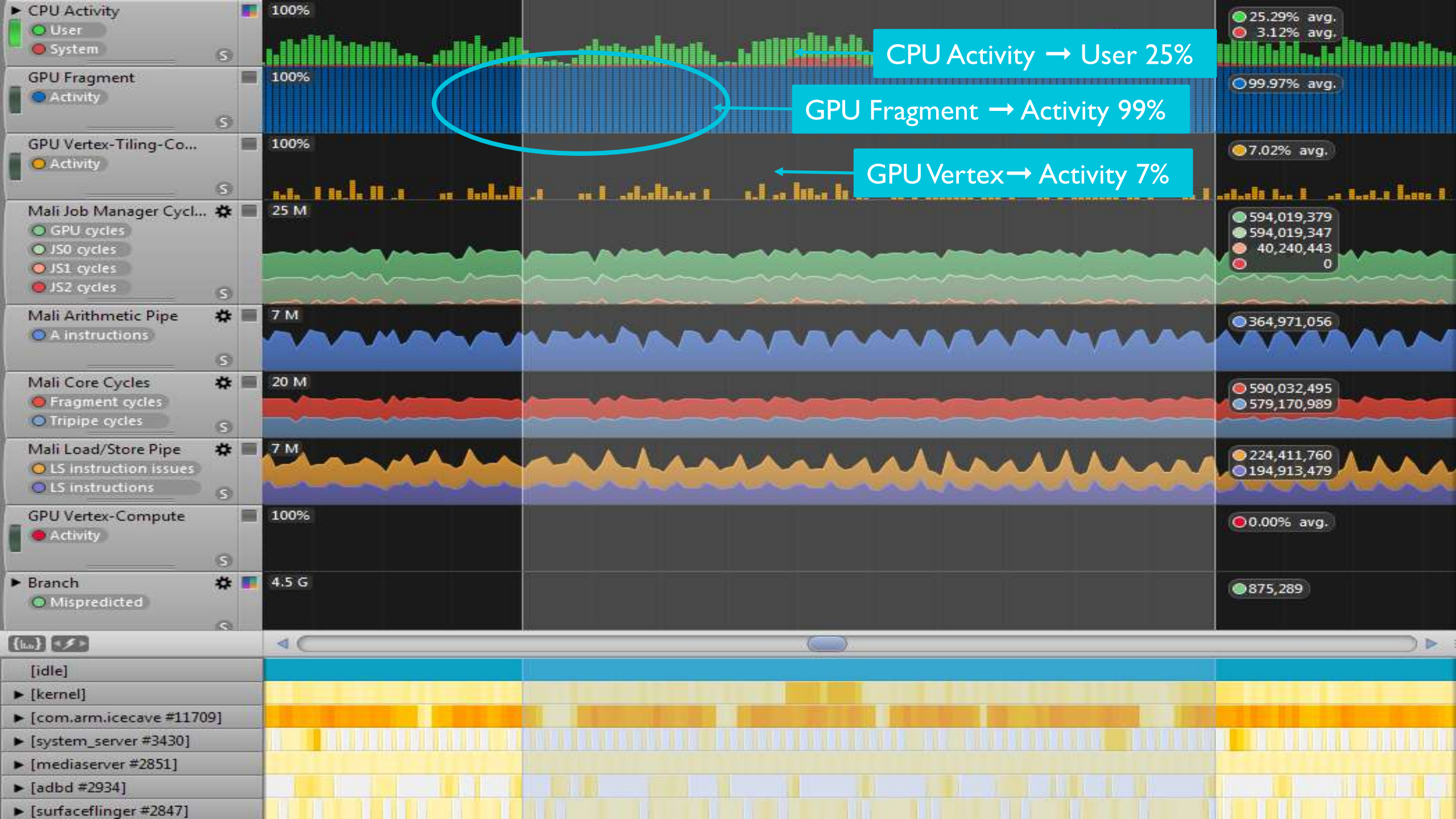
52%  
faster

# Main Bottlenecks

- CPU
  - Too many draw calls
  - Complex physics
- Vertex processing
  - Too many vertices
  - Too much computation per vertex
- Fragment processing
  - Too many fragments, overdraw
  - Too much computation per fragment
- Bandwidth
  - Big and uncompressed textures
  - High resolution framebuffer







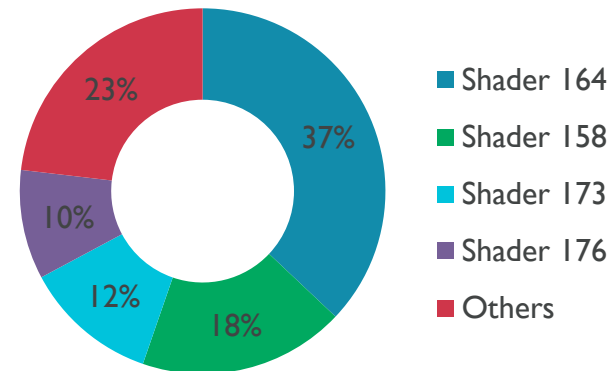
# Vertex Count and Shader Optimizations

Identify the top heavyweight vertex shaders

```
iceCaveWithCaptures.mgd Shader164 ✕  
  
in vec4 _glesVertex;  
in vec3 _glesNormal;  
in vec4 _glesMultiTexCoord0;  
in vec4 _glesTANGENT;  
uniform highp mat4 glstate_matrix_mvp;  
uniform highp mat4 _Object2World;  
uniform highp mat4 _World2Object;  
uniform mediump vec3 _LightPos01;  
out mediump vec2 xlv_TEXCOORD0;  
out mediump vec4 xlv_TEXCOORD1;  
out mediump vec3 xlv_TEXCOORD2;  
out mediump vec3 xlv_TEXCOORD3;  
out mediump vec3 xlv_TEXCOORD4;  
out mediump vec3 xlv_TEXCOORD5;  
out mediump vec4 xlv_TEXCOORD6;  
out mediump vec4 xlv_projTexCoordOut_projTexCoord;  
out mediump float xlv_projTexCoordOut_rangeFade;  
out mediump vec4 xlv_TEXCOORD19;  
void main ()  
{  
    mediump vec4 tmpvar_1;  
    mediump vec3 tmpvar_2;  
    mediump vec3 tmpvar_3;  
    mediump vec4 tmpvar_4;  
    mediump vec4 tmpvar_5;  
    mediump float tmpvar_6;  
    mediump vec4 tmpvar_7;  
    highp vec4 tmpvar_8;  
    tmpvar_8 = (glstate_matrix_mvp * _glesVertex);
```

Program	Name	A	L/S	T	Total	Vertices	Total cycles	% cycles
163	Shader164	24	16	0	40	95,856	2,683,968	37.0%
157	Shader158	32	25	0	57	32,256	1,322,496	18.2%
172	Shader173	44	32	0	76	16,893	861,543	11.9%
175	Shader176	47	28	0	75	14,010	700,500	9.6%
171	Shader169	9	10	0	19	35,739	536,085	7.4%
97	Shader98	8	3	0	11	70,722	495,054	6.8%
154	Shader155	25	20	0	45	12,240	403,920	5.6%
195	Shader193	9	13	0	22	3,360	60,480	0.8%
130	Shader131	10	6	0	16	4,836	58,032	0.8%
160	Shader161	25	20	0	45	984	32,472	0.4%
166	Shader167	38	31	0	69	432	21,168	0.3%
85	Shader86	11	6	0	17	1,752	21,024	0.3%
187	Shader188	9	7	0	16	1,176	14,112	0.2%
181	Shader182	38	20	0	58	324	13,608	0.2%
109	Shader110	24	7	0	31	600	10,800	0.1%
82	Shader83	9	7	0	16	858	10,296	0.1%

Vertex Cycles  
Per Program





# Inspect the Tripipe Counters

Reduce the load on the Arithmetic pipeline

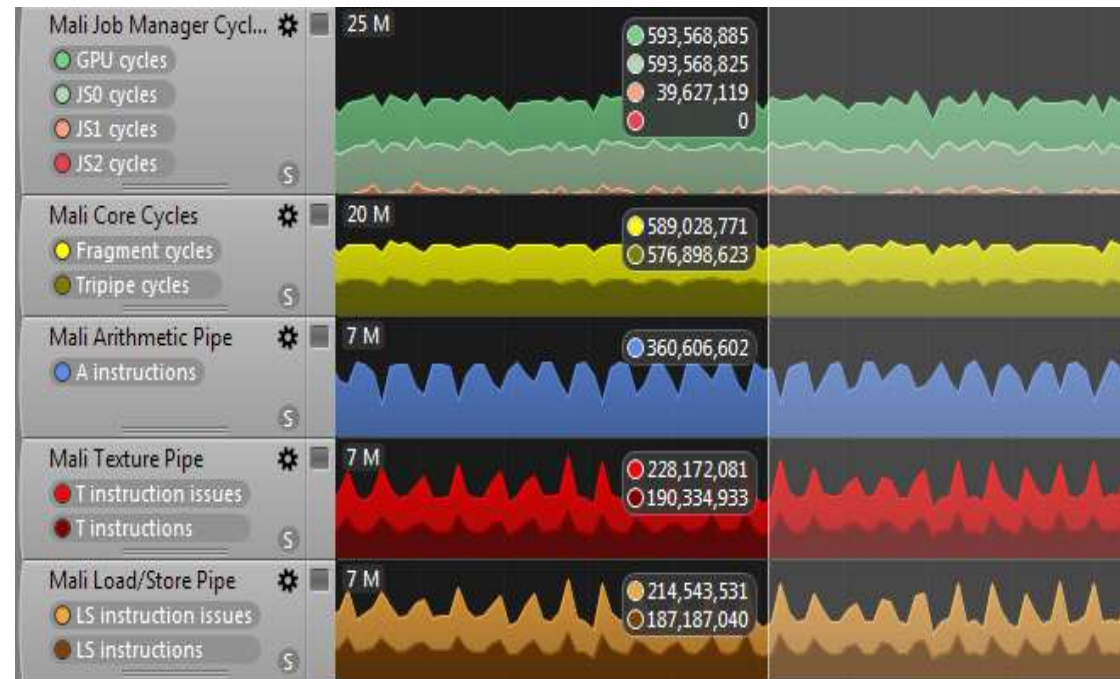
GPU Cycles 593m

Tripipe Cycles 576m

Arithmetic 361m

Texture 228m

Load & Store 214m



# Tripipes Counters

## Cycles per instruction metrics

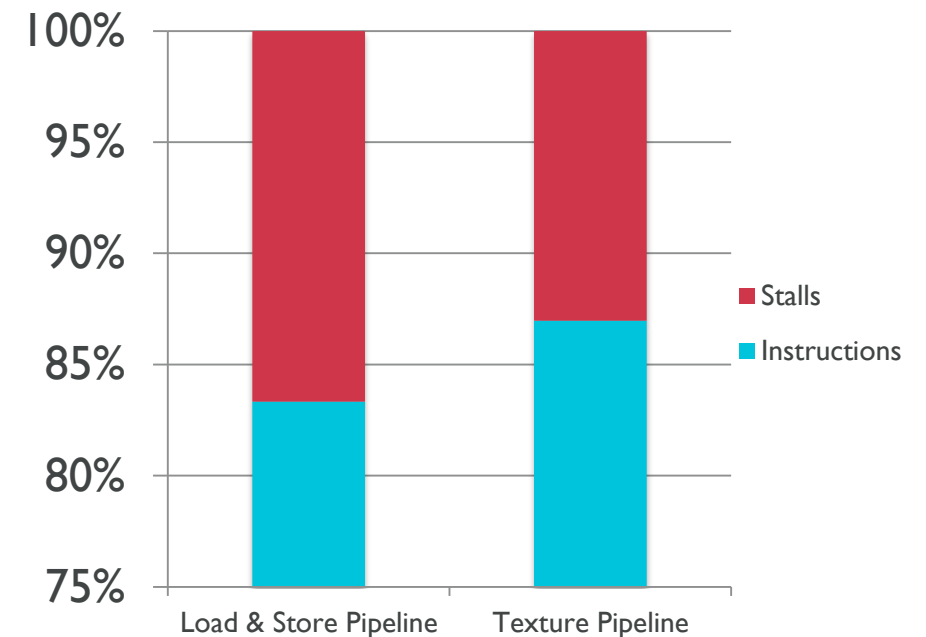
- It's easy to calculate a couple of CPI (cycles per instruction) metrics:

- For the load/store pipeline we have:

$$\begin{aligned} & 228\text{m (Mali Load/Store Pipe} \rightarrow \text{LS instruction issues)} \\ & / 190\text{m (Mali Load/Store Pipe} \rightarrow \text{LS instructions)} \\ & = 1.2 \text{ cycles/instruction} \end{aligned}$$

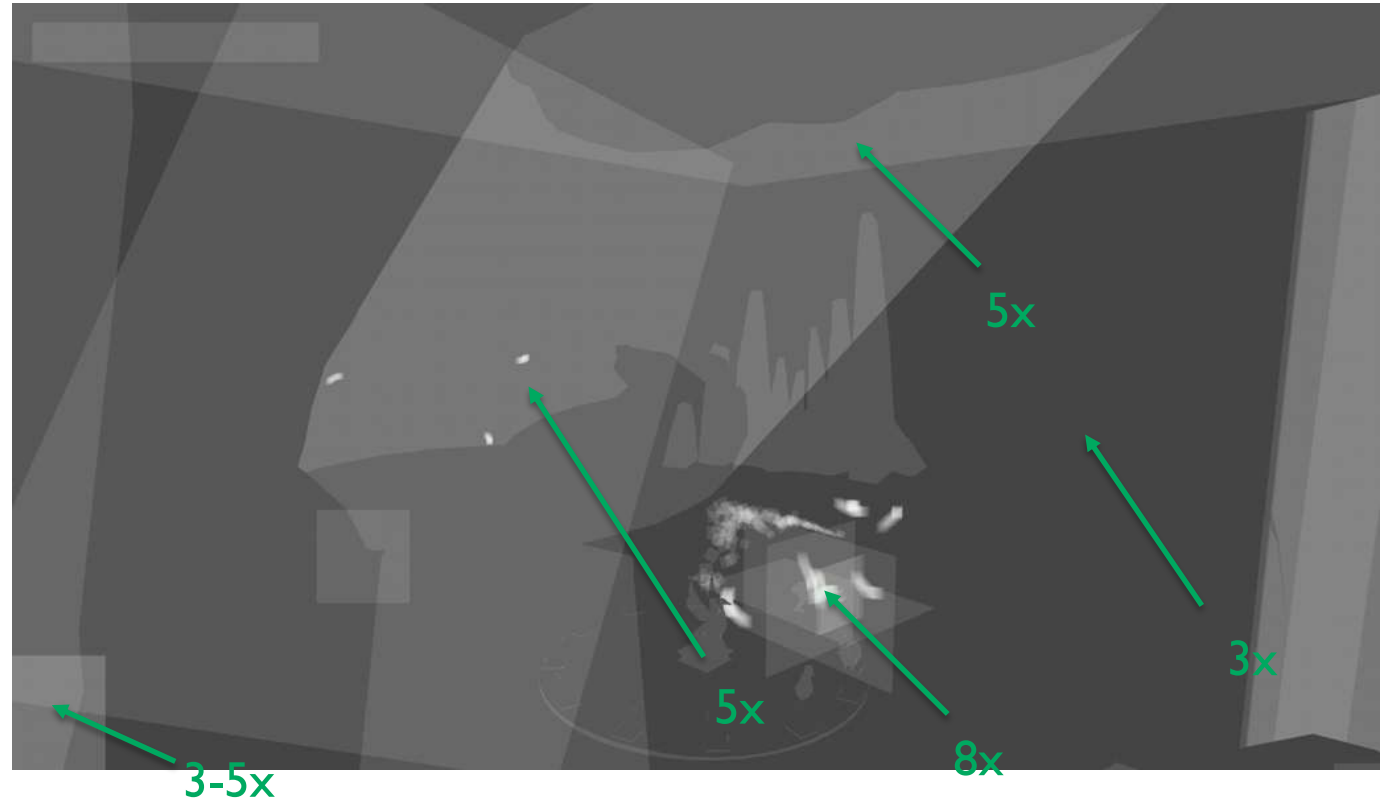
- For the texture pipeline we have:

$$\begin{aligned} & 215\text{m (Mali Texture Pipe} \rightarrow \text{T instruction issues)} \\ & / 187\text{m (Mali Texture Pipe} \rightarrow \text{T instructions)} \\ & = 1.15 \text{ cycles/instruction} \end{aligned}$$



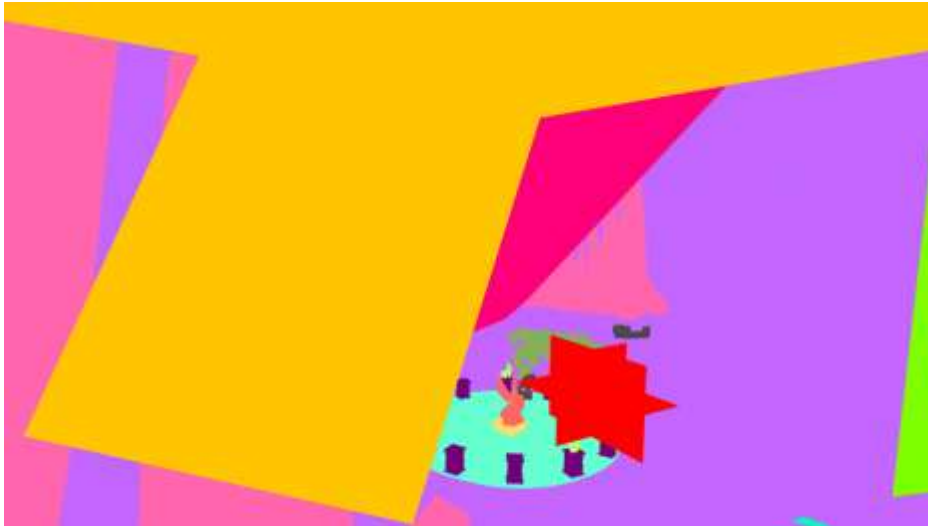
# Frame Analysis

Check the overdraw factor



# Shader Map and Fragment Count

Identify the top heavyweight fragment shaders



Assets

Vertex Shaders

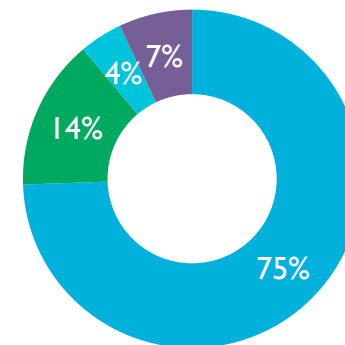
Fragment Shaders

Textures

Program	Name	Instructions	Shortest	Longest	Instances	Total cycles
175	Shader 177	5	5	5	7537773	37688865
280	Shader 282	5	5	5	1459254	7296270
181	Shader 183	5	5	5	415710	2078550
187	Shader 189	6	6	6	197329	1183974
73	Shader 75	4	4	4	279555	1118220
382	Shader 384	8	8	8	129913	1039304
289	Shader 291	6	6	6	16856	101136
208	Shader 210	7	3	6	7975	39875
262	Shader 264	5	5	5	6025	30125
400	Shader 402	5	5	5	914	4570

Fragment Count Per Program

~10m instances  
/ (2560×1600) pixel  
= 2.44

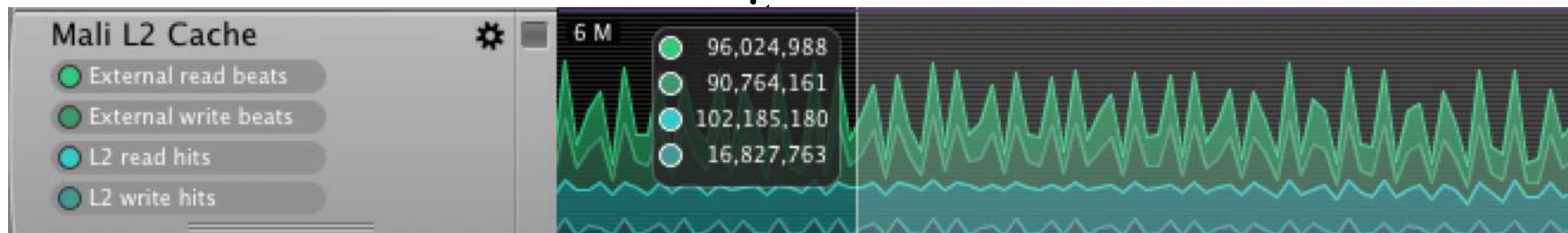


- Program 175
- Program 280
- Program 181
- Others

# Bandwidth

- When creating embedded graphics applications, bandwidth is a scarce resource
  - A typical embedded device can handle 5.0 Gigabytes a second of bandwidth
  - A typical desktop GPU can do **in excess of 100** Gigabytes a second
- The application is **not bandwidth bound** as it performs, over a period of one second:

$$(96\text{m (Mali L2 Cache} \rightarrow \text{External read beats)} + 90.7\text{m (Mali L2 Cache} \rightarrow \text{External write beats)}) \times 16 \\ \approx 2.9 \text{ GB/s}$$
- Since bandwidth usage is related to energy consumption it's always worth optimizing





# Practical Optimizations

**ARM**

Stacy Smith

ARM Senior Software Engineer

GDC

March 2016

# What MGD Can Tell Us

Capturing a full frame render tells us:

- What order things are drawn in
- Whether things are drawn needlessly
- How much state we're shifting each time

# What Streamline Can Tell Us

Capturing a trace tells us:

- What our process bottlenecks are
- How much bandwidth we consume
- Calculations such as average triangle size

# But why??



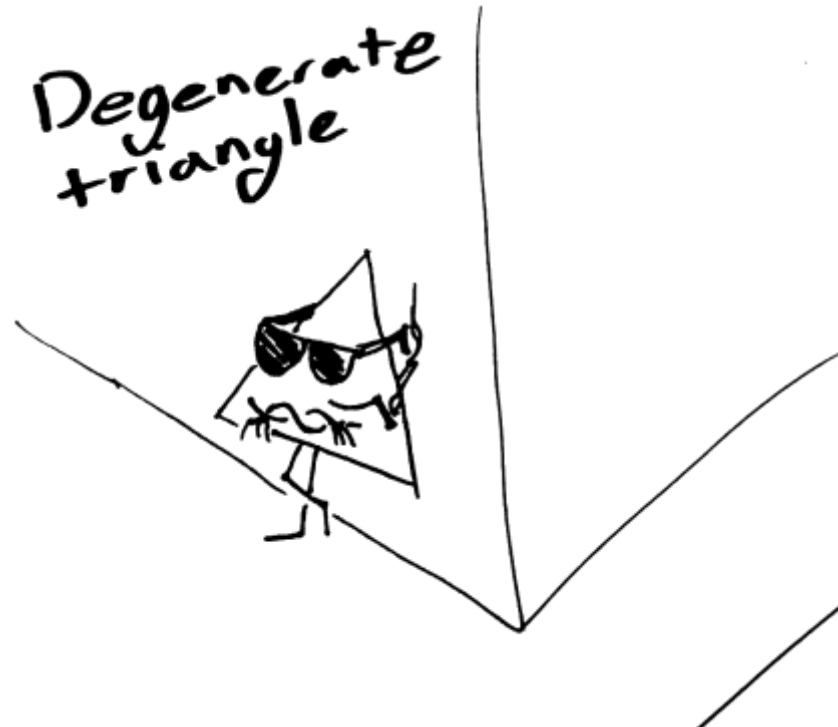
# The Sensible Six Best Practices

- Batching
- Overdraw
- Culling
- LoD
- Compression
- Antialiasing



# Draw Calls

- CPU load affected by `glDrawElements` & `glDrawArrays`
- Immediate performance win: put multiple static meshes in the same model, possibly connected by degenerate triangles



# Batched Draw Calls

- Can be as simple as putting whole static scenes in a single mesh
- Combined static meshes can be generated at build time for better level design flexibility
- But what about non static objects?

# Dynamically Batched

Using vertex attributes to tag & parameterize meshes

Different mesh ids can use:

- Different transforms
- Different colours
- Different textures
- Different shader branch

Know your limits!

# Overdraw

- Wasted calculations for any overdrawn pixels
- Bad for fragment load
- Complicated ways to do this with batches, instancing and sorted indexed transformations
- But also easy design choices

# Eliminating Overdraw

## Draw order sequencing in your scene

- Some things are always in front:
  - The contents of convex spaces
  - HUDs
- Some things are always behind:
  - Skyboxes
  - The ground



# Minimising Overdraw Impact

Statistically some things are USUALLY in front

- Two things may have equal possibility of foregroundness, but one is more expensive to get wrong
- Cheap depth pre-pass if both are expensive to draw

# Culling

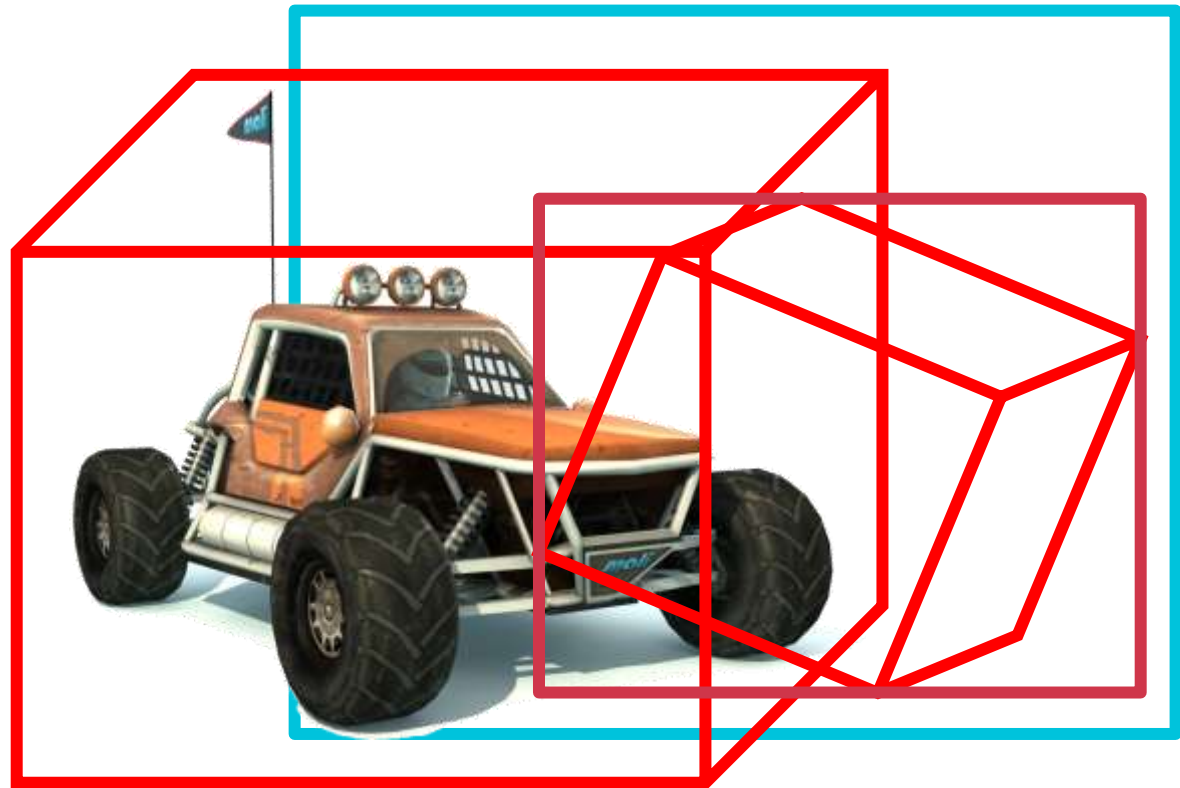
Culling can help with vertex and sometimes CPU load

If you can quickly decide if something isn't visible, don't draw it.

# Frustum Culling

Off screen triangles are culled anyway, but they all have to be transformed

- CPU side bounding box culling
- Even easier for grid layouts

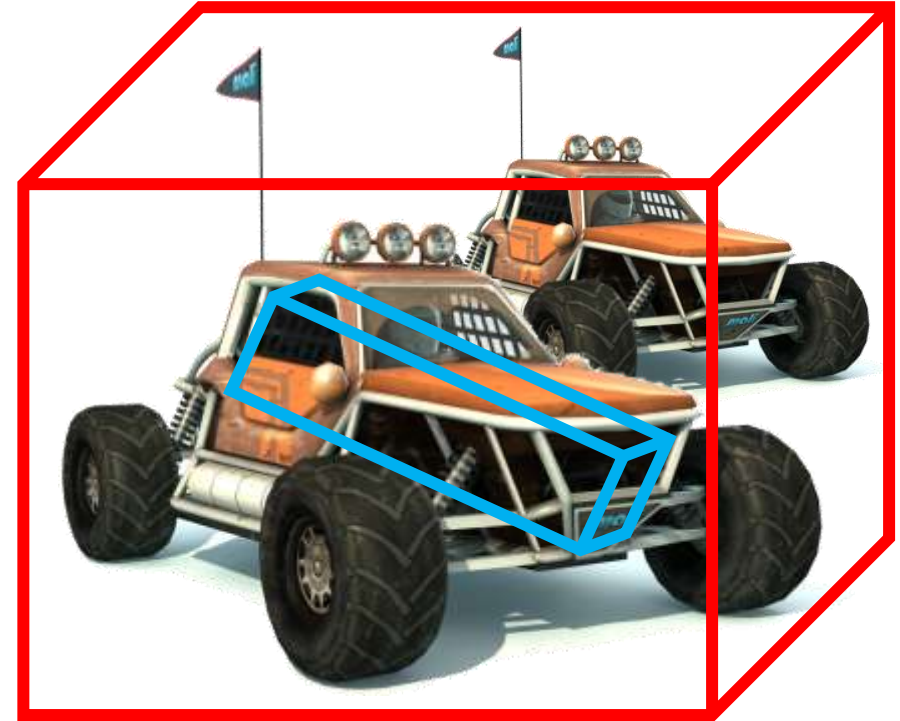


# Occlusion Culling

Ultimate overdraw reduction:

If an object is wholly obfuscated, why draw it?

- Complex task for individual objects
- Surprisingly easy for linked interior spaces (portal culling)



# Distance Culling

Why draw stuff too far away to see?

- False horizon
- Fogging

Or just make it easier to draw...



# Level of Detail

- Not just vertex waste
- Triangle load
- Avoid triangles with pixel coverage in single digits
- If something on screen gets tiny, replace it with something simpler

# Level of Detail

Can be done in simple or complex ways

- Pick a good base detail level
- Often just having a high and low detail model is enough
- Parameterising LoD with draw call batching  
(A crazy cool technique not enough people are using)

# Impostor!

At the greatest distances, a sprite may suffice

Separate batch of billboards

Helps with perspective:

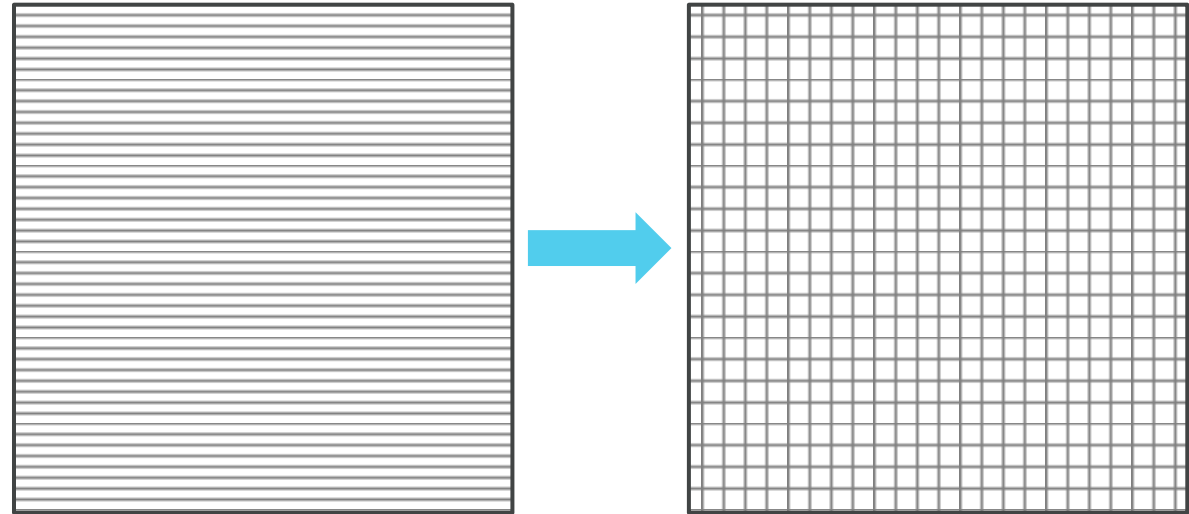
- A few objects fit in the foreground
- Many, many more fit in the background

# Texture Compression

There's something about textures that not a lot of people know

- Textures get blocked automatically

- Good for the cache!
- Compression replicates this



# Compression Formats

What doesn't work?

- ~~Linear encodings~~
- ~~Progressive encodings~~
- ~~Dictionary based encoding~~
- ~~Variable output size~~

Formats?

- ASTC
- ETC 1 & 2

What works?

- Block based
- Deterministic
- Immediate
- Random lookup



# Mip Mapping

Mip Mapping is an all round best buddy:

- Less interference noise
- Better for the cache
- Reduces bandwidth
- ~~▪ Will help you hide a body~~

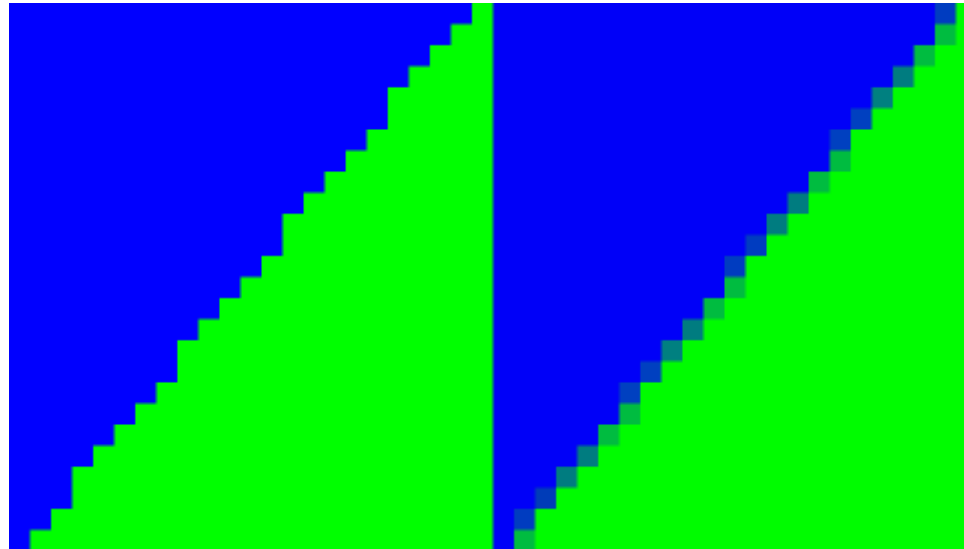
But you'll need to compress mipmaps too



# Antialiasing

Get the most from every pixel

- The cost of full screen 4x MSAA is negligible on Mali GPUs



Antialiasing sample code can be found in the Mali SDKs

<http://malideveloper.arm.com/resources/sdks/>

# Drilling Down

- Ice Cave
- 7 render passes (shown as 9)
- 206 drawcalls



▼ 📷 Frame 23 307743 vertices, 206 draws, 61648 unique indices, 9 render passes, 148312270 bytes

- ▶ 🖼️ RenderPass 0 (FrameBuffer 0) 0 vertices, 0 draws, 0 unique indices
- ▶ 🖼️ RenderPass 1 (FrameBuffer 13) 73194 vertices, 32 draws, 14829 unique indices
- ▶ 🖼️ RenderPass 2 (FrameBuffer 14) 0 vertices, 1 draw, 0 unique indices
- ▶ 🖼️ RenderPass 3 (FrameBuffer 11) 75576 vertices, 42 draws, 15174 unique indices
- ▶ 🖼️ RenderPass 4 (FrameBuffer 12) 0 vertices, 1 draw, 0 unique indices
- ▶ 🖼️ RenderPass 5 (FrameBuffer 1) 158967 vertices, 128 draws, 31641 unique indices
- ▶ 🖼️ RenderPass 6 (FrameBuffer 0) 0 vertices, 0 draws, 0 unique indices
- ▶ 🖼️ RenderPass 7 (FrameBuffer 2) 0 vertices, 1 draw, 0 unique indices
- ▶ 🖼️ RenderPass 8 (FrameBuffer 0) 6 vertices, 1 draw, 4 unique indices

# Ice Cave's Passes

- 2 'passes' are nothing
- Pass 2, 4 and 7 are buffer blits
- Pass 8 is composition
- Pass 1 is shadows
- Pass 3 is reflections
- Pass 5 is the main event

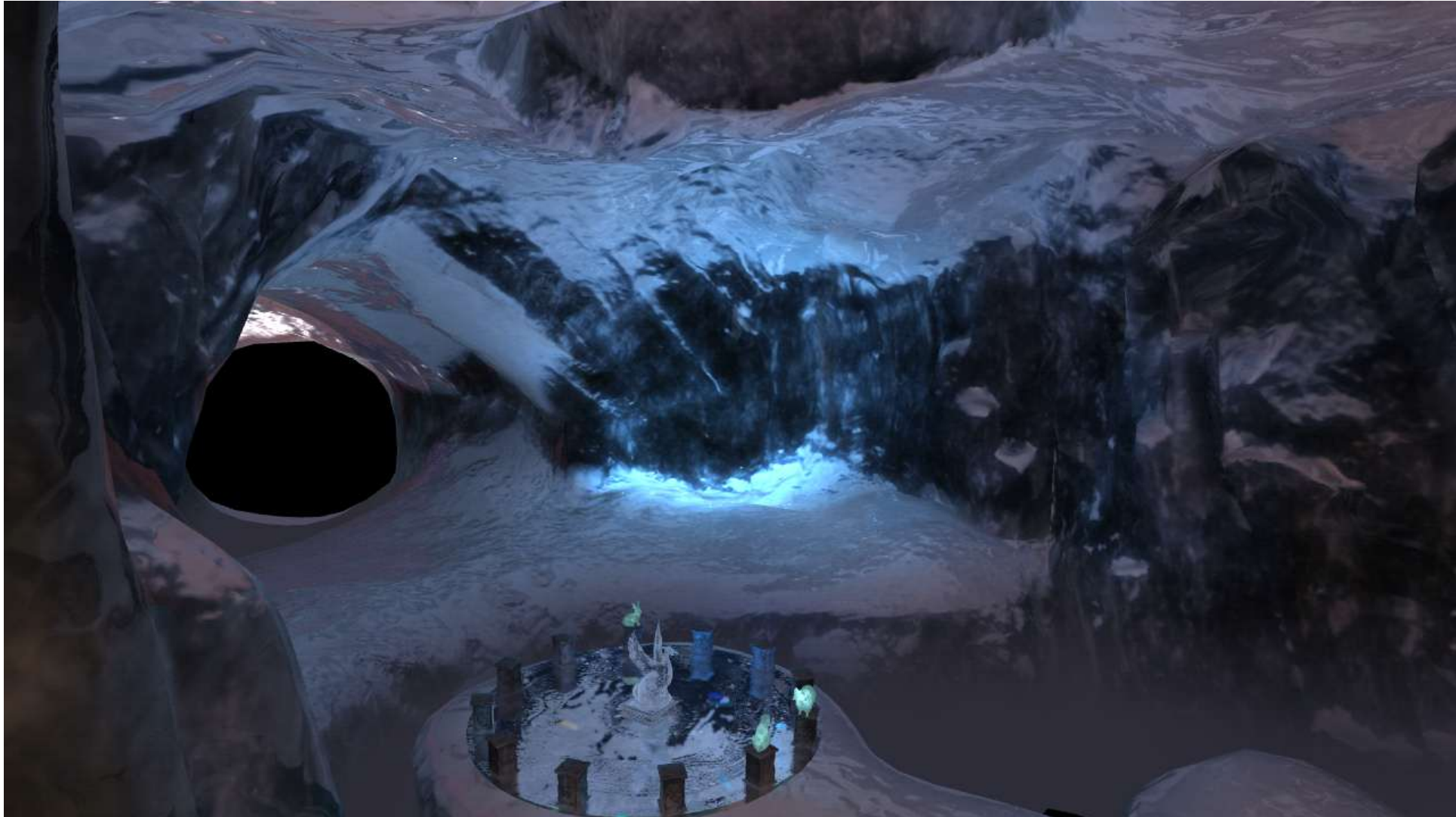
▼ 📷 Frame 23 307743 vertices, 206 draws, 61648 unique indices, 9 render passes, 148312270 bytes

RenderPass 0 (Framebuffer 0)	0 vertices, 0 draws, 0 unique indices
RenderPass 1 (Framebuffer 13)	73194 vertices, 32 draws, 14829 unique indices
RenderPass 2 (Framebuffer 14)	0 vertices, 1 draw, 0 unique indices
RenderPass 3 (Framebuffer 11)	75576 vertices, 42 draws, 15174 unique indices
RenderPass 4 (Framebuffer 12)	0 vertices, 1 draw, 0 unique indices
RenderPass 5 (Framebuffer 1)	158967 vertices, 128 draws, 31641 unique indices
RenderPass 6 (Framebuffer 0)	0 vertices, 0 draws, 0 unique indices
RenderPass 7 (Framebuffer 2)	0 vertices, 1 draw, 0 unique indices
RenderPass 8 (Framebuffer 0)	6 vertices, 1 draw, 4 unique indices

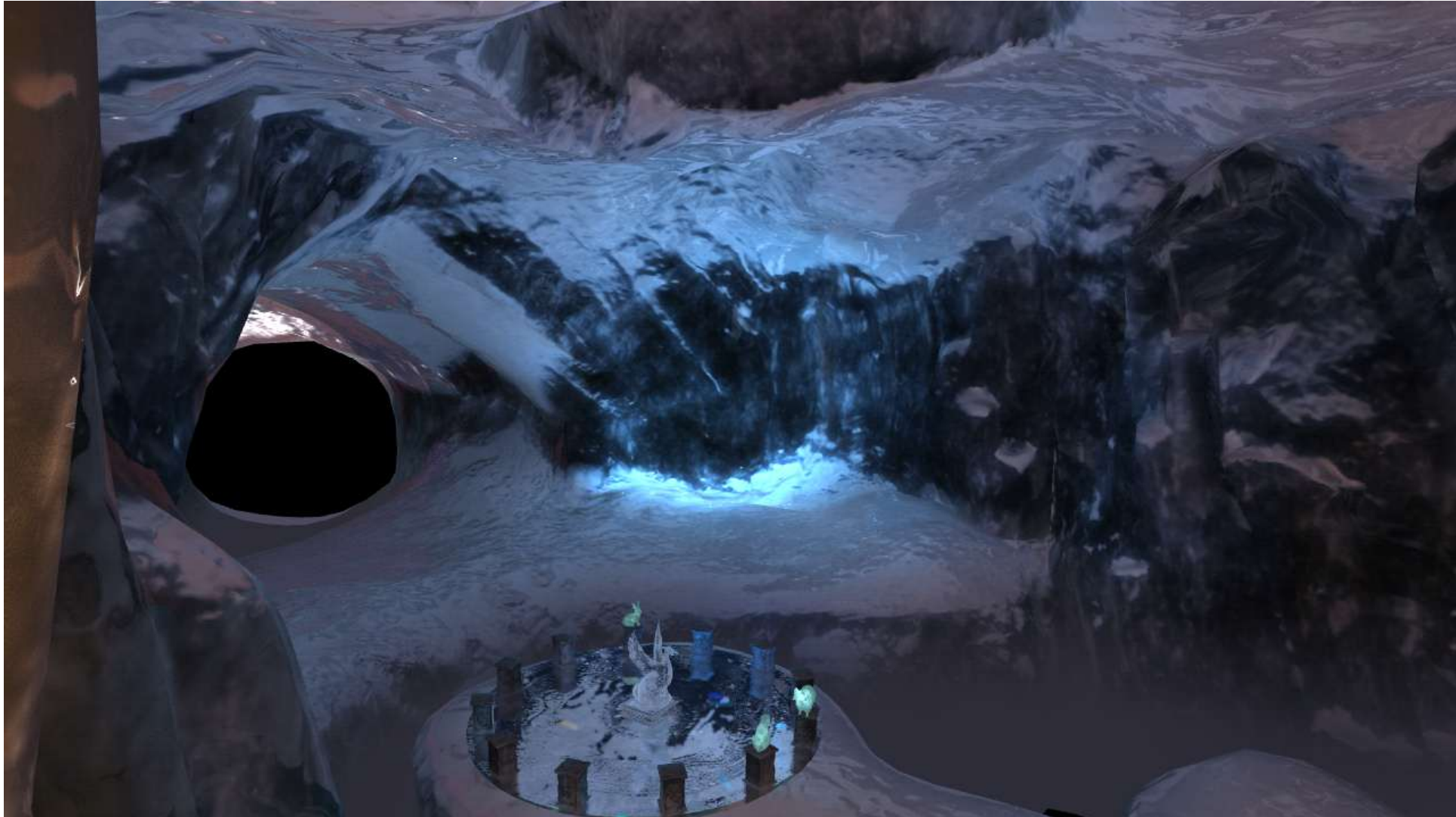




# Ice Cave's Layers (Overdraw)



# Ice Cave's Layers (Batching)



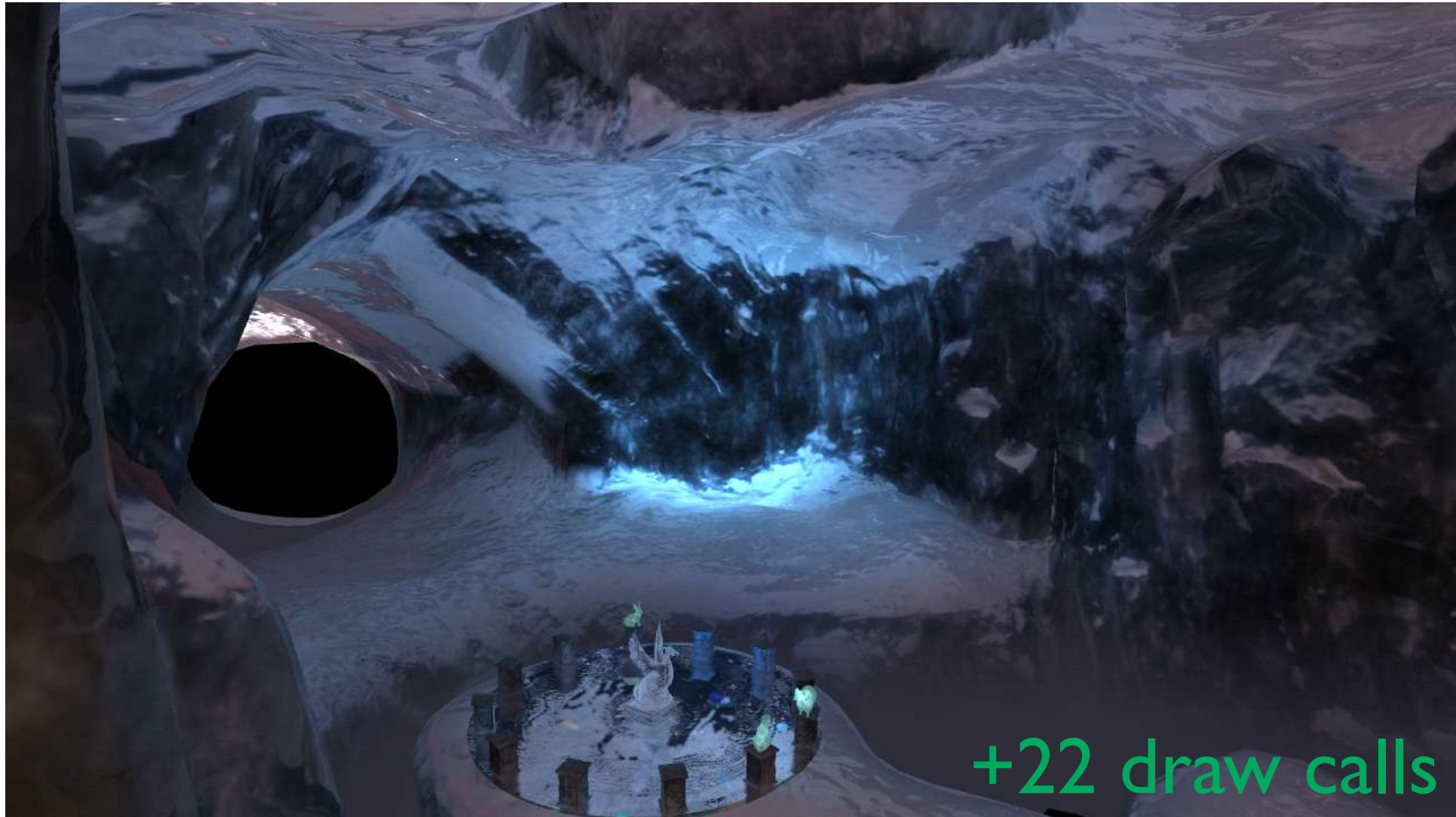


# Ice Cave's Layers (Skybox)





# Ice Cave's Layers (Culling)



# Ice Cave's Layers (Further Overdraw)





# Ice Cave's Layers (Further Overdraw)



# In conclusion

## Best Practices:

- Batching
- Overdraw
- Culling
- LoD
- Compression
- Antialiasing



# Thank you!

The ARM logo is displayed in a bold, white, sans-serif font. It is positioned on the left side of the slide, with the letters 'A', 'R', and 'M' stacked vertically and slightly overlapping.

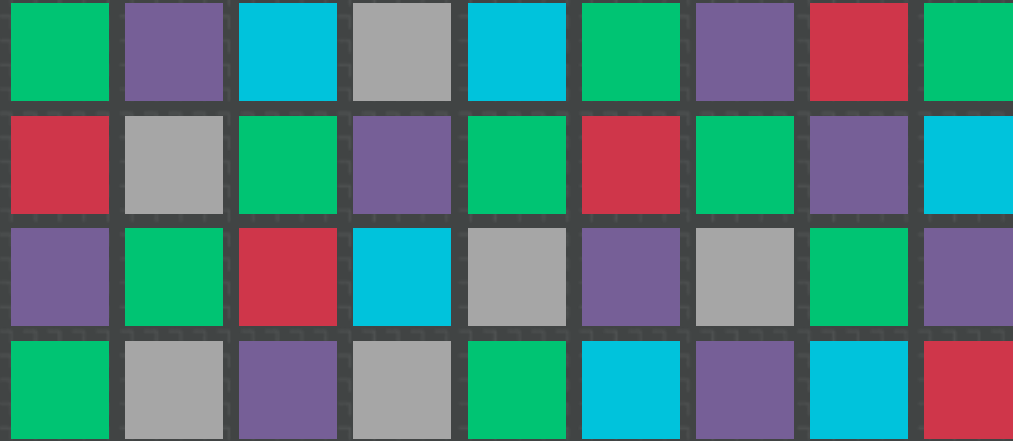
More slides:

<http://malideveloper.arm.com/gdc2016>

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

Copyright © 2016 ARM Limited

# To Find Out More....

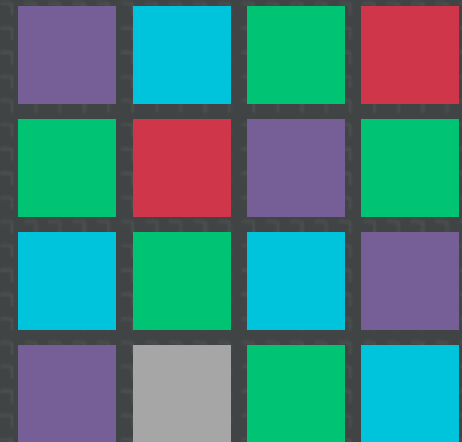


## ARM Booth #1624 on Expo Floor:

- Live demos of the techniques shown in this session
- In-depth Q&A with ARM engineers
- More tech talks at the ARM Lecture Theatre

[http://malideveloper.arm.com/gdc2016:](http://malideveloper.arm.com/gdc2016)

- Revisit this talk in PDF and video format post GDC
- Download the tools and resources



# More Talks From ARM at GDC 2016



Available post-show at the Mali Developer Center: [malideveloper.arm.com/](http://malideveloper.arm.com/)



## Vulkan on Mobile with Unreal Engine 4 Case Study

Weds. 9:30am, West Hall 3022



## Making Light Work of Dynamic Large Worlds

Weds. 2pm, West Hall 2000



## Achieving High Quality Mobile VR Games

Thurs. 10am, West Hall 3022



## Optimize Your Mobile Games With Practical Case Studies

Thurs. 11:30am, West Hall 2404



## An End-to-End Approach to Physically Based Rendering

Fri. 10am, West Hall 2020