Optimize Your Mobile Games With Practical Case Studies

ARM

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

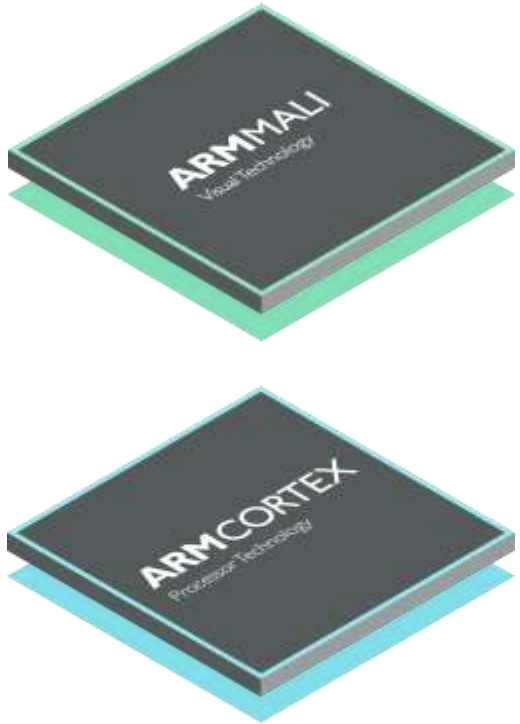
Importance of Analysis & Debug

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

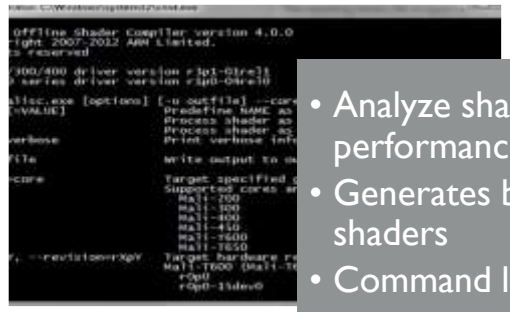
Mali Graphics Debugger

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

ARM DS-5 Streamline

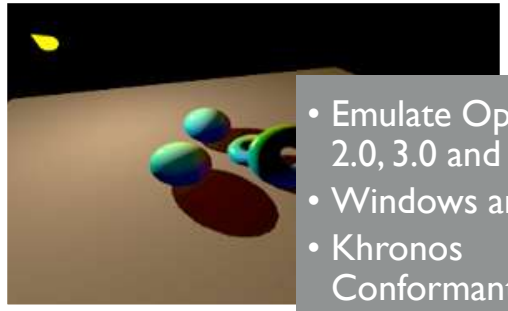
Mali GPU
- Timeline
- HW Counters
- OpenCL visualizer

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

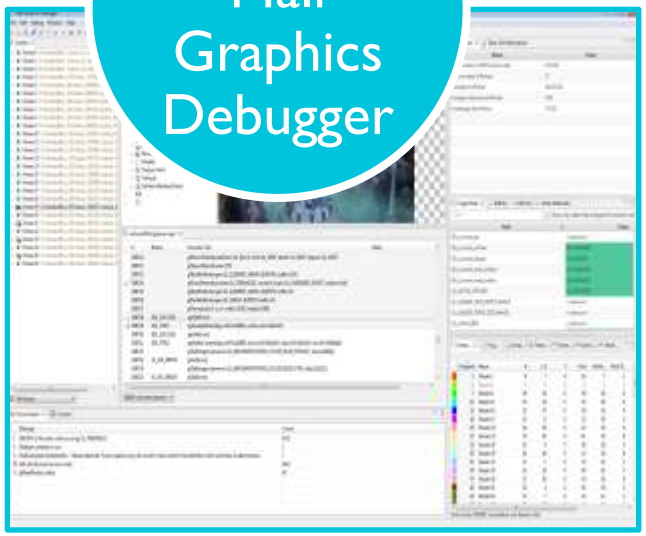
Third Party Tools

Integration with partners' tools

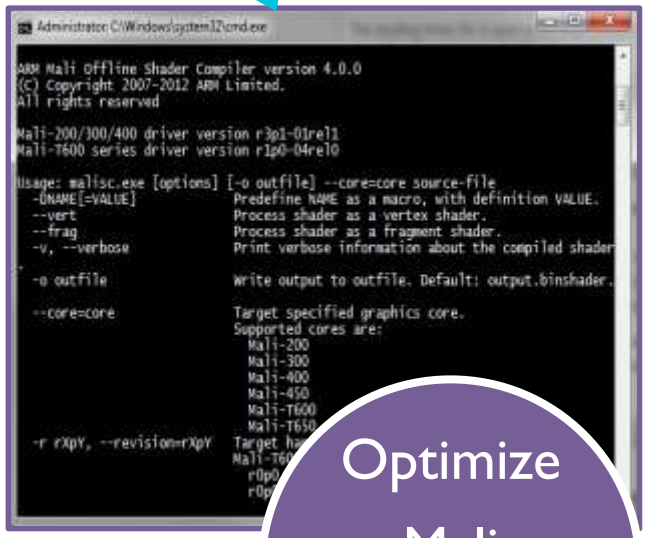
ARM

Performance Analysis and Debug With Mali GPUs

Analyze
DS-5
Streamline



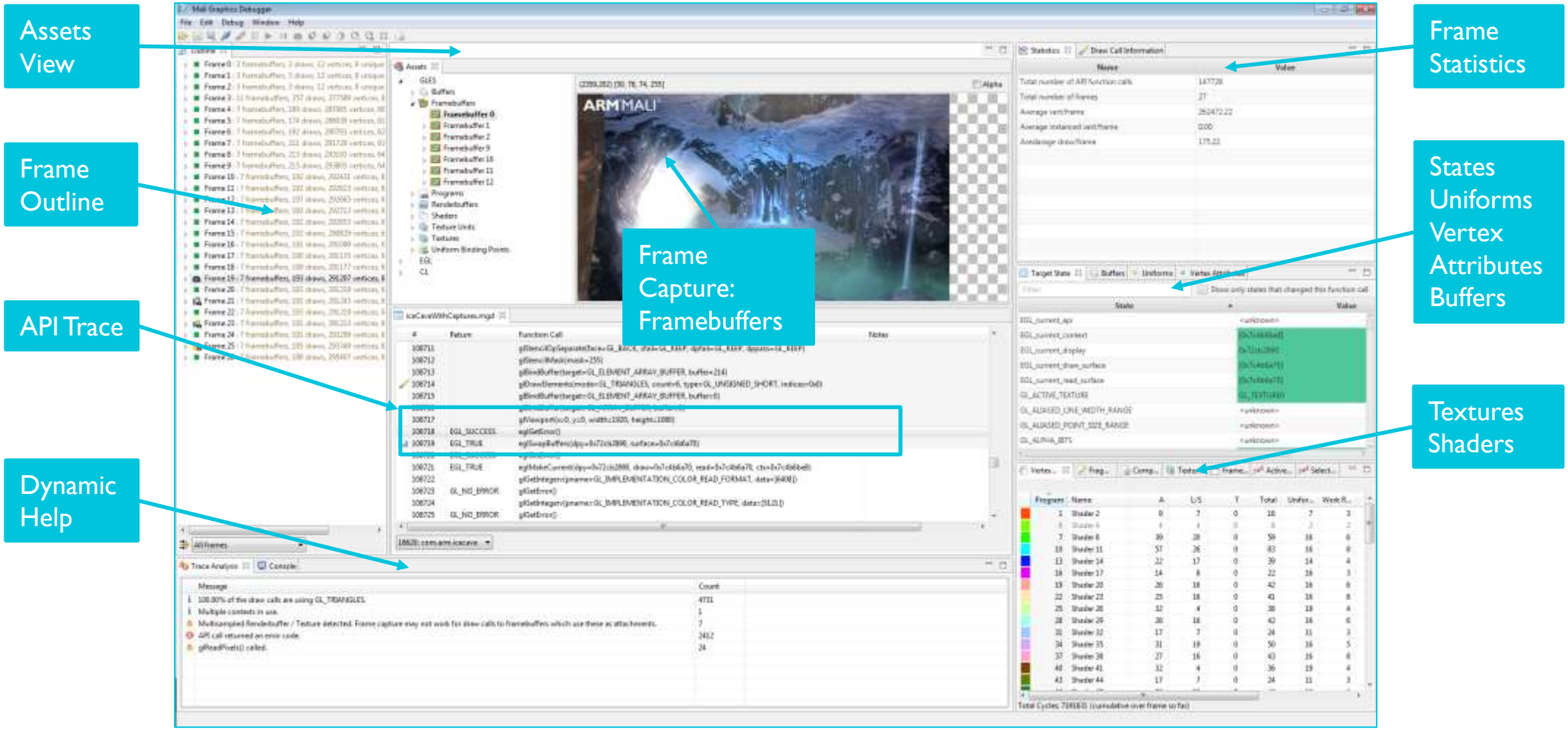
Debug
Mali
Graphics
Debugger



Optimize
Mali
Offline
Compiler

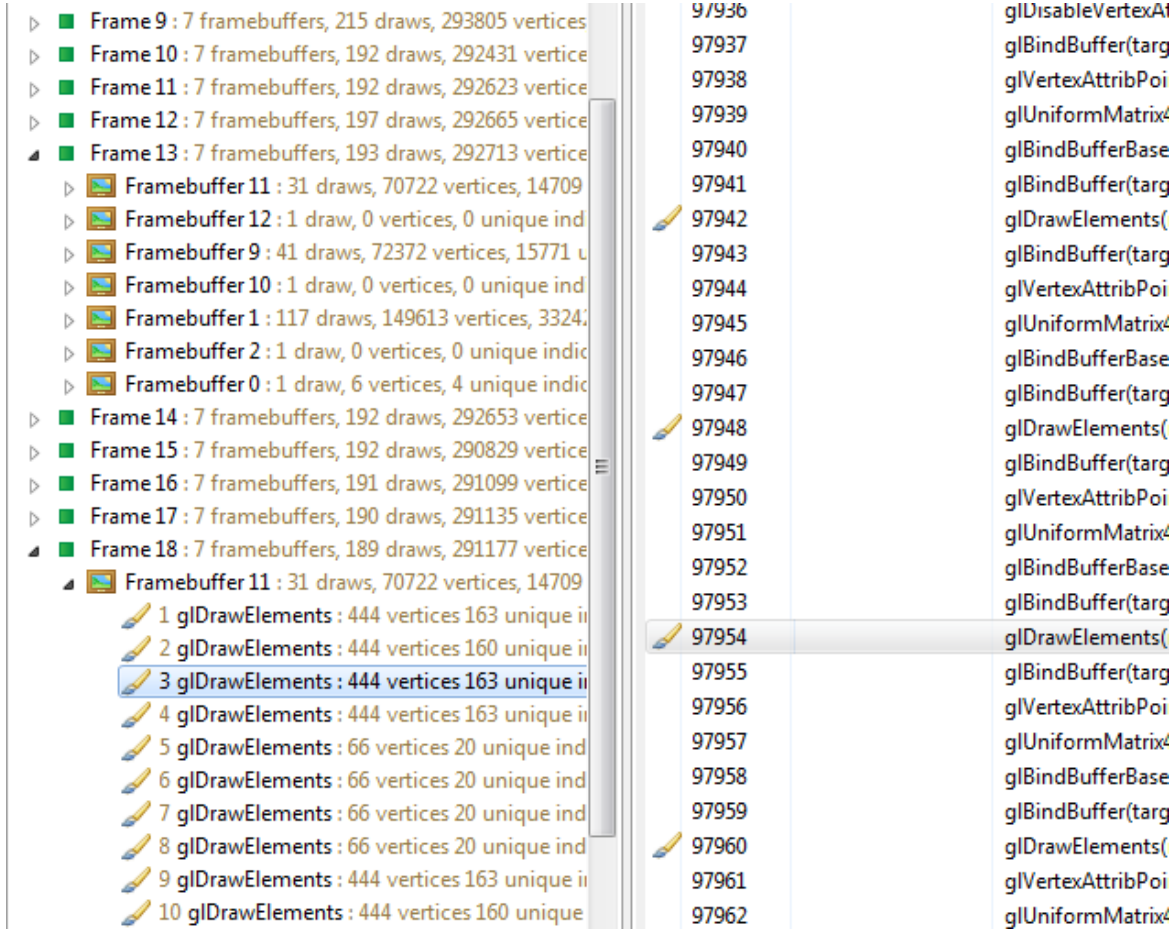
ARM

Investigation With the ARM Mali Graphics Debugger



Mali Graphics Debugger

Outline view

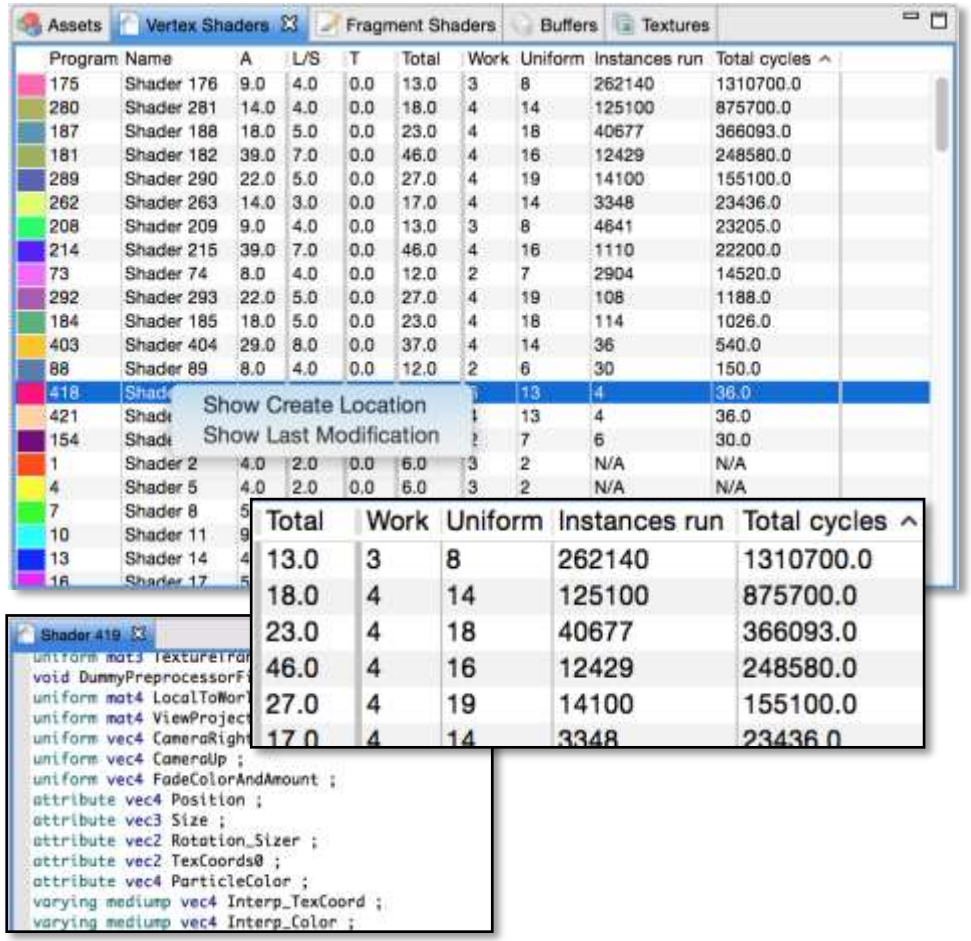


- Quick access to key graphics events
  - Frames
  - Render targets
  - Draw calls
- Investigate at frame level
  - Find which draw calls have higher geometry impact
- Jump between the outline view and the trace view seamlessly

ARM

Mali Graphics Debugger

Shaders reports and statistics



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:

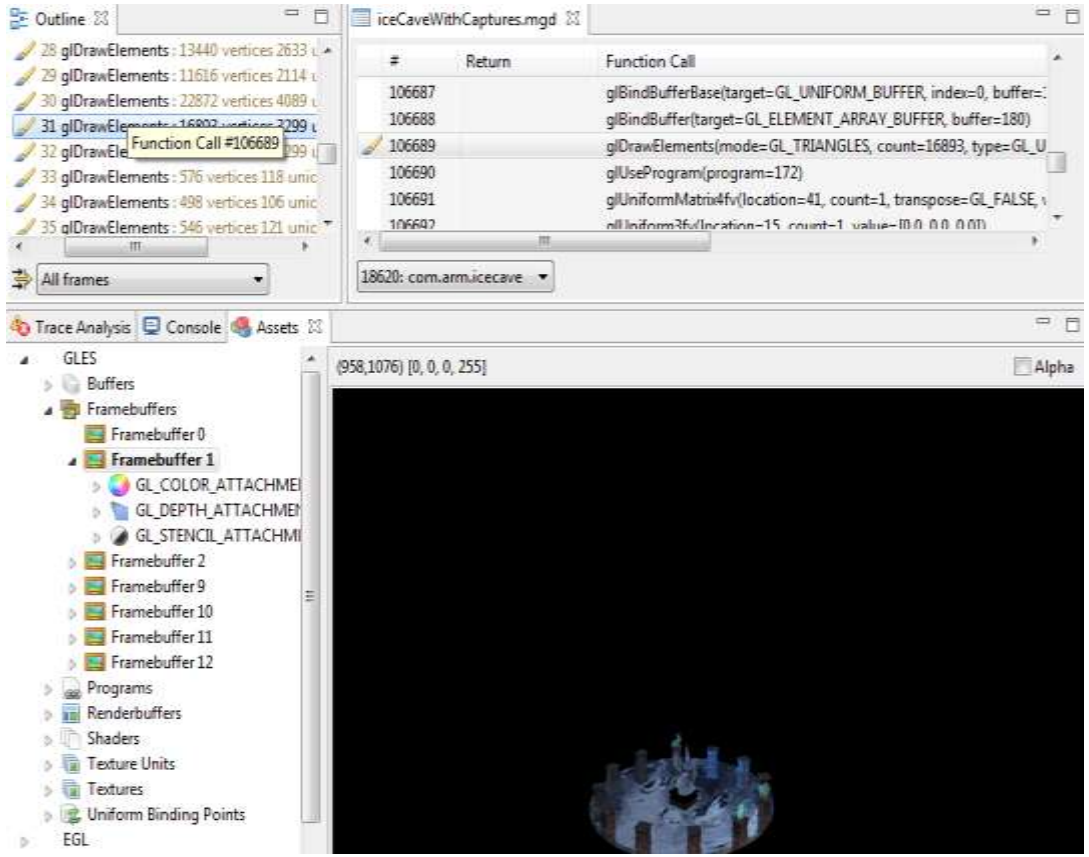
- number of instructions for each GPU pipeline
- number of work registers and uniform registers

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

ARM

Mali Graphics Debugger

Frame capture and analysis



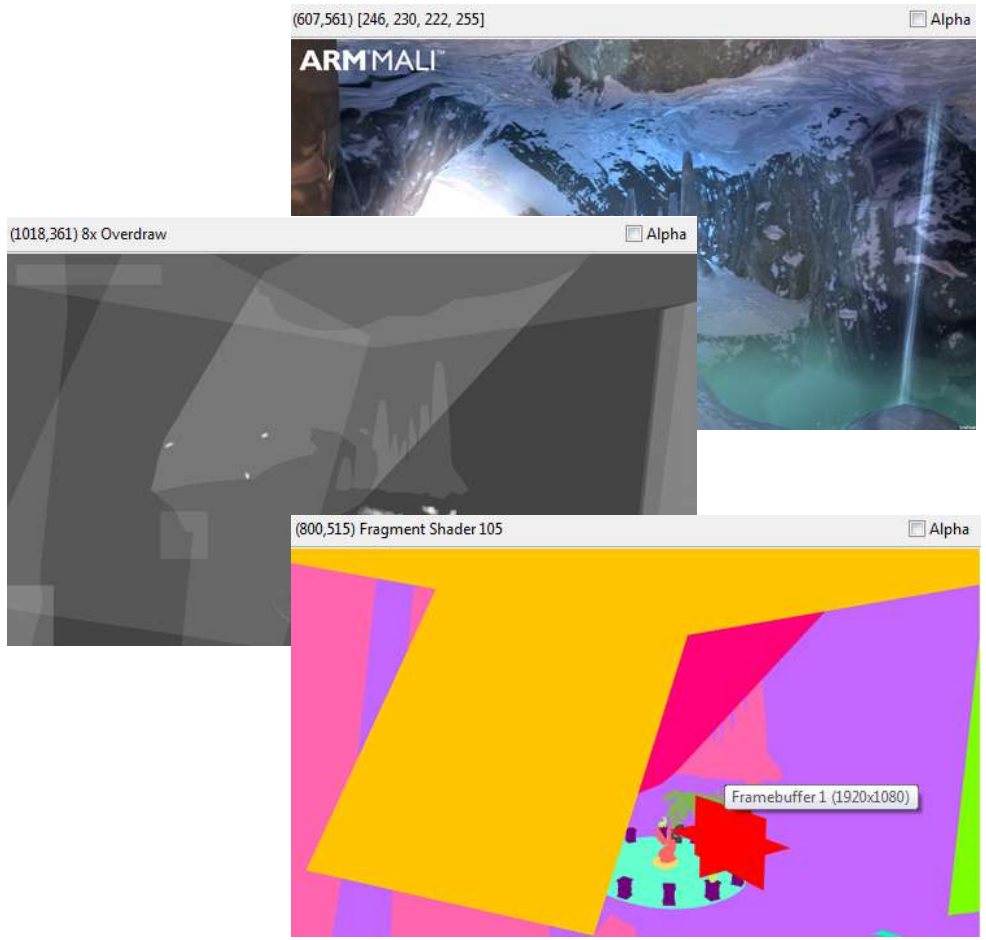
Frames can be fully captured to analyze the effect of each draw call

- 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

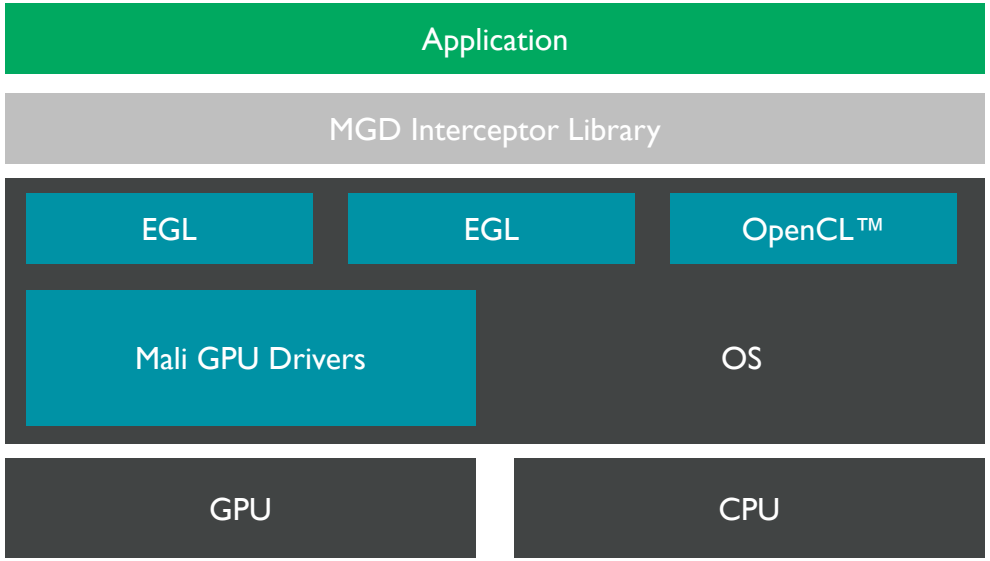
TCP/IP
Ethernet
USB

Target

Android
Linux

32-bit and 64-bit

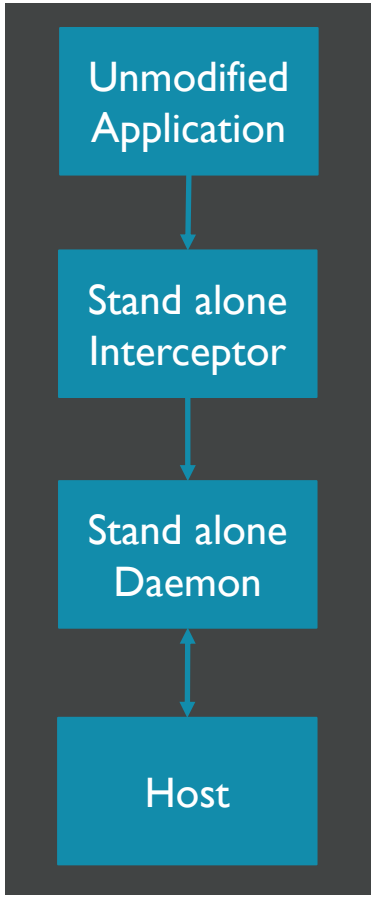
Under the hood of the
Mali Graphics Debugger



ARM

How to get MGD to Work

- On a rooted system



Original application doesn't need modifying in any way

In-fact all OpenGL ES applications pass through the interceptor so you may need to select the one you are interested in

Daemon is installed as a standalone binary and sends trace data to the host

In return the host sends commands back to the daemon

- On a non-rooted system

Application links against the interceptor which provides all the Open GL ES entry points

Must have access to source code to rebuild with the interceptor in

Daemon binary can't be placed in /system/bin as the system partition is mounted as read only

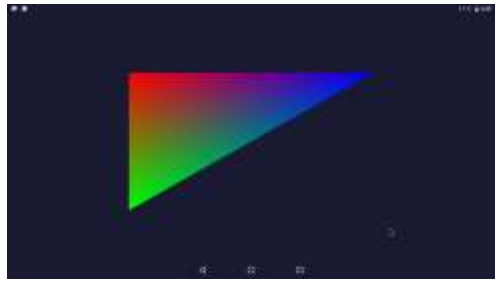
Instead use the MGD Daemon App which you can install and functions just like the binary

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

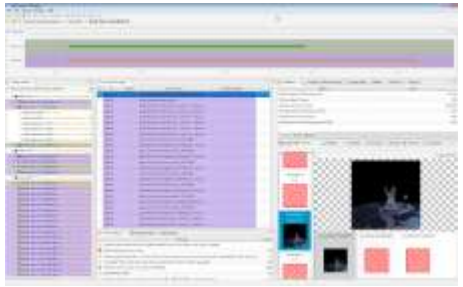
ARM

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



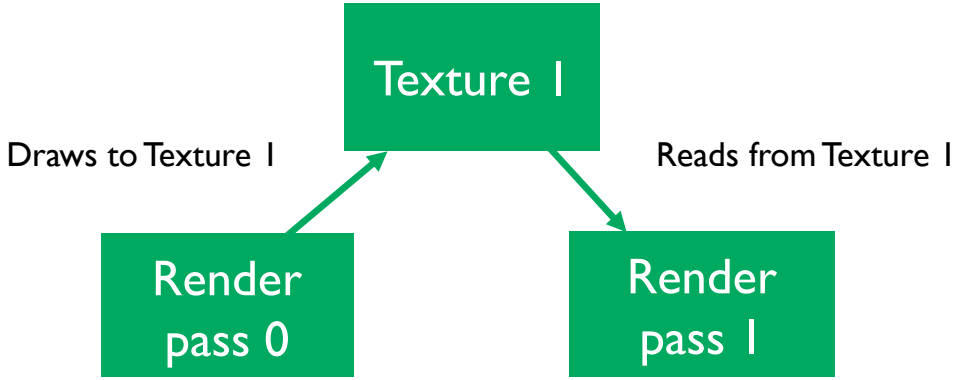
- VR Support
  - Frames are now registered through glFlush commands
  - All standard MGD features work
  - Frame Capture will now complete all render passes active when capture started



New Features in the Mali Graphics Debugger

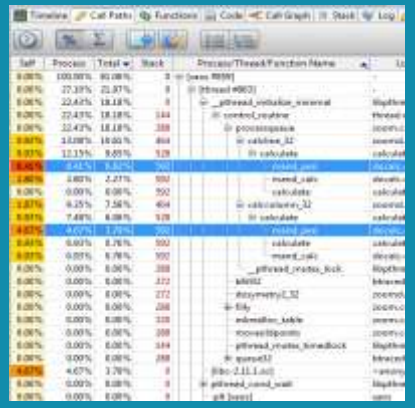
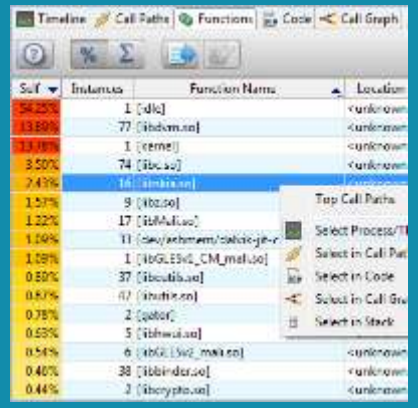
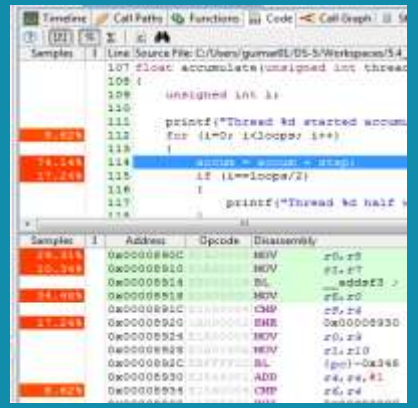
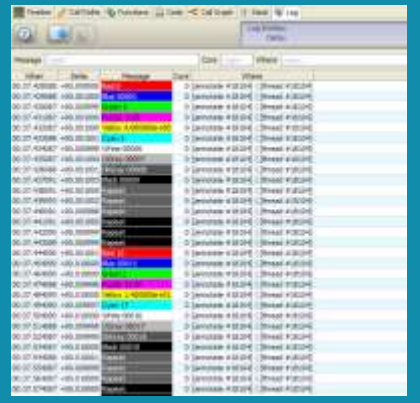
- Geometry Viewer
  - Visualize all of your geometry per draw call
  - Zero overhead to the application so you get this feature for free
  - Open all your old traces and they will now have this support

- 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

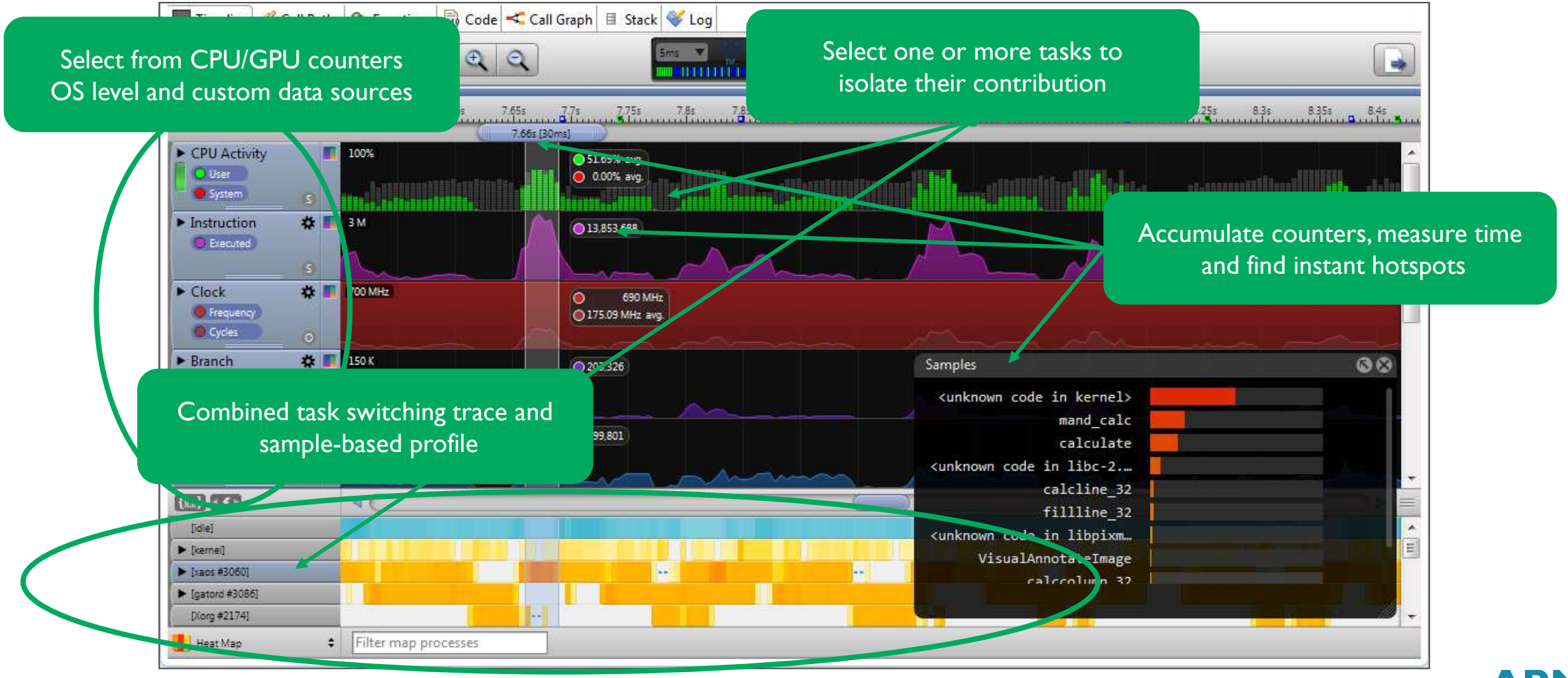
Source and instruction level profile. Color coded source code lines for easy identification

Log

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

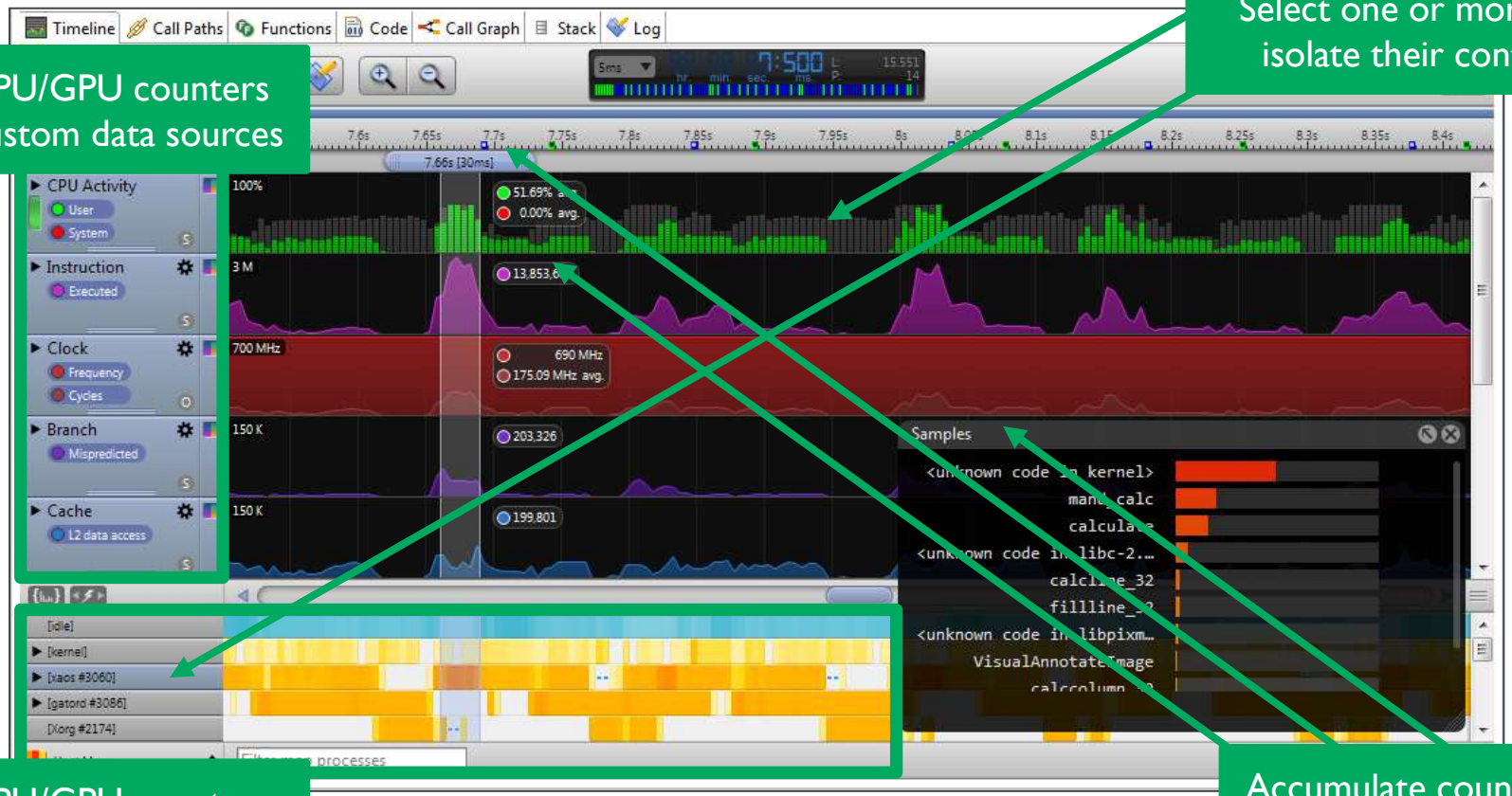
Timeline: Heat Map

Identify hotspots and system bottlenecks at a glance



Timeline: Heat Map

Identify hotspots and system bottlenecks at a glance



Select from CPU/GPU counters
OS level and custom data sources

Select one or more tasks to
isolate their contribution

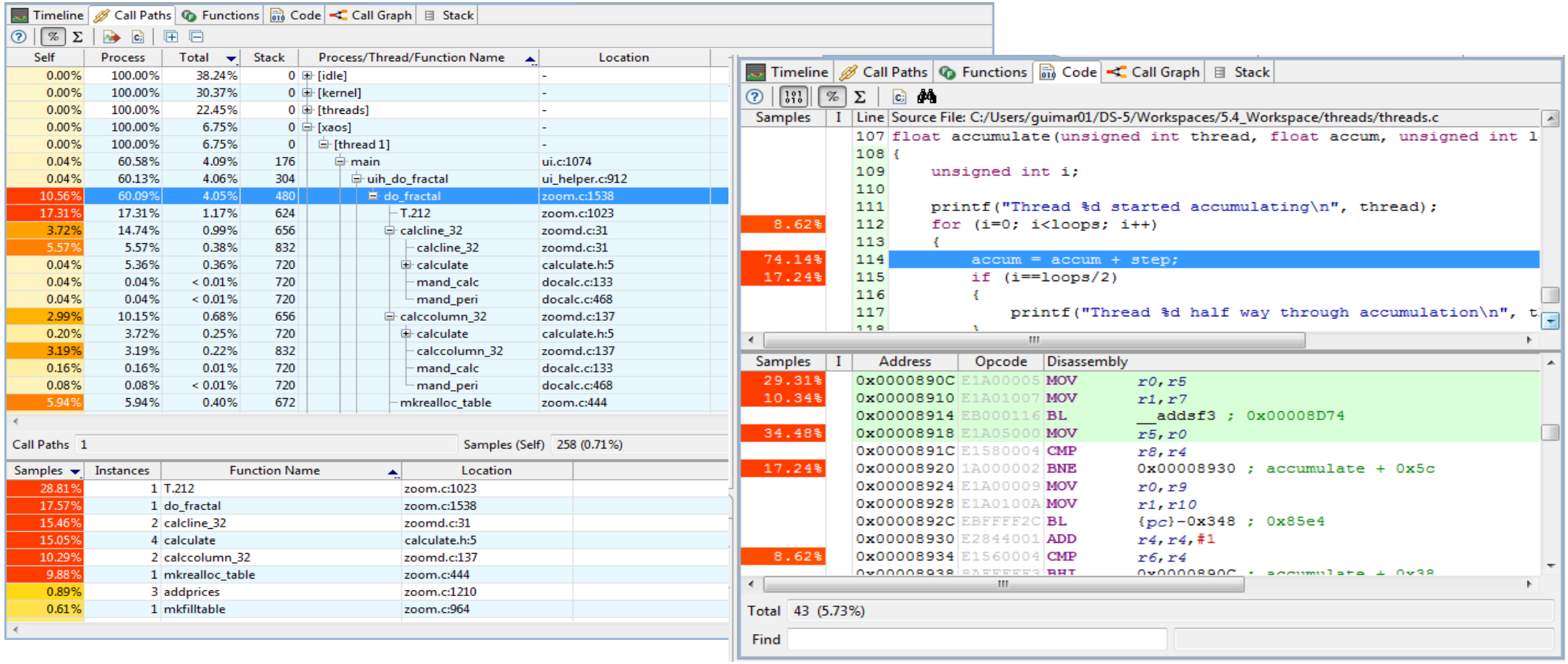
Select from CPU/GPU counters
OS level and custom data sources

Accumulate counters, measure
time and find instant hotspots

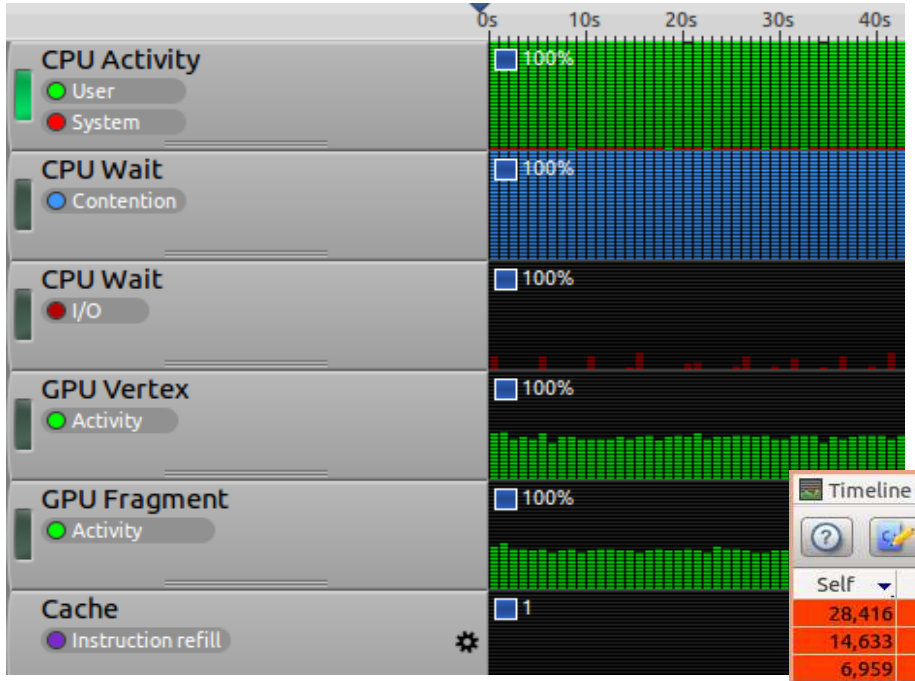
ARM

Profiling Reports

Analysis of call paths, source code and generated assembly



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>

ARM

Optimizing

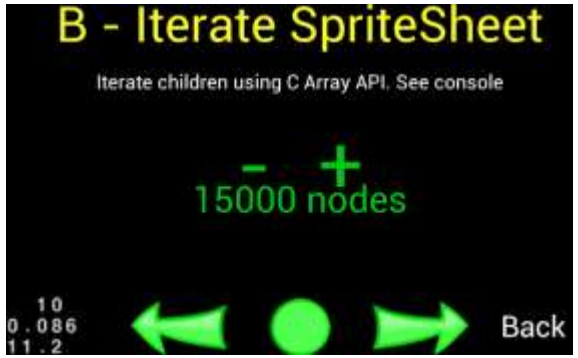
Optimization work led to reduced time in memcpy

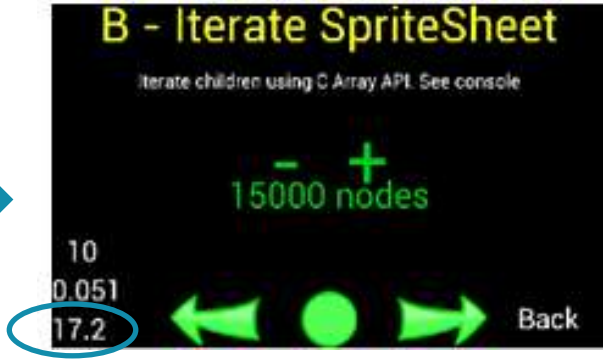
Self	% Self	Instances	Function Name
5,966	54.62%	6	memcpy
1,678	15.36%	1	cocos2d::CCSprite::updateTransform()
628	5.75%	1	IterateSpriteSheetCArray::update(float)
525	4.81%	1	cocos2d::CCSprite::setVisible(bool)
280	2.56%	1	cocos2d::CCSpriteBatchNode::draw()
164	1.50%	1	cocos2d::CCTextureAtlas::updateQuad(co
87	0.80%	15	[gator]
57	0.52%	1	sc8810_idle
51	0.47%	1	cocos2d::CCNode::updateTransform()
50	0.46%	12	__schedule
41	0.38%	1	scanObject(const Object*, GcMarkContex
32	0.29%	1	__delay

Self	% Self	Instances	Function Name
3,807	35.01%	3	cocos2d::CCSprite::updateTransform()
1,030	9.47%	1	cocos2d::CCNode::updateTransform()
994	9.14%	1	IterateSpriteSheetCArray::update(float)
990	9.10%	1	memcpy
899	8.27%	2	cocos2d::CCSprite::isQuadColorsDirty()
795	7.31%	1	cocos2d::CCSprite::setVisible(bool)
161	1.48%	1	cocos2d::CCSpriteBatchNode::draw()
161	1.48%	1	cocos2d::CCTextureAtlas::updateQuad(co
121	1.11%	17	[gator]
75	0.69%	15	__schedule

Real world optimization of 6FPS from memcpy alone

FPS



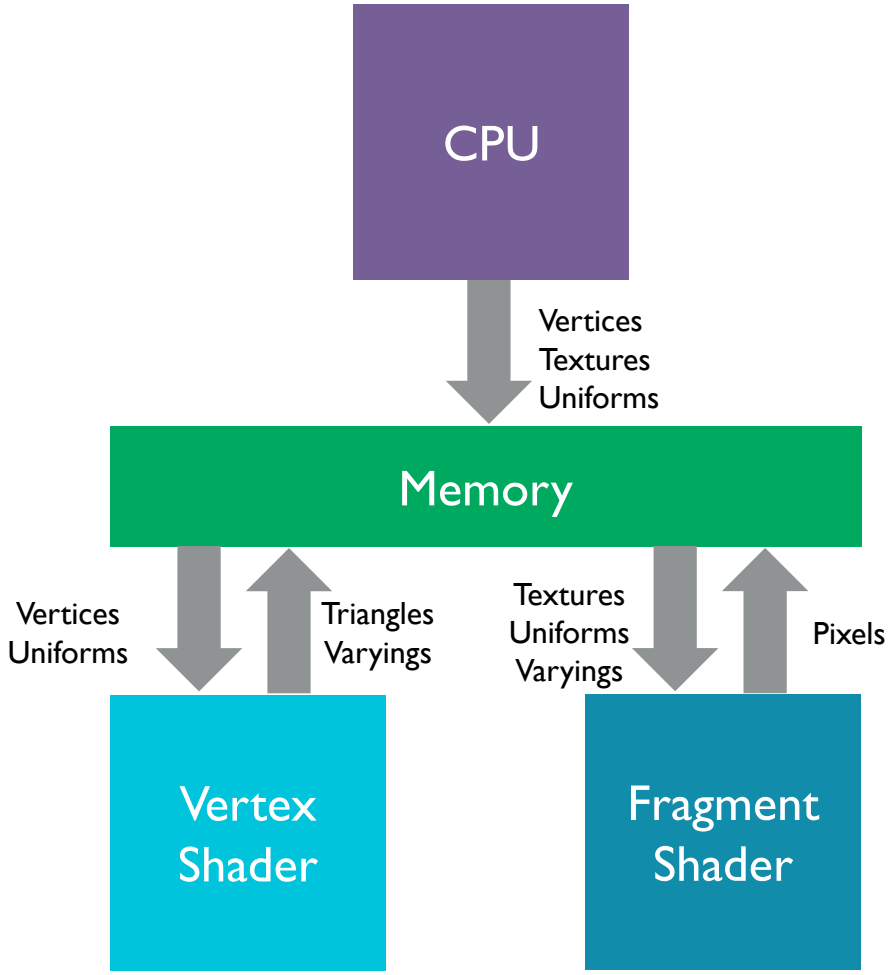


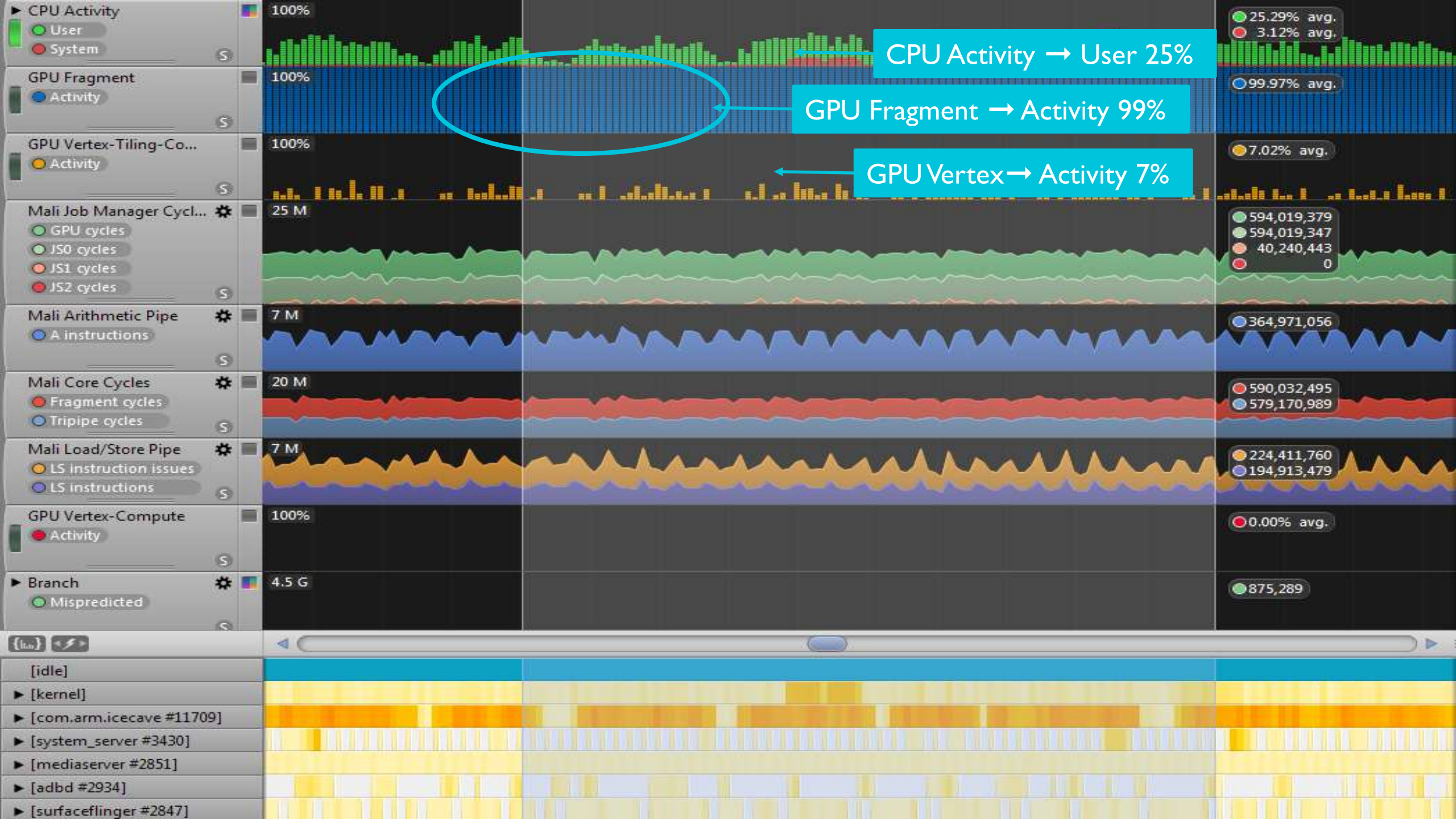
52% faster

ARM

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




CPU Activity
User
System
100%
25.29% avg.
3.12% avg.
CPU Activity → User 25%
GPU Fragment
Activity
100%
99.97% avg.
GPU Fragment → Activity 99%
GPU Vertex-Tiling-Co...
Activity
100%
7.02% avg.
GPU Vertex → Activity 7%
Mali Job Manager Cycl...
GPU cycles
JS0 cycles
JS1 cycles
JS2 cycles
25 M
594,019,379
594,019,347
40,240,443
0
Mali Arithmetic Pipe
A instructions
7 M
364,971,056
Mali Core Cycles
Fragment cycles
Tripipe cycles
20 M
590,032,495
579,170,989
Mali Load/Store Pipe
LS instruction issues
LS instructions
7 M
224,411,760
194,913,479
GPU Vertex-Compute
Activity
100%
0.00% avg.
Branch
Mispredicted
4.5 G
875,289
[idle]
[kernel]
[com.arm.icecave #11709]
[system_server #3430]
[mediaserver #2851]
[adbd #2934]
[surfaceflinger #2847]

Vertex Count and Shader Optimizations

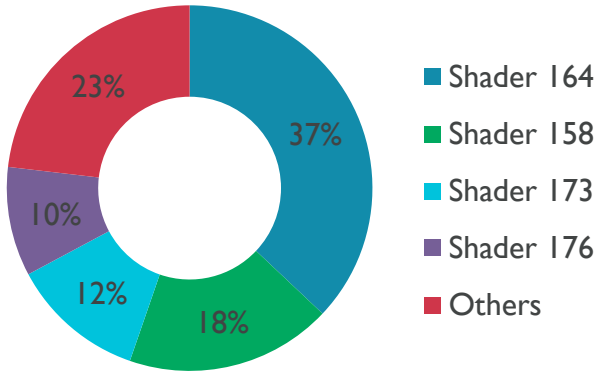
Identify the top heavyweight vertex shaders

iceCaveWithCaptures.mgd | Shader 164

```
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	Shader 164	24	16	0	40	95,856	2,683,968	37.0%
157	Shader 158	32	25	0	57	32,256	1,322,496	18.2%
172	Shader 173	44	32	0	76	16,893	861,543	11.9%
175	Shader 176	47	28	0	75	14,010	700,500	9.6%
171	Shader 169	9	10	0	19	35,739	536,085	7.4%
97	Shader 98	8	3	0	11	70,722	495,054	6.8%
154	Shader 155	25	20	0	45	12,240	403,920	5.6%
195	Shader 193	9	13	0	22	3,360	60,480	0.8%
130	Shader 131	10	6	0	16	4,836	58,032	0.8%
160	Shader 161	25	20	0	45	984	32,472	0.4%
166	Shader 167	38	31	0	69	432	21,168	0.3%
85	Shader 86	11	6	0	17	1,752	21,024	0.3%
187	Shader 188	9	7	0	16	1,176	14,112	0.2%
181	Shader 182	38	20	0	58	324	13,608	0.2%
109	Shader 110	24	7	0	31	600	10,800	0.1%
82	Shader 83	9	7	0	16	858	10,296	0.1%

Vertex Cycles Per Program



ARM

Inspect the Tripipe Counters

Reduce the load on the Arithmetic pipeline

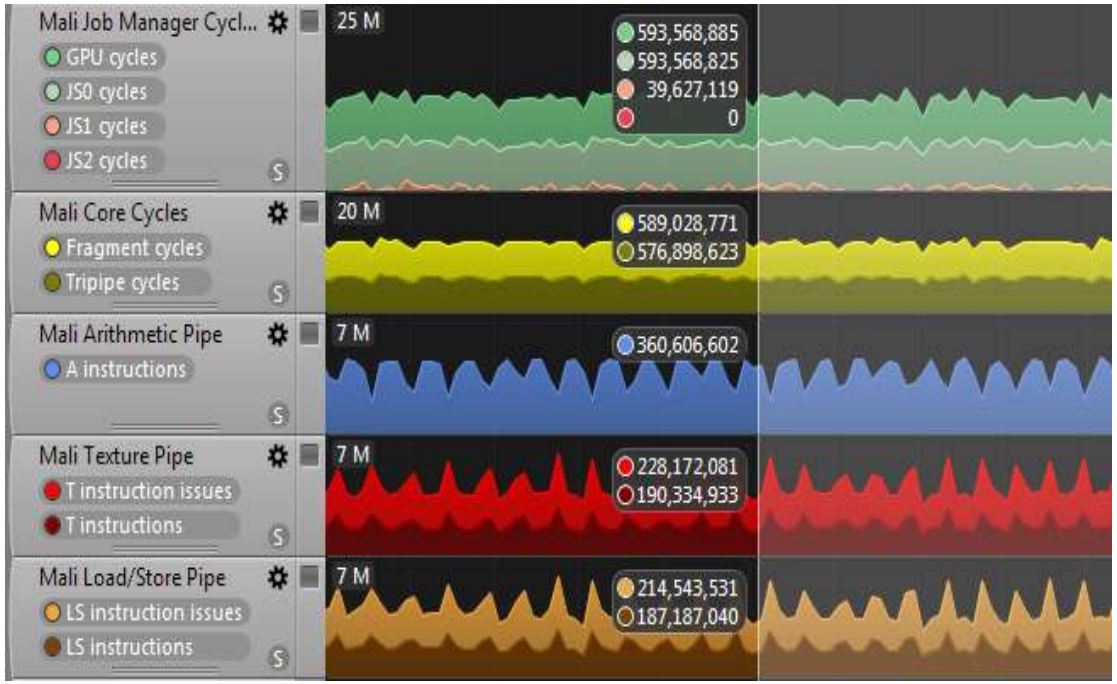
GPU Cycles 593m

Tripipe Cycles 576m

Arithmetic 361m

Texture 228m

Load & Store 214m



ARM

Tripipe Counters

Cycles per instruction metrics

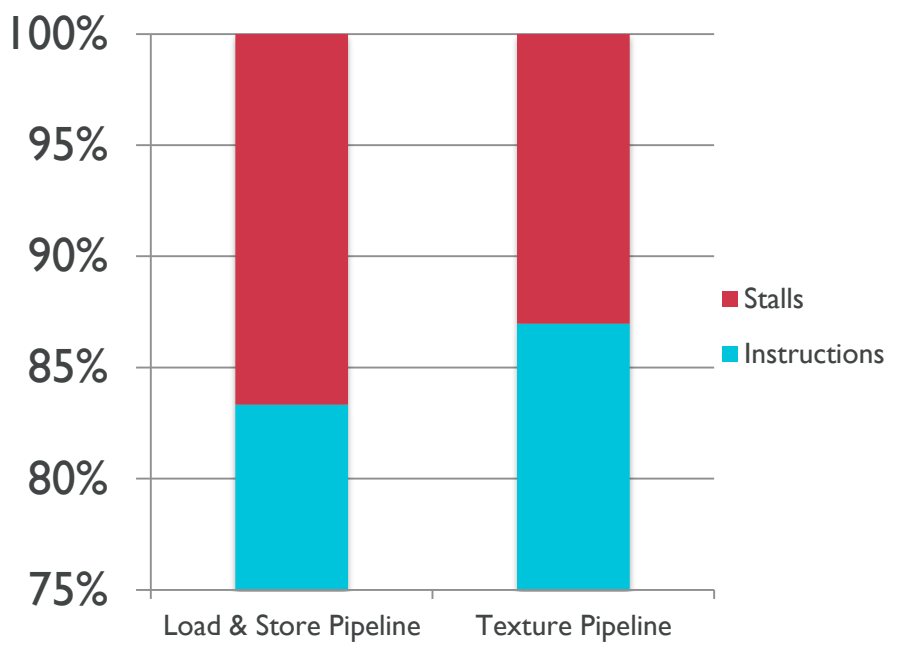
- It's easy to calculate a couple of CPI (cycles per instruction) metrics:

- For the load/store pipeline we have:

  228m (*Mali Load/Store Pipe → LS instruction issues*)
  / 190m (*Mali Load/Store Pipe → LS instructions*)
  = 1.2 cycles/instruction

- For the texture pipeline we have:

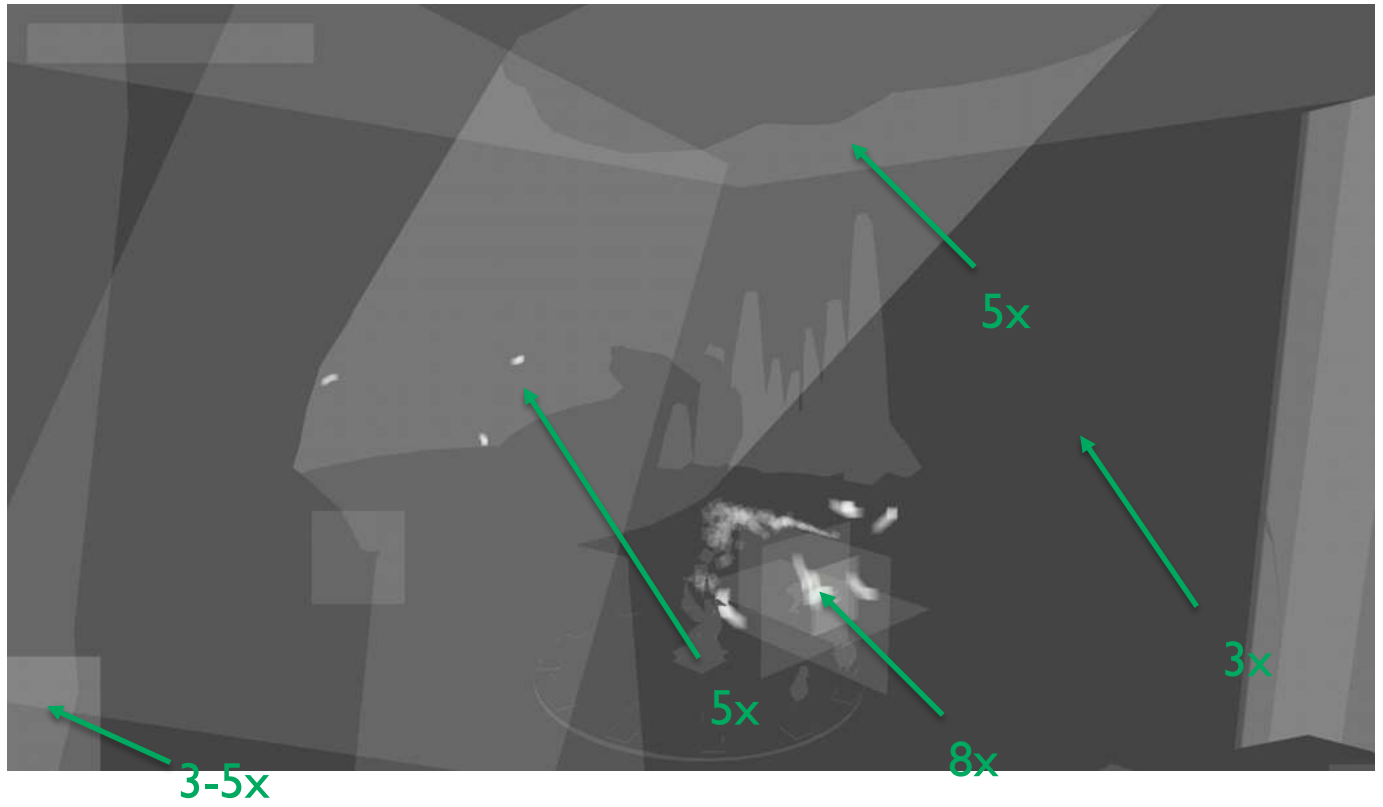
  215m (*Mali Texture Pipe → T instruction issues*)
  / 187m (*Mali Texture Pipe → T instructions*)
  = 1.15 cycles/instruction



ARM

Frame Analysis

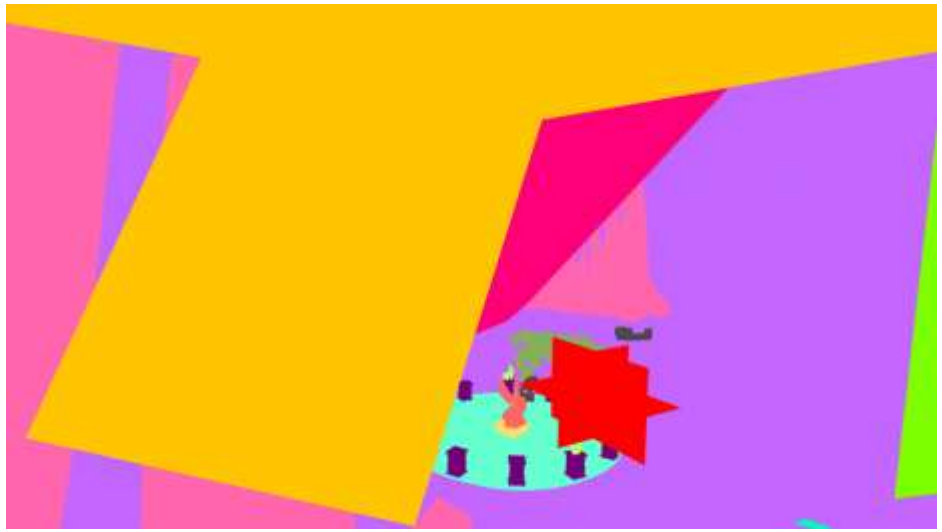
Check the overdraw factor



ARM

Shader Map and Fragment Count

Identify the top heavyweight fragment shaders

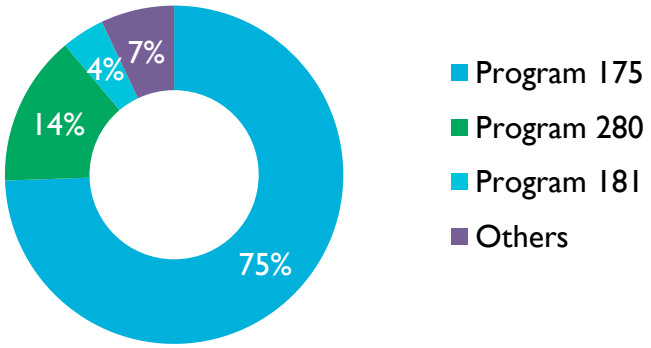


Assets | Vertex Shaders | Fragment Shaders | Textures

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

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

Fragment Count Per Program



ARM

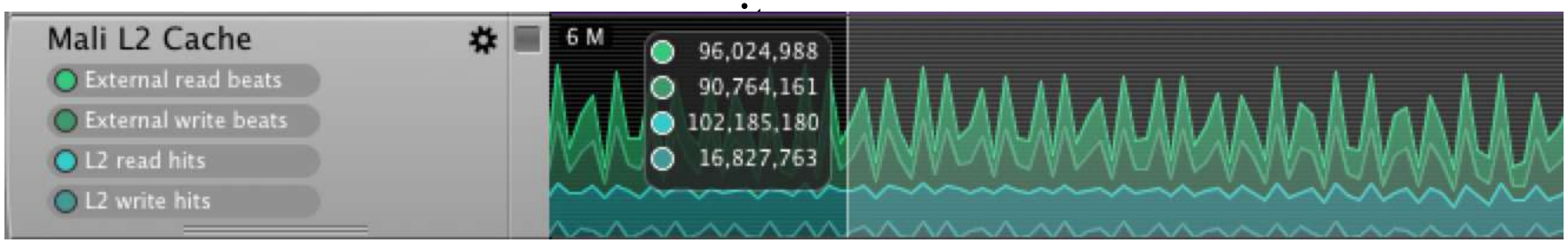
Bandwidth

- 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:

(96m (*Mali L2 Cache → External read beats*) +
90.7m (*Mali L2 Cache → External write beats*)) x 16
~= 2.9 GB/s

- Since bandwidth usage is related to energy consumption it's always worth optimizing



ARM

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??

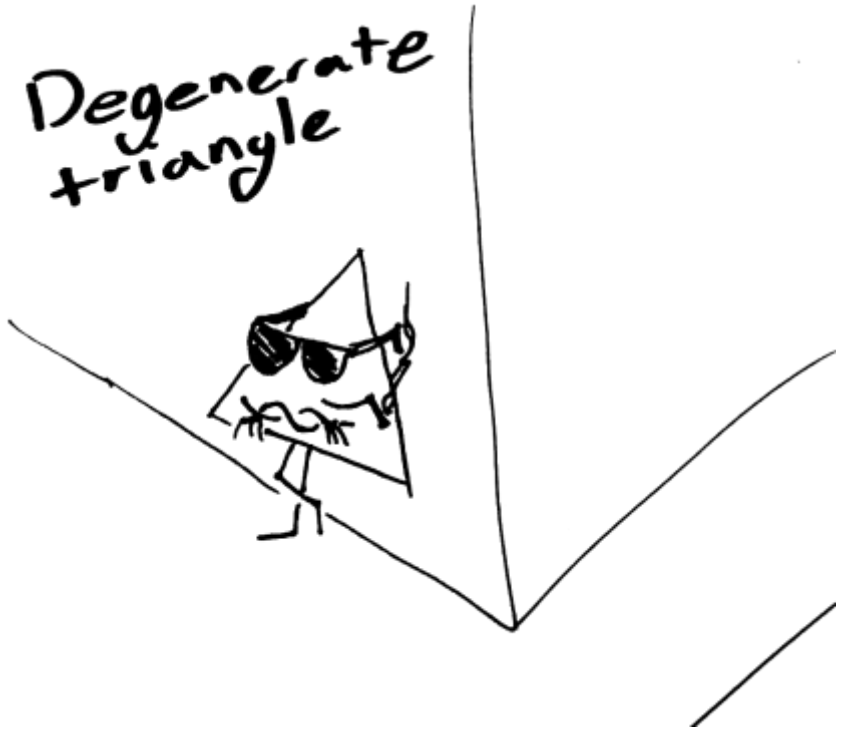
The Sensible Six Best Practices

- 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 thing 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

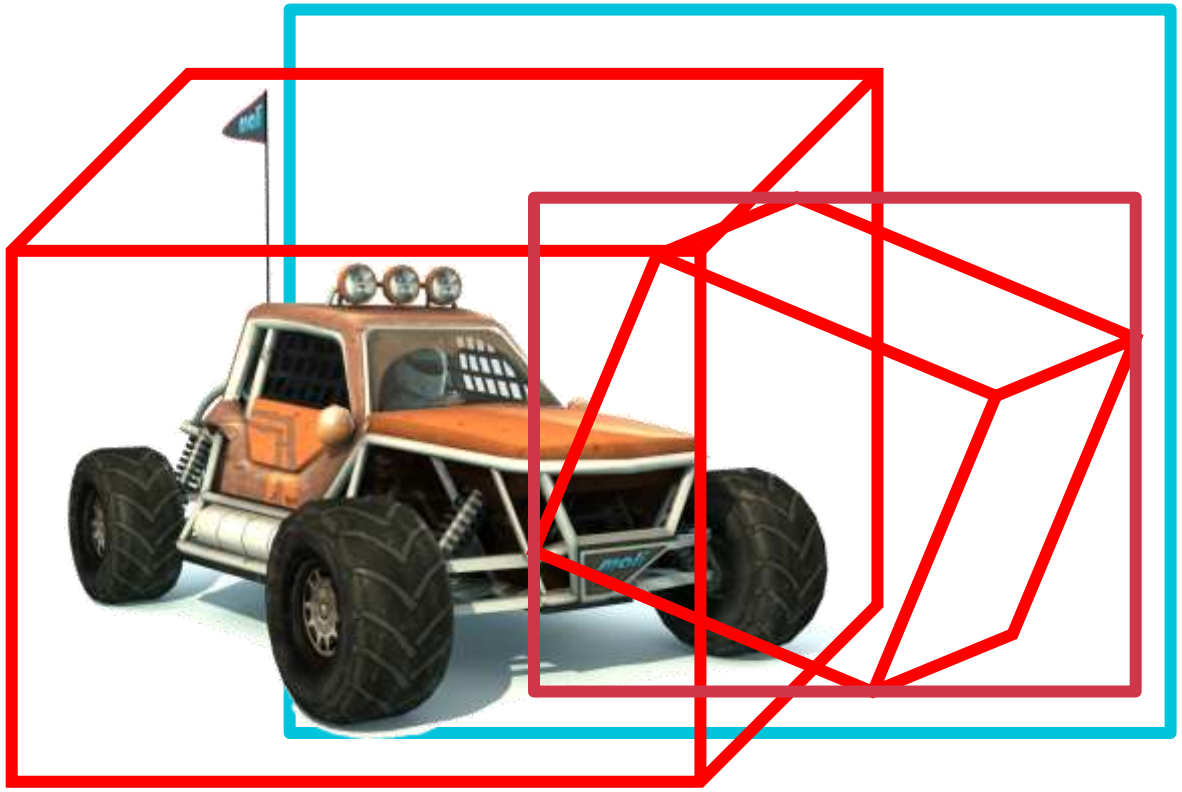
Culling can help with vertex and sometimes CPU load

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

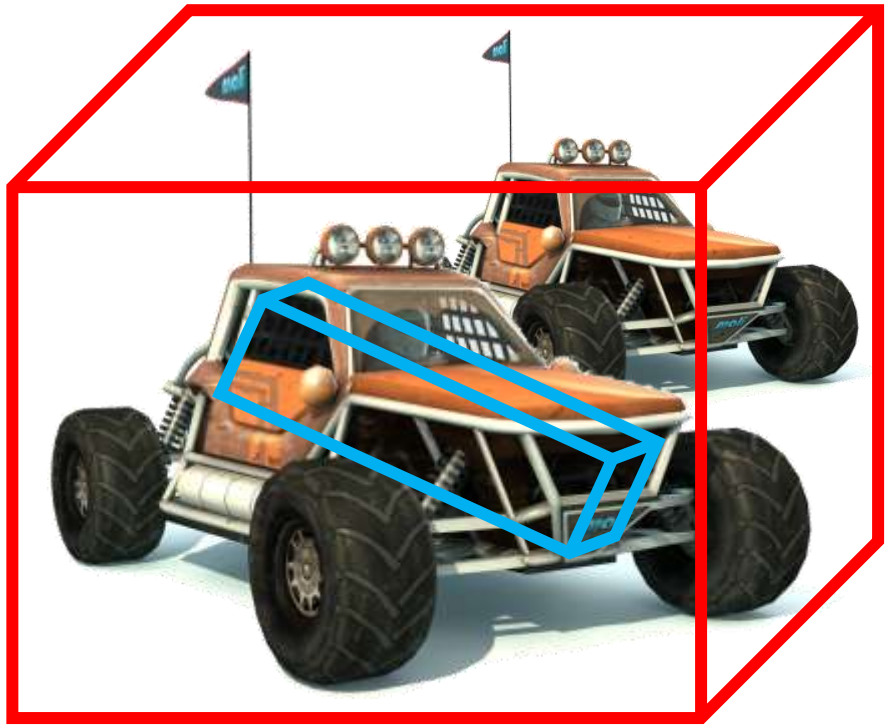
Occlusion Culling

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

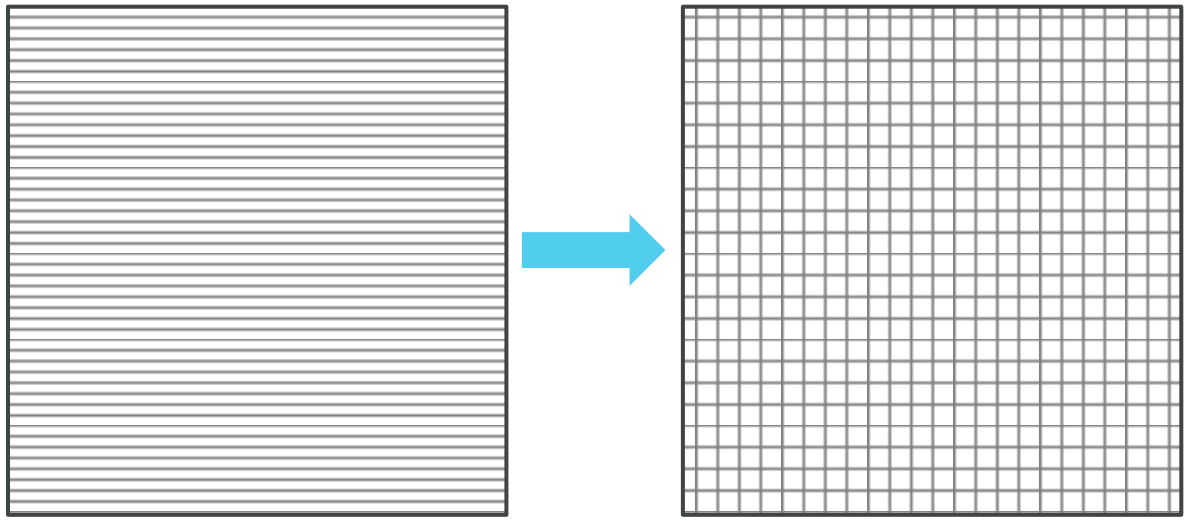
Texture Compression

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?

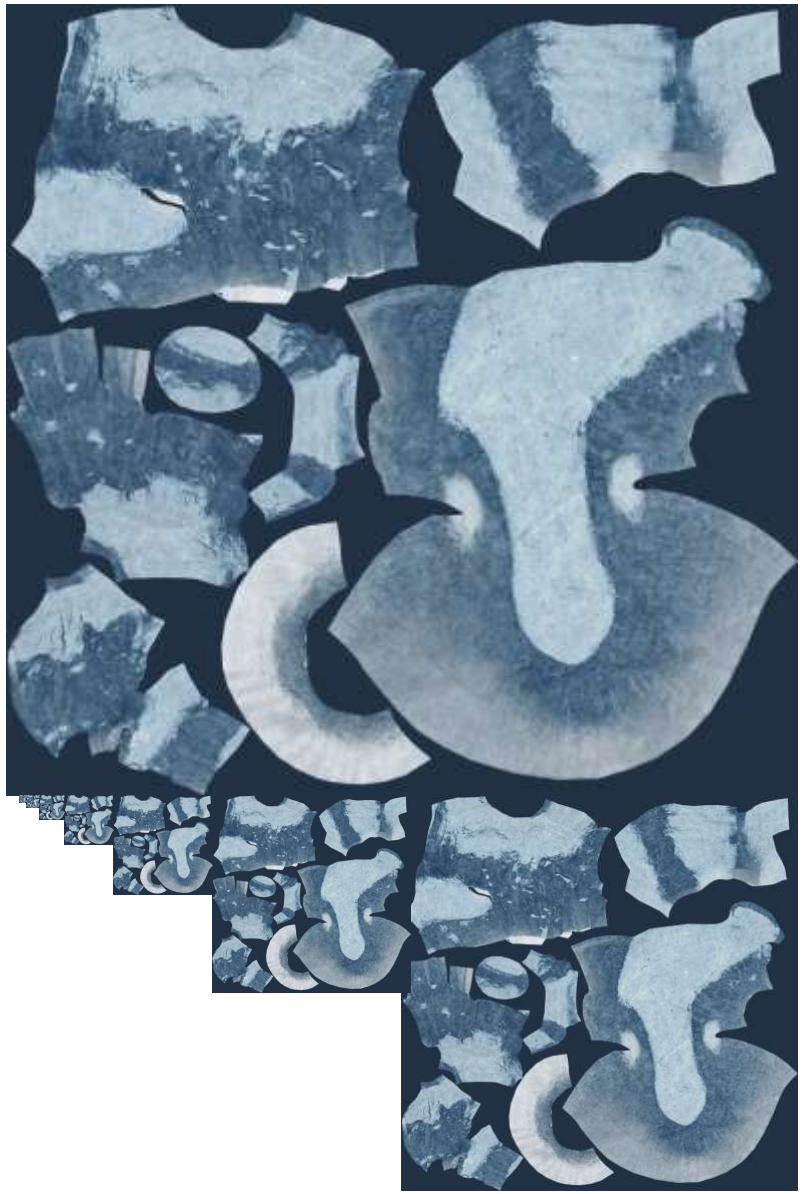
- Block based
- Deterministic
- Immediate
- Random lookup

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~~

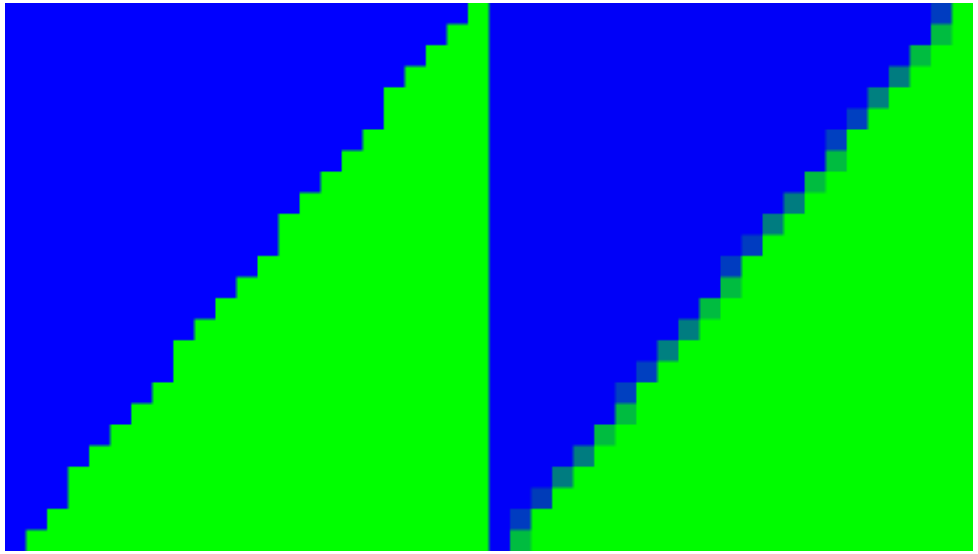
But you'll need to compress mipmaps too



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

ARM

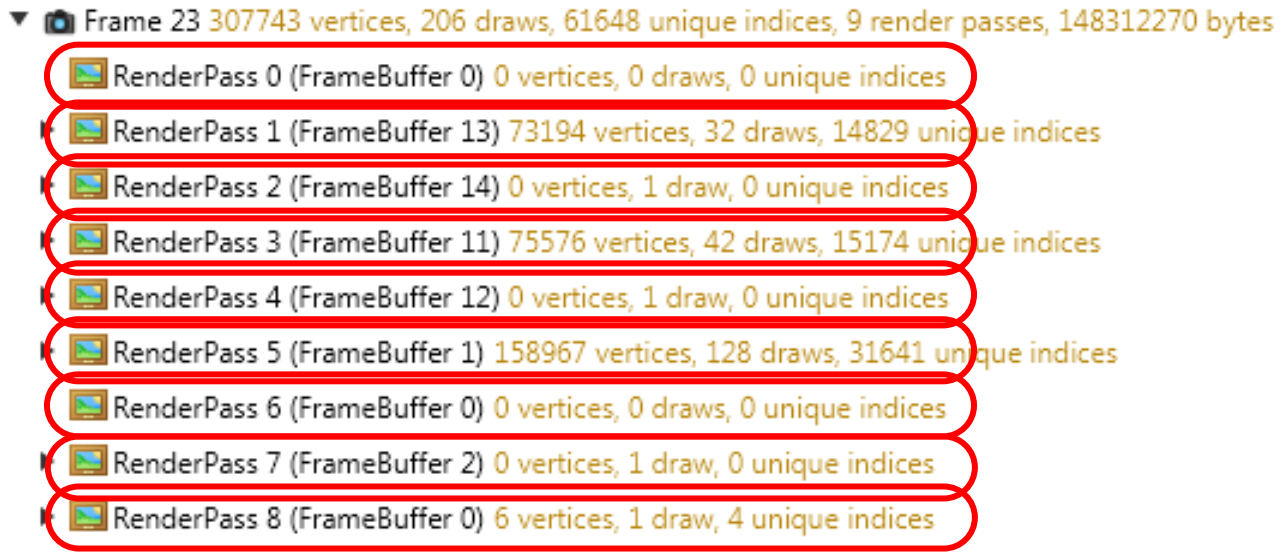
Ice Cave's Passes

- 2 'passes' are nothing
- Pass 2, 4 and 7 are buffer blits
- Pass 8 is composition

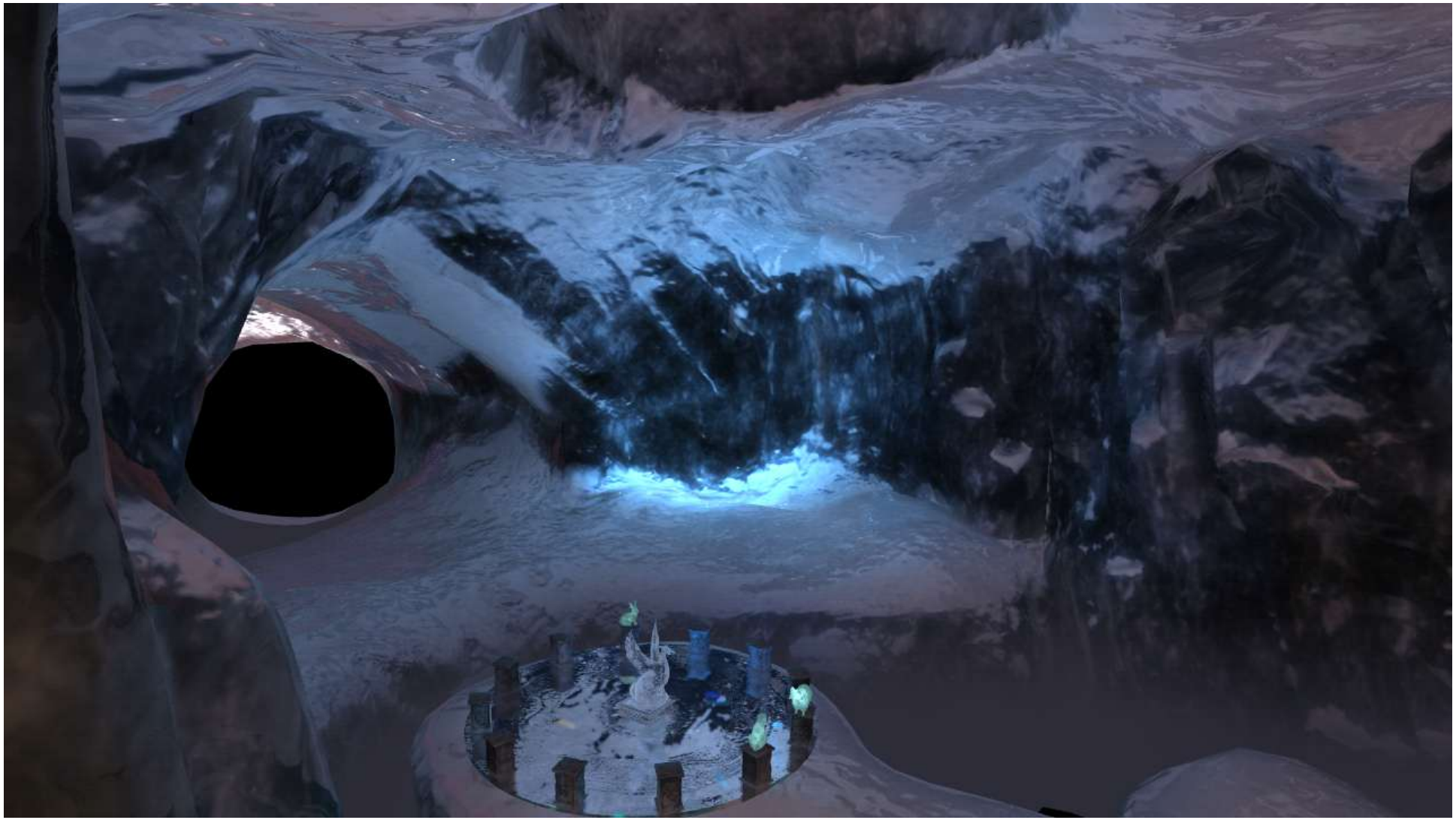
- Pass 1 is shadows

- Pass 3 is reflections

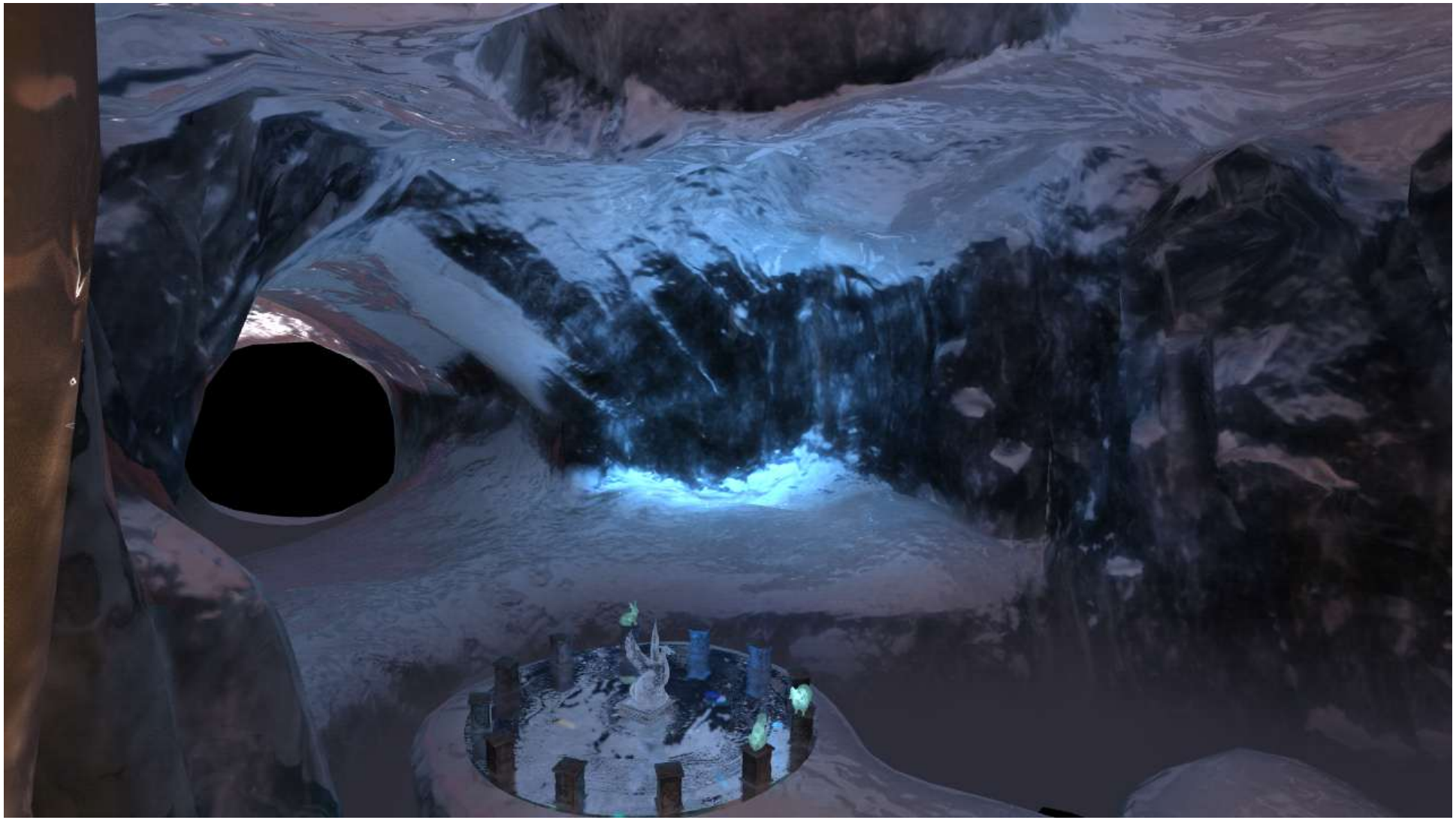
- Pass 5 is the main event



Ice Cave's Layers (Overdraw)

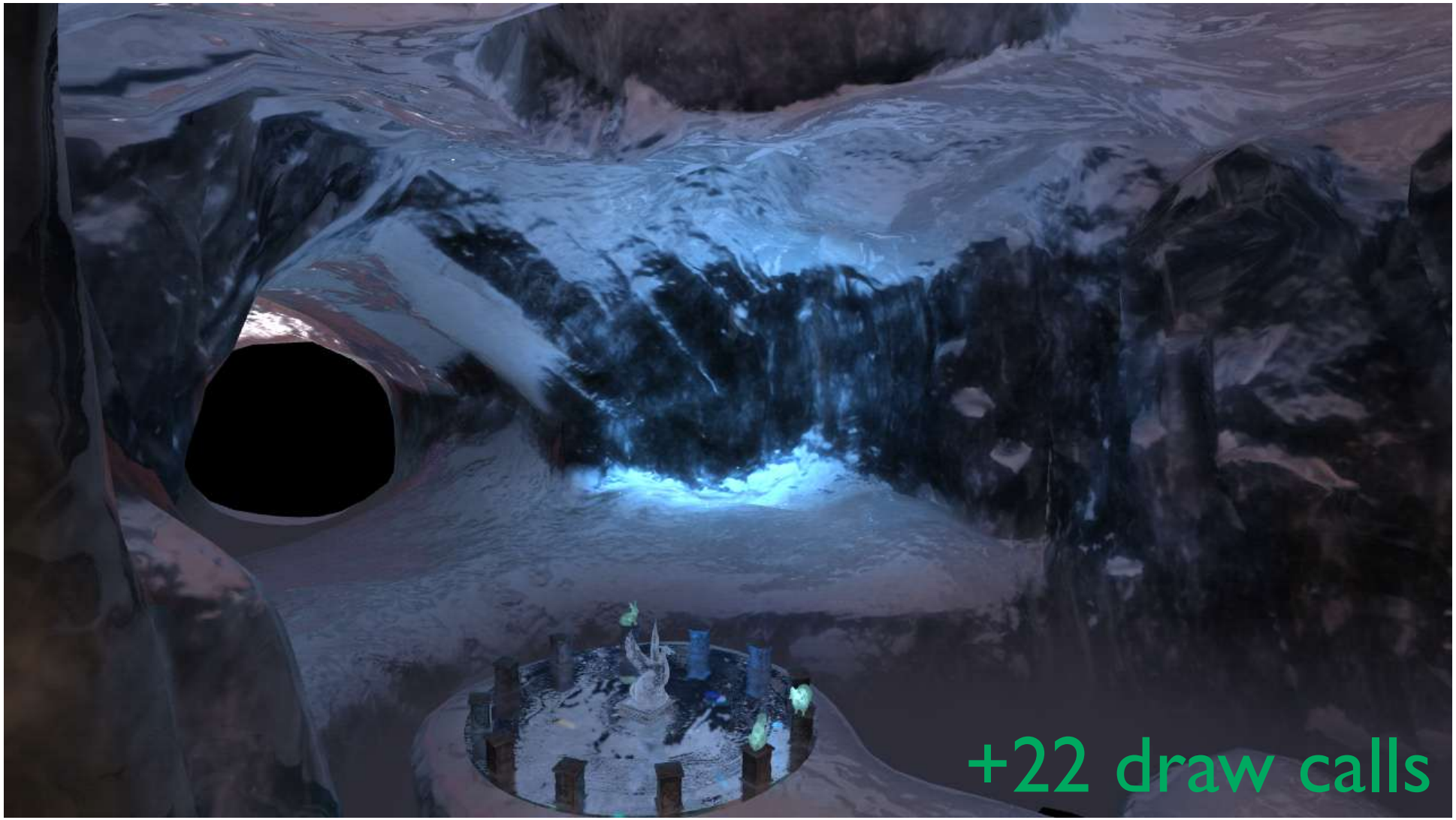


Ice Cave's Layers (Batching)



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:

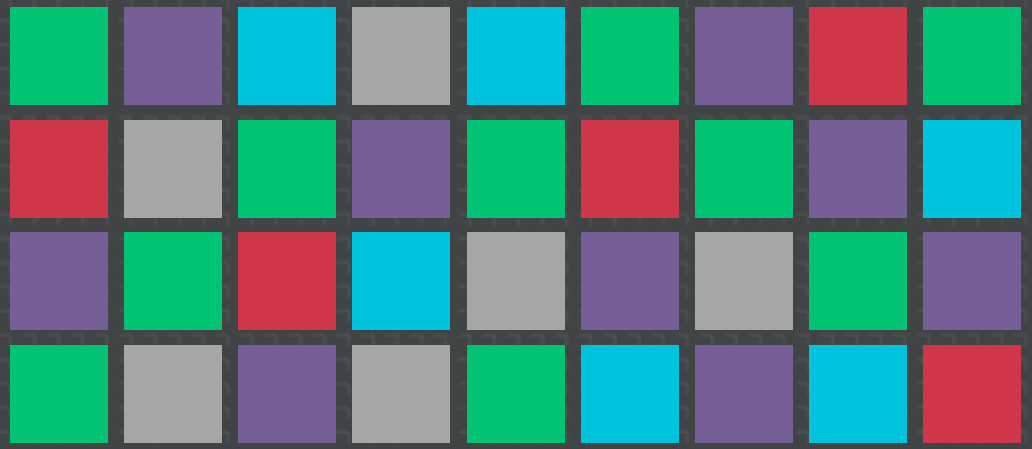
- Batching
- Overdraw
- Culling
- LoD
- Compression
- Antialiasing

Thank you!

ARM

More slides:
http://malideveloper.arm.com/gdc2016

To Find Out More….

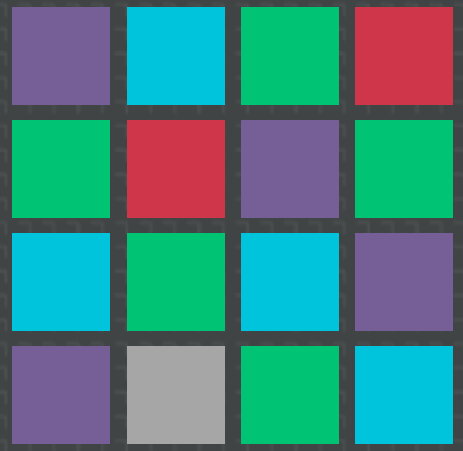


ARM Booth #1624 on Expo Floor:

- 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:

- 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/

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