Performance Optimization and Debug Tools for mobile games with PlayCanvas

Jonathan Kirkham, *Senior Software Engineer, ARM*
Will Eastcott, CEO, PlayCanvas

Introduction

Jonathan Kirkham, ARM

- Worked with ARM technology and graphics at University
- Joined ARM in 2011 to work on 3D graphics
- Developing performance analysis tools and debuggers for the Mali GPUs

Agenda

1. Introduction to WebGL™ on mobile
   - Rendering Pipeline
2. PlayCanvas experience
   - WebGL Inspector
3. Performance analysis and debugging tools for WebGL
   - Generic optimization tips
4. Q & A

Bring the Power of OpenGL® ES to Mobile Browsers

What is WebGL™?

- A cross-platform, royalty free web standard
- Low-level 3D graphics API
- Based on OpenGL® ES 2.0
- A shader based API using GLSL (OpenGL Shading Language)
- Some concessions made to JavaScript™ (memory management)

Why WebGL?

- It brings plug-in free 3D to the web, implemented right into the browser.
- Major browser vendors are members of the WebGL Working Group:
  - Apple (Safari® browser)
  - Google (Chrome™ browser)
  - Mozilla (Firefox® browser)
  - Opera (Opera™ browser)

Introduction to WebGL™

- How does it fit in a web browser?
  - You use JavaScript™ to control it.
  - Your JavaScript is embedded in HTML5 and uses its Canvas element to draw on.

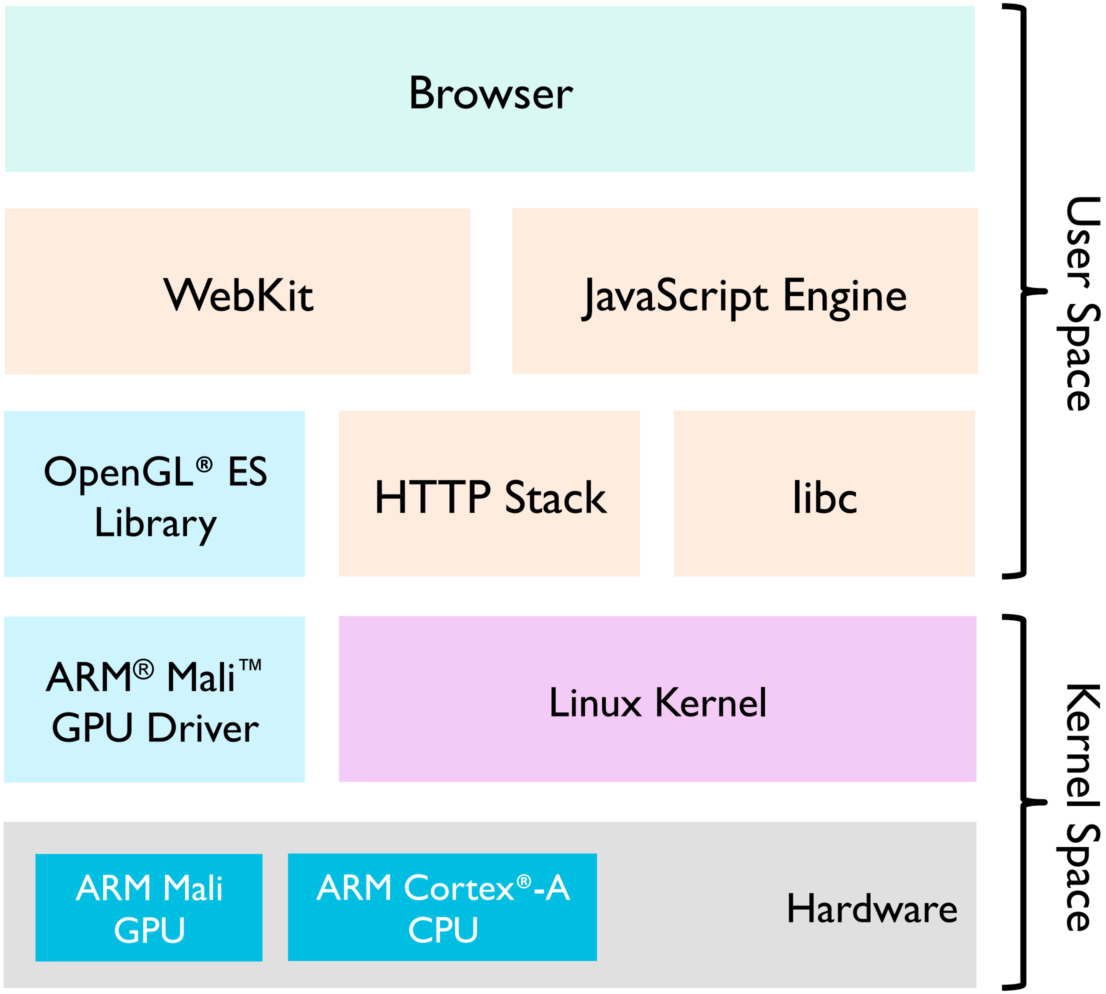
- What do you need to start creating graphics?
  - Obtain WebGLrenderingContext object for a given HTMLCanvasElement.
  - It creates a drawing buffer into which the API calls are rendered.
  - For example:

```
var canvas = document.getElementById('canvas1');
var gl = canvas.getContext('webgl');
canvas.width = newWidth;
canvas.height = newHeight;
gl.viewport(0, 0, canvas.width, canvas.height);
```

WebGL™ Stack

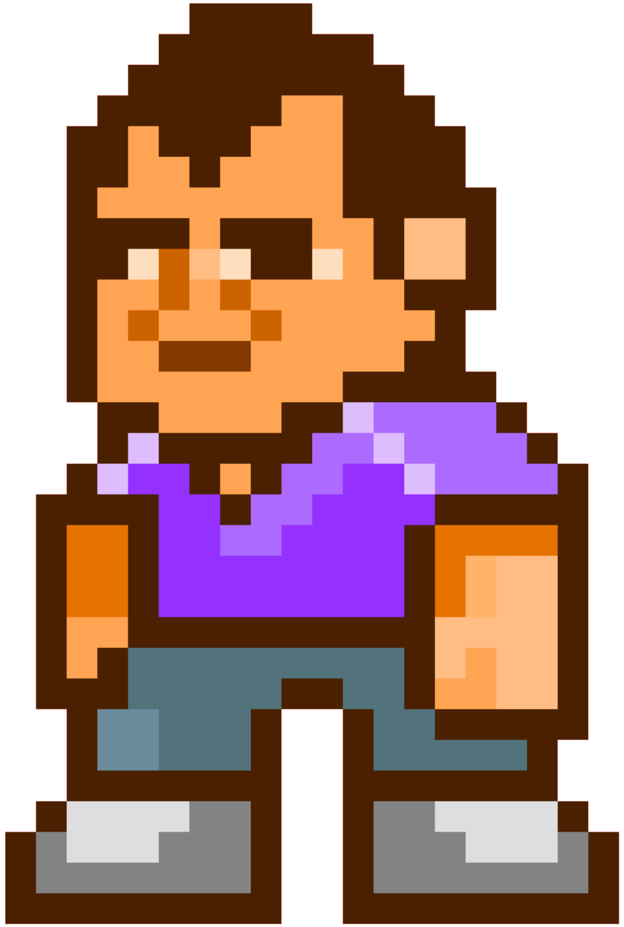
What is happening when a WebGL page is loaded

- User enters URL
- HTTP stack requests the HTML page
- Additional requests will be necessary to get JavaScript™ code and other resources
- JavaScript code will be pre-parsed while loading other assets and the DOM tree is built
- JavaScript code will contain calls to the WebGL API
  - They will go back to WebKit®, which calls OpenGL® ES 2.0 library
  - Shaders are compiled
  - Textures, vertex buffers & uniforms must be loaded to the GPU
  - Rendering can start



See Chromium Rendering Stack: http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

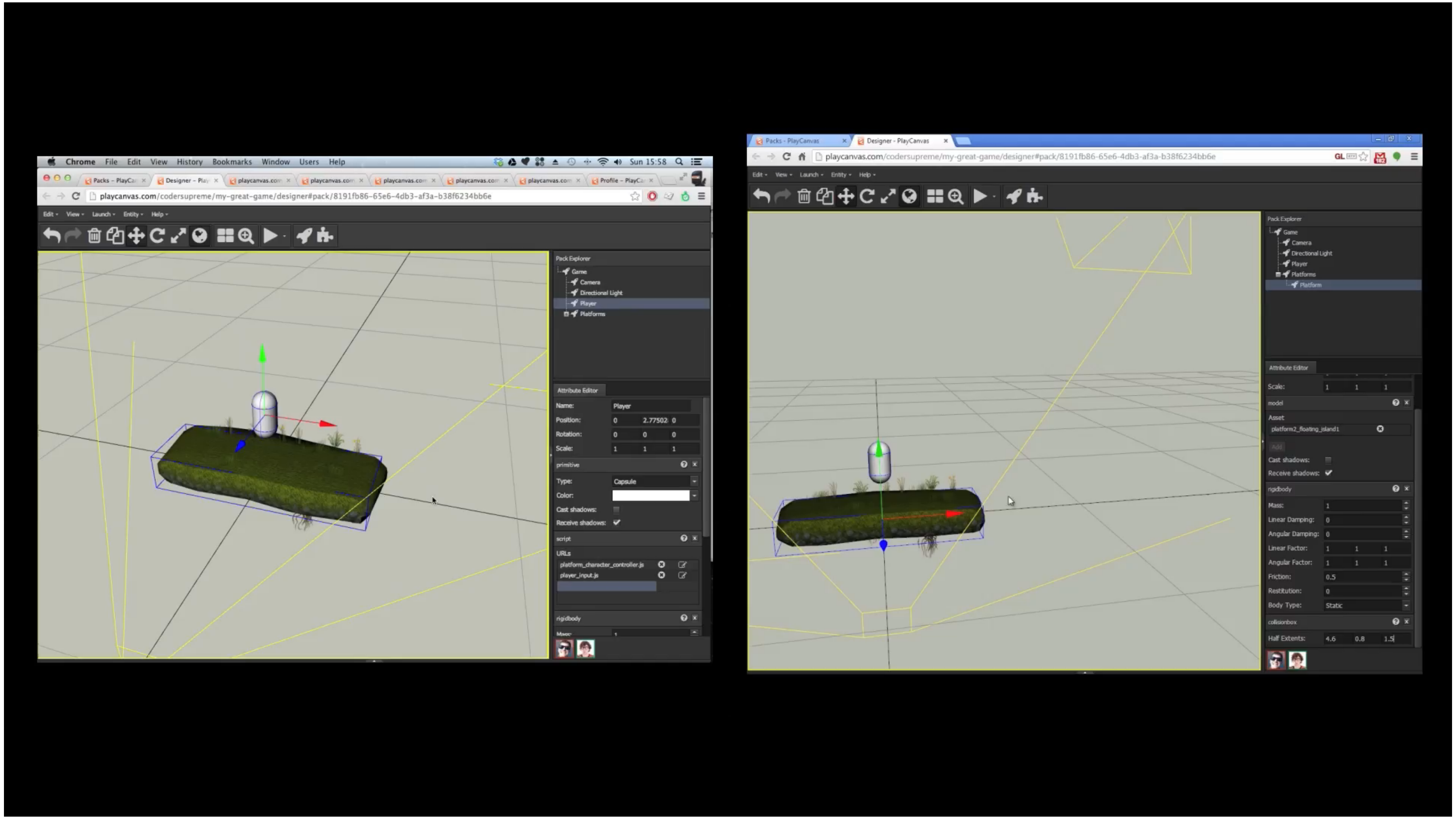
Introducing Me…



…and PlayCanvas

PlayCanvas is the world's easiest to use WebGL Game Engine. It's free, it's open source and it's backed by amazing developer tools.

Google Docs for Games: Realtime Collaboration

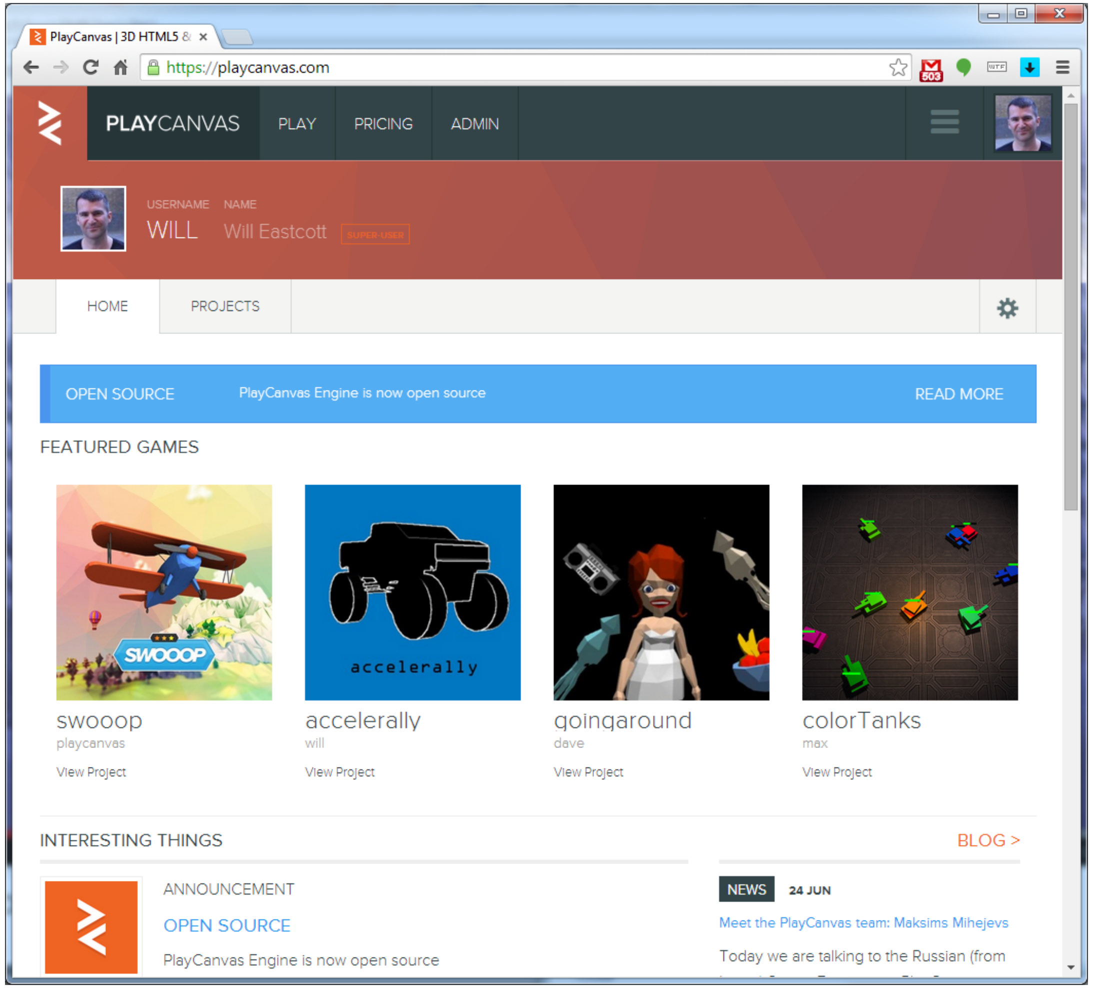


ARM

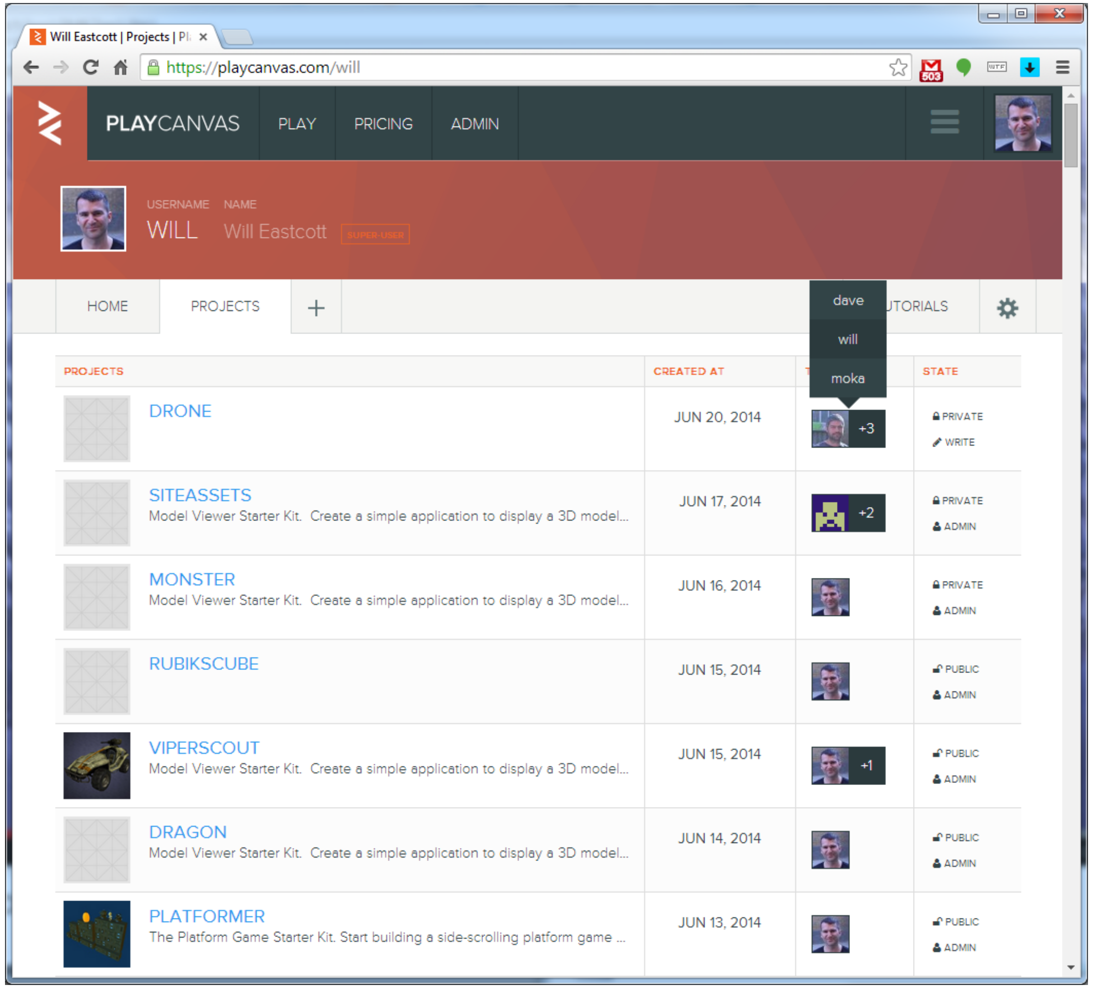
Game Development Goes Mobile on ARM

ARM

At Last: A *Real* Community for Game Dev

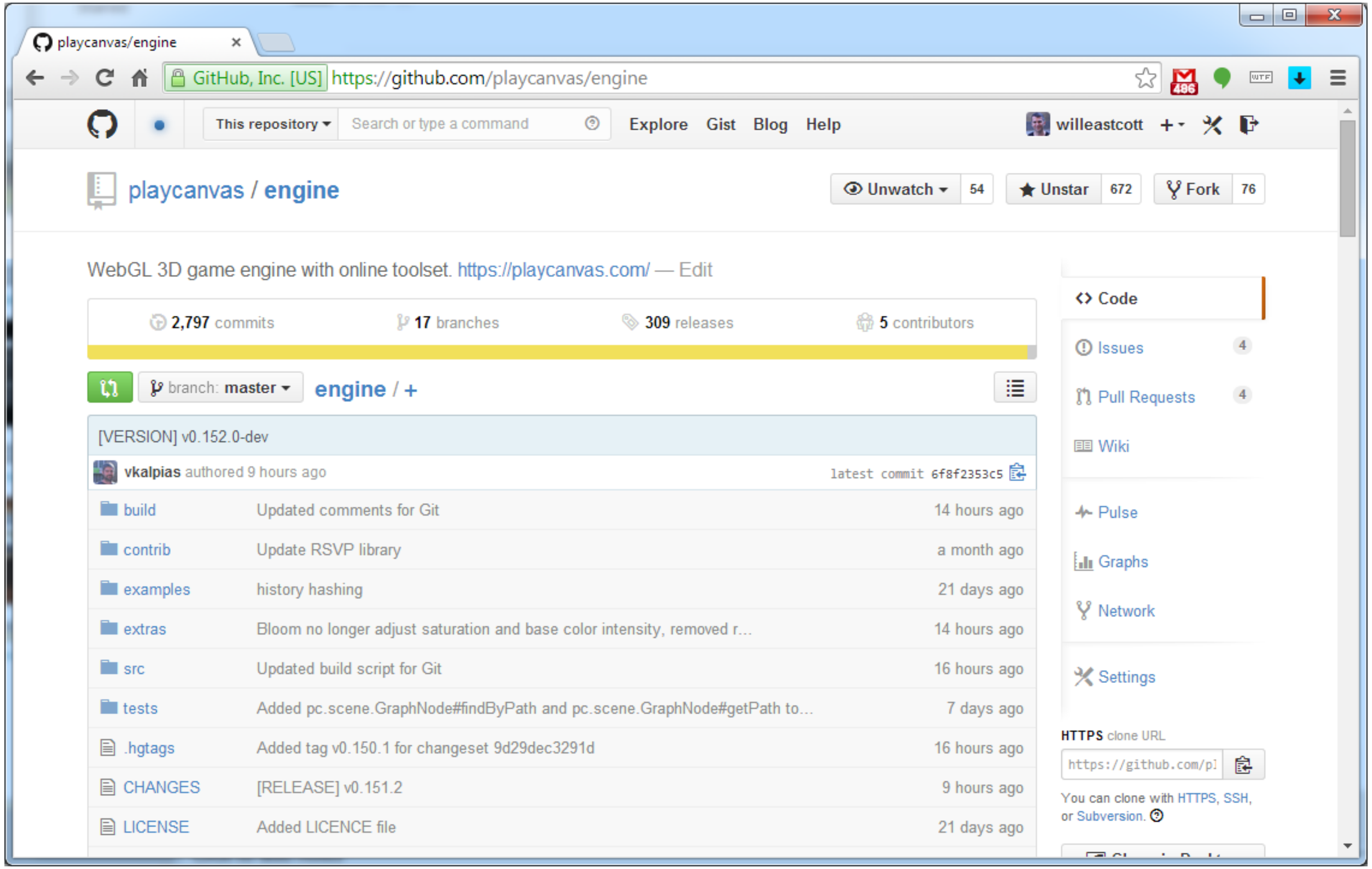


At Last: A *Real* Community for Game Dev



ARM

Open Sourced: https://github.com/playcanvas/engine



ARM

The Building of a WebGL Game: SWOOOP

ARM

What We Did Right

- We didn't use physics
- We didn't use realtime shadows
- We didn't use post effects
- We adopted an art style which only required low res texturing
- We kept the number of draw calls below 150
- We added visual flare with cheap GPU based effects like particles and UV scrolling

What We Did Wrong

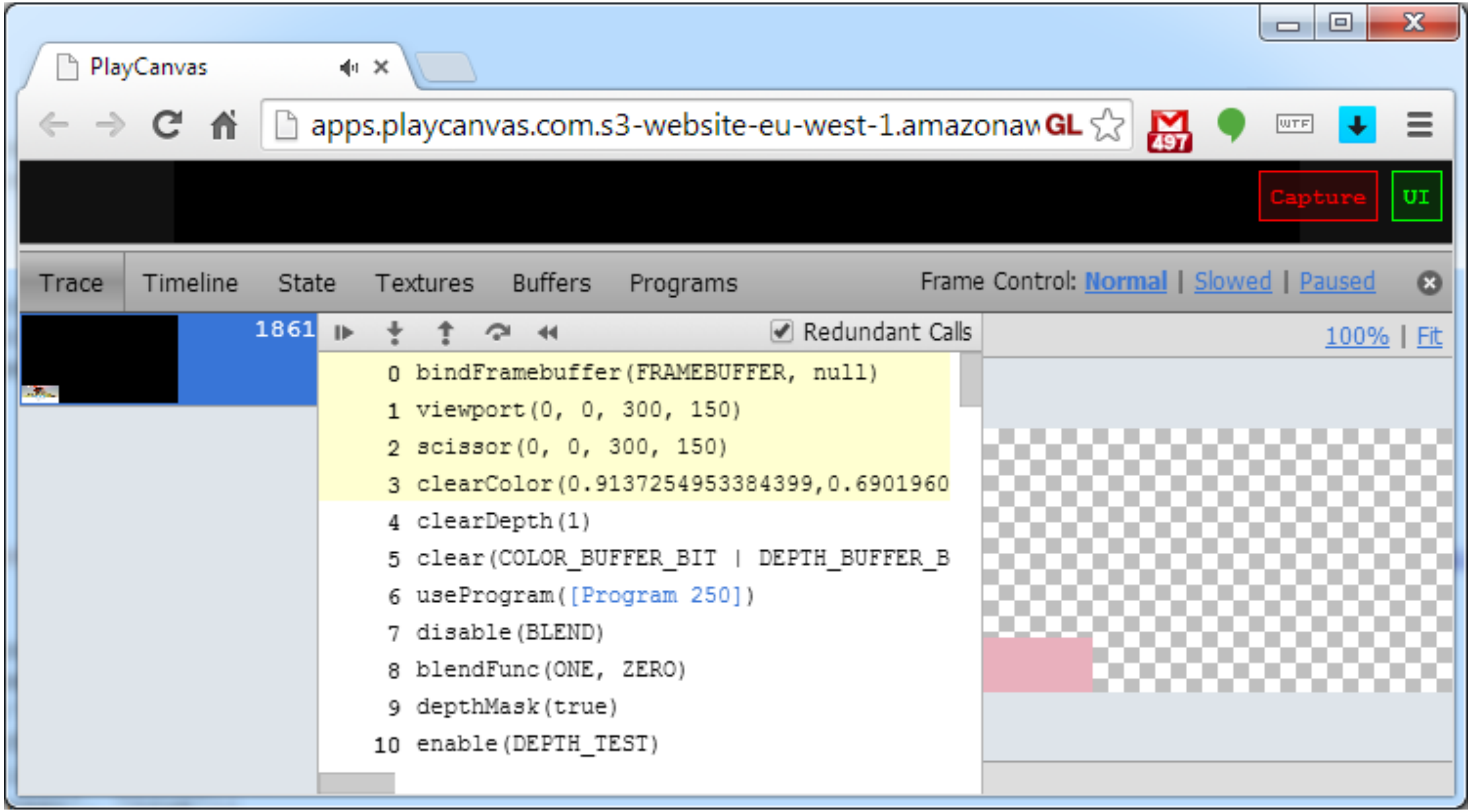
- We adopted an art style which generated a lot of vertex data
- Keeping draw calls low generated more vertex data
- We used realtime lighting on the environment
- Each gem was a separate draw call

Optimizing Your WebGL Code: Options

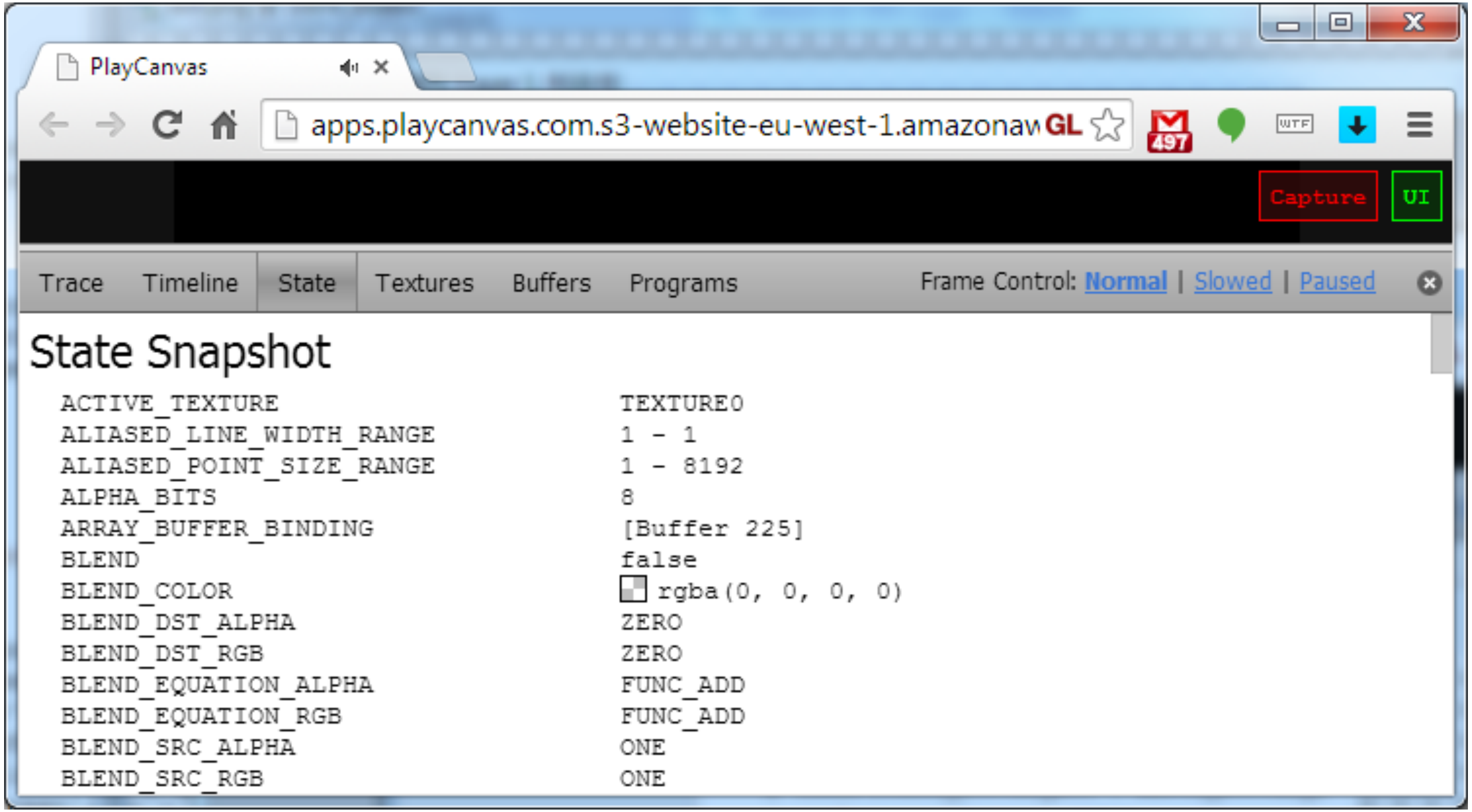
- Learn from the Open Source community
- In-browser Developer Tools
- GLSL Optimizer (https://github.com/aras-p/glsl-optimizer)
- WebGL Inspector (http://benvanik.github.io/WebGL-Inspector/)

ARM

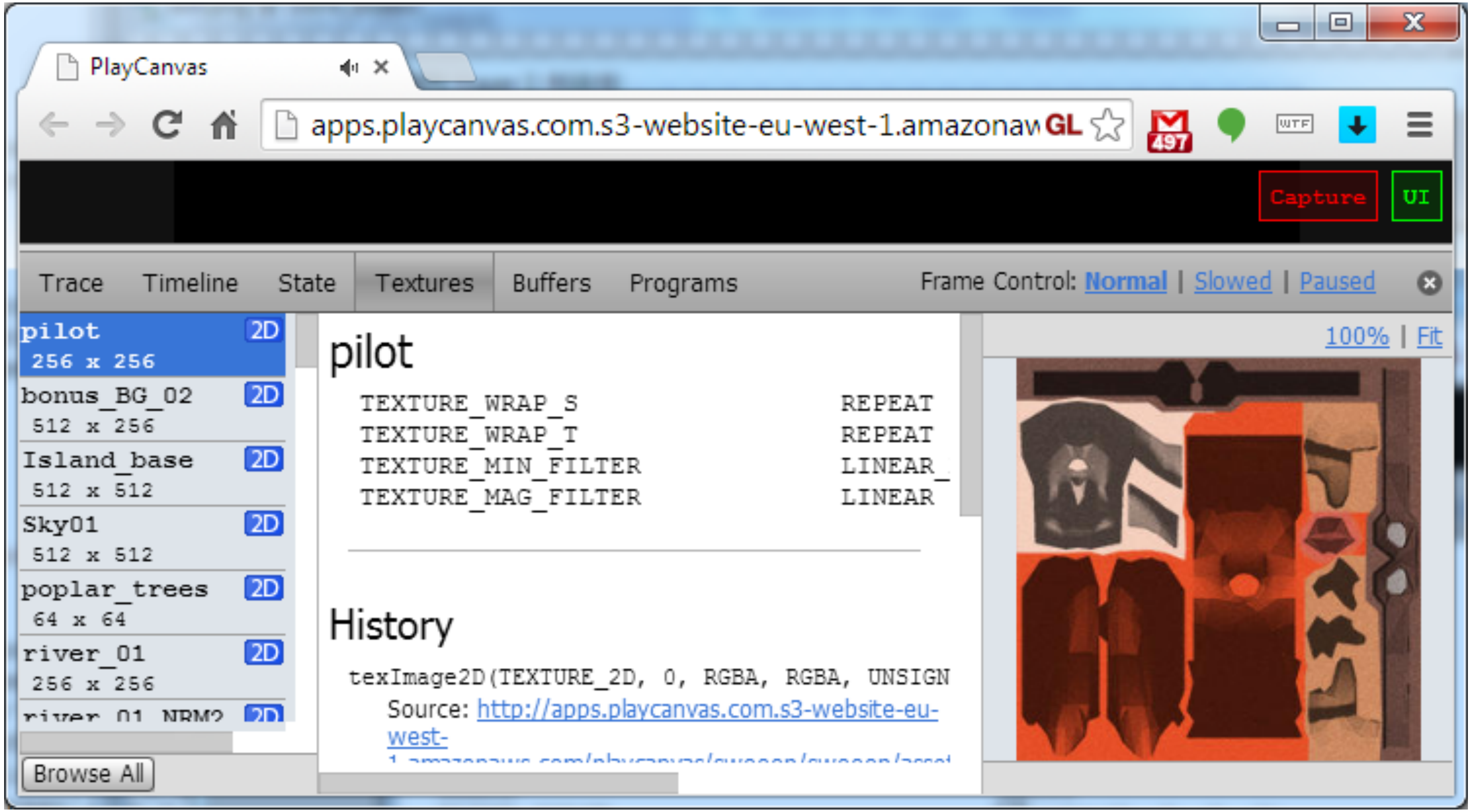
WebGL Inspector: Function Tracing



WebGL Inspector: Render State

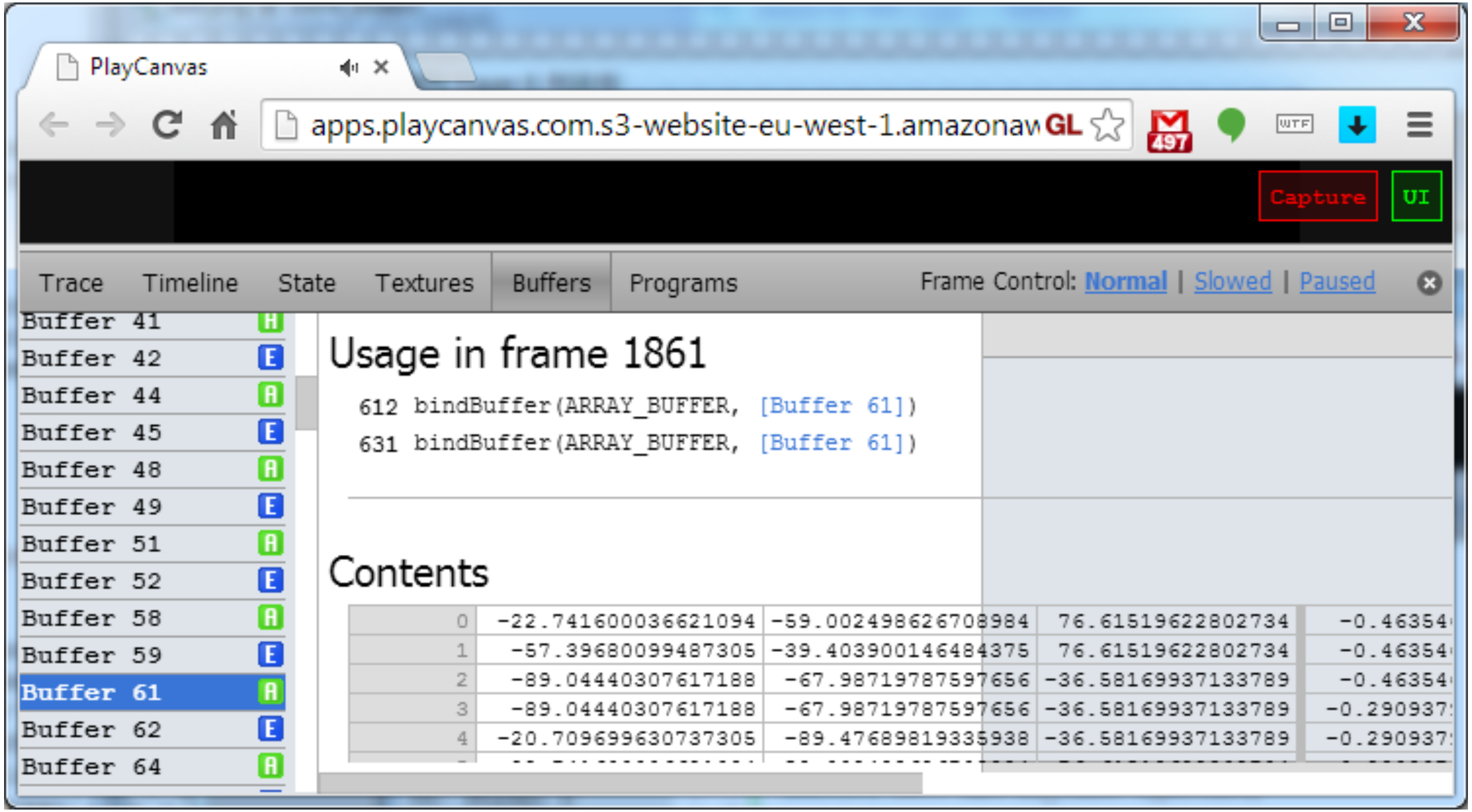


WebGL Inspector: Textures



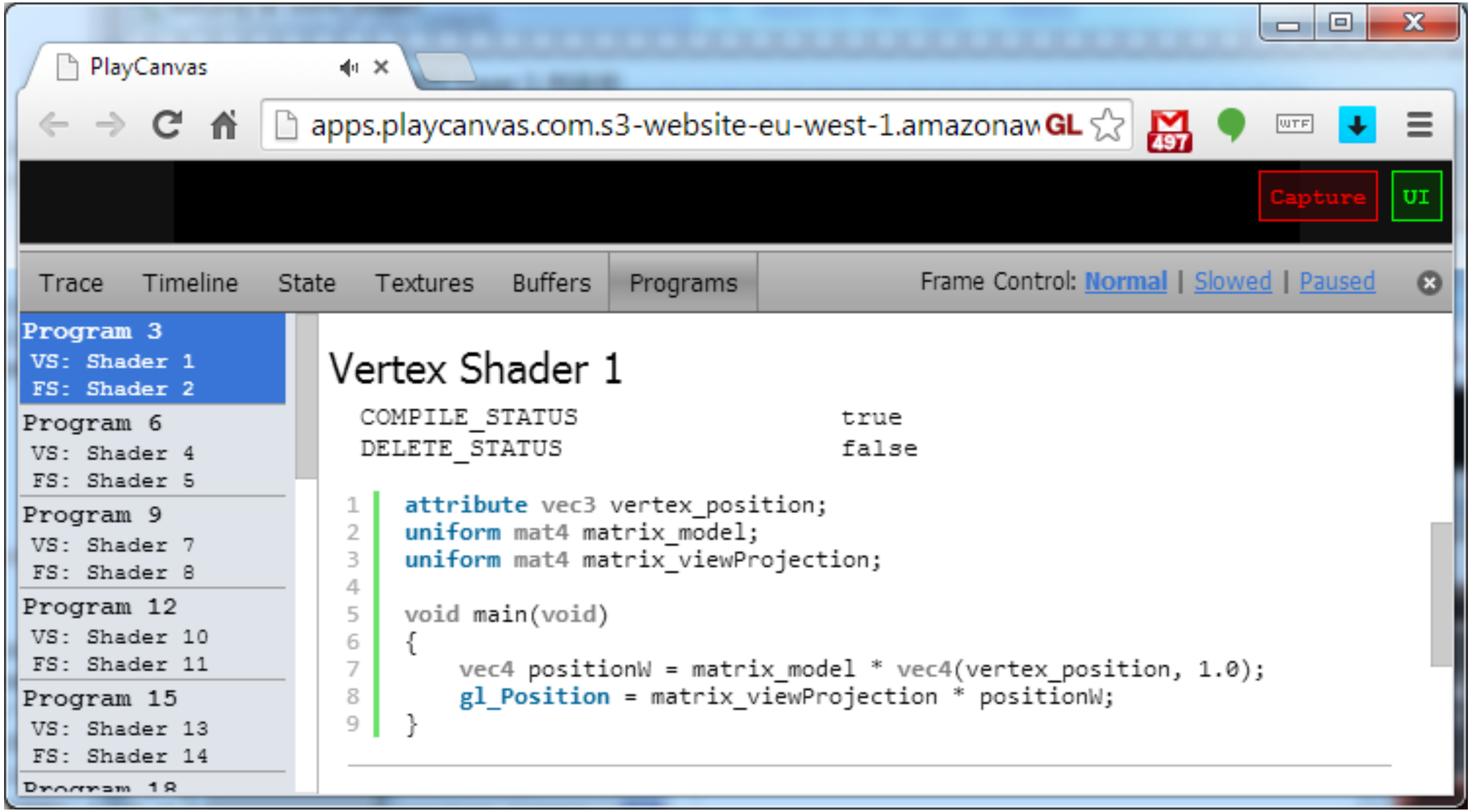
ARM

WebGL Inspector: Vertex and Index Buffers



ARM

WebGL Inspector: Shader Code



Understanding the GPU is Key

- Easy to optimize your graphics pipeline on the CPU
- Harder to know how to optimize on the GPU
- Use ARM tools to get special insight into how your graphics data is being processed

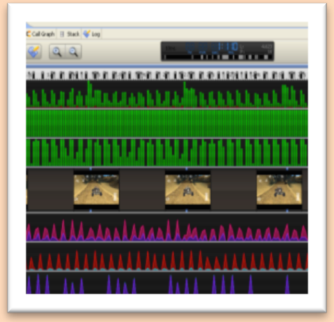
Importance of Analysis & Debug

- 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 can do the heavy lifting

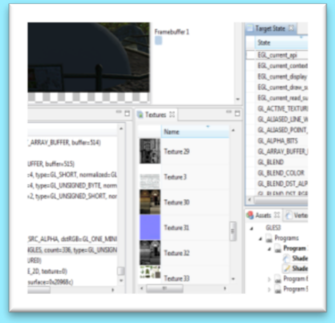
ARM

Performance Analysis & Debug



ARM® DS-5 Streamline Performance Analyzer

- System-wide performance analysis
- Combined ARM Cortex® Processors and Mali™ GPU visibility
- Optimize for performance & power across the system



ARM Mali Graphics Debugger

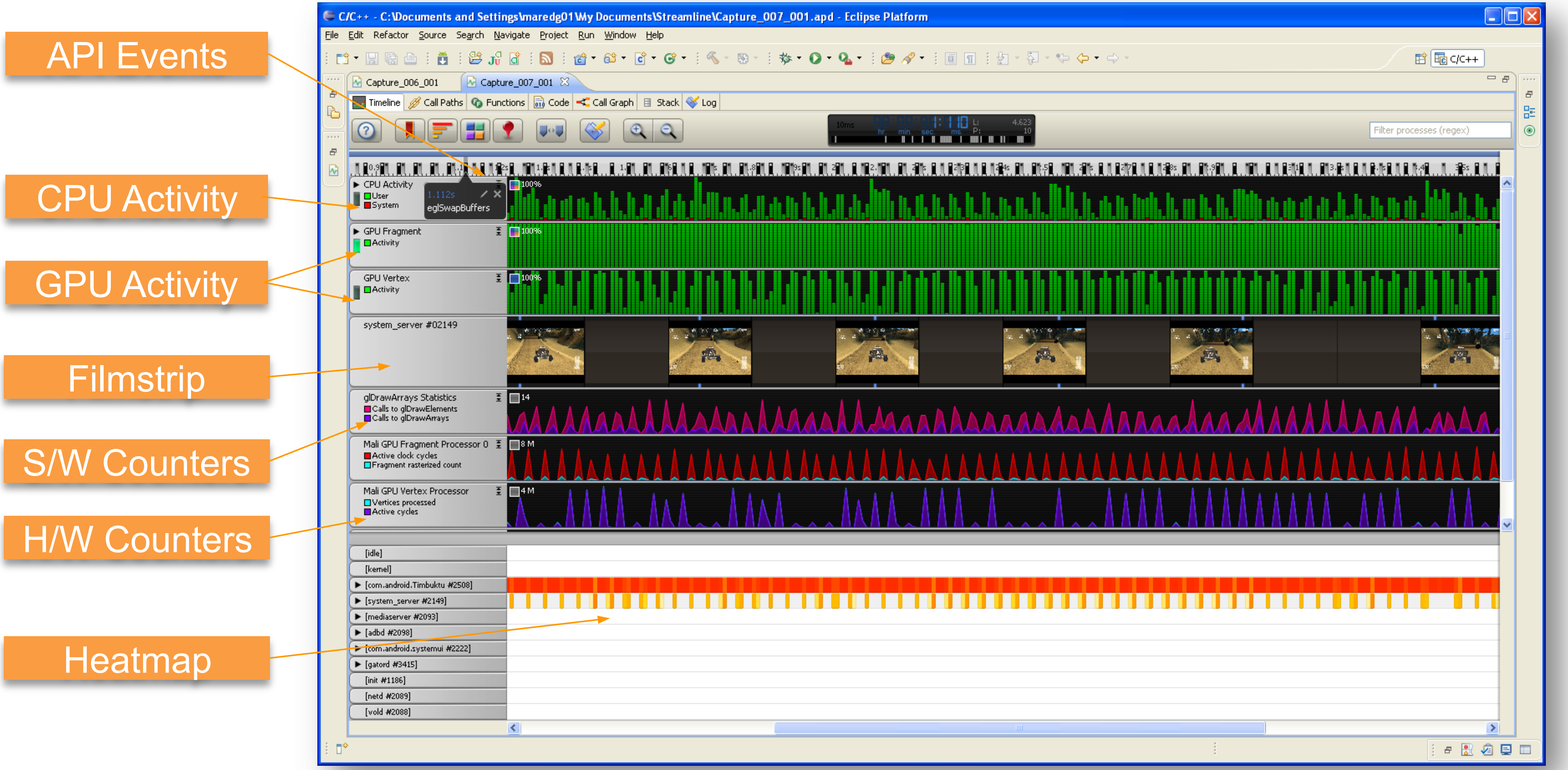
- API Trace & Debug Tool
- Understand graphics and compute issues at the API level
- Debug and improve performance at frame level
- Support for OpenGL® ES 1,1, 2.0, 3.0 and OpenCL™ 1.1

Offline Compilers

- Understand complexity of GLSL shaders and CL kernels
- Support for ARM Mali-4xx and Mali-T6xx GPU families

ARM

ARM® DS-5 Streamline Performance Analyzer

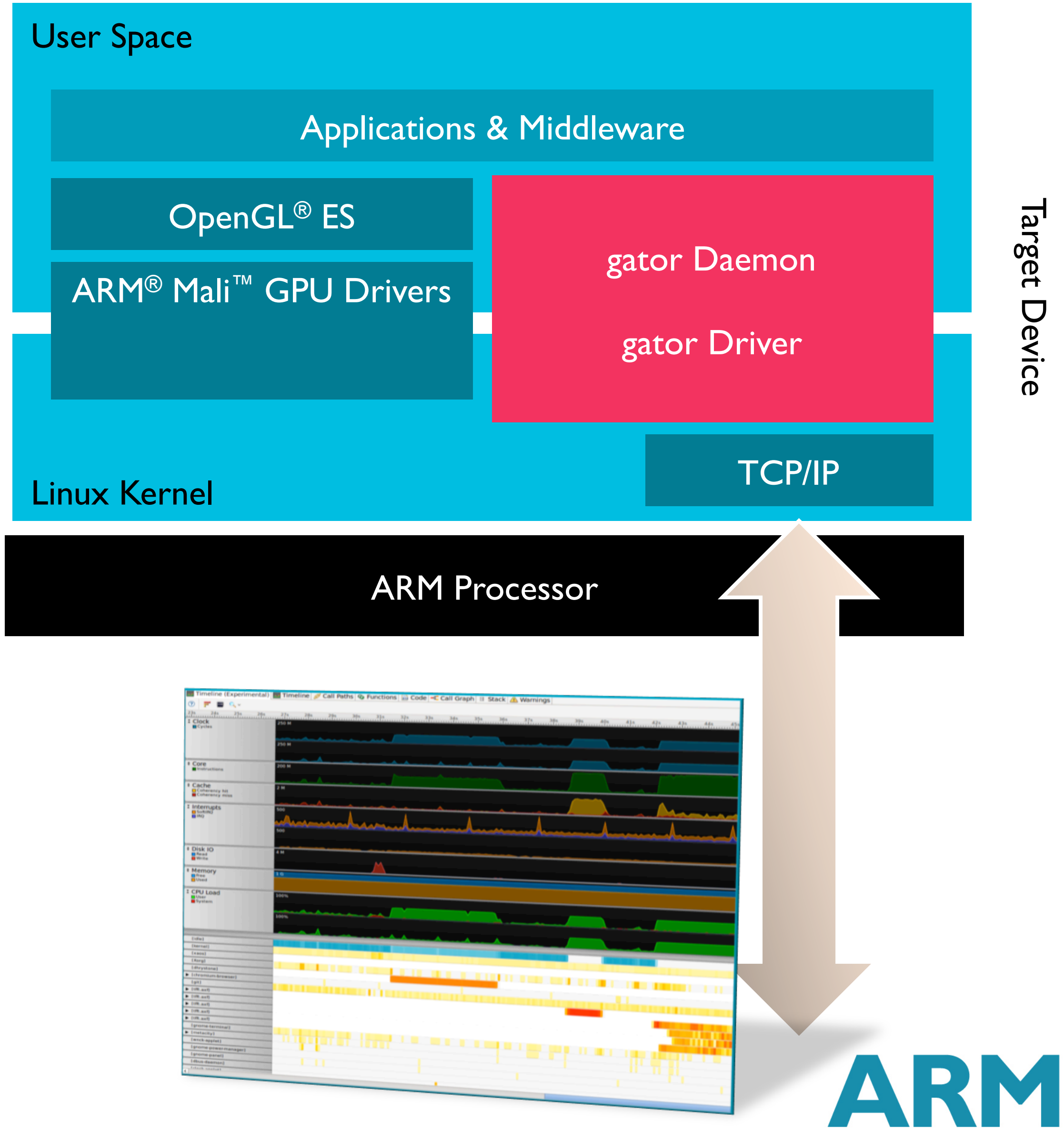


The Basics

- Software based solution
  - ICE/trace units not required
  - Support for Linux kernel 2.6.32+ on target
  - Eclipse plug-in or command line

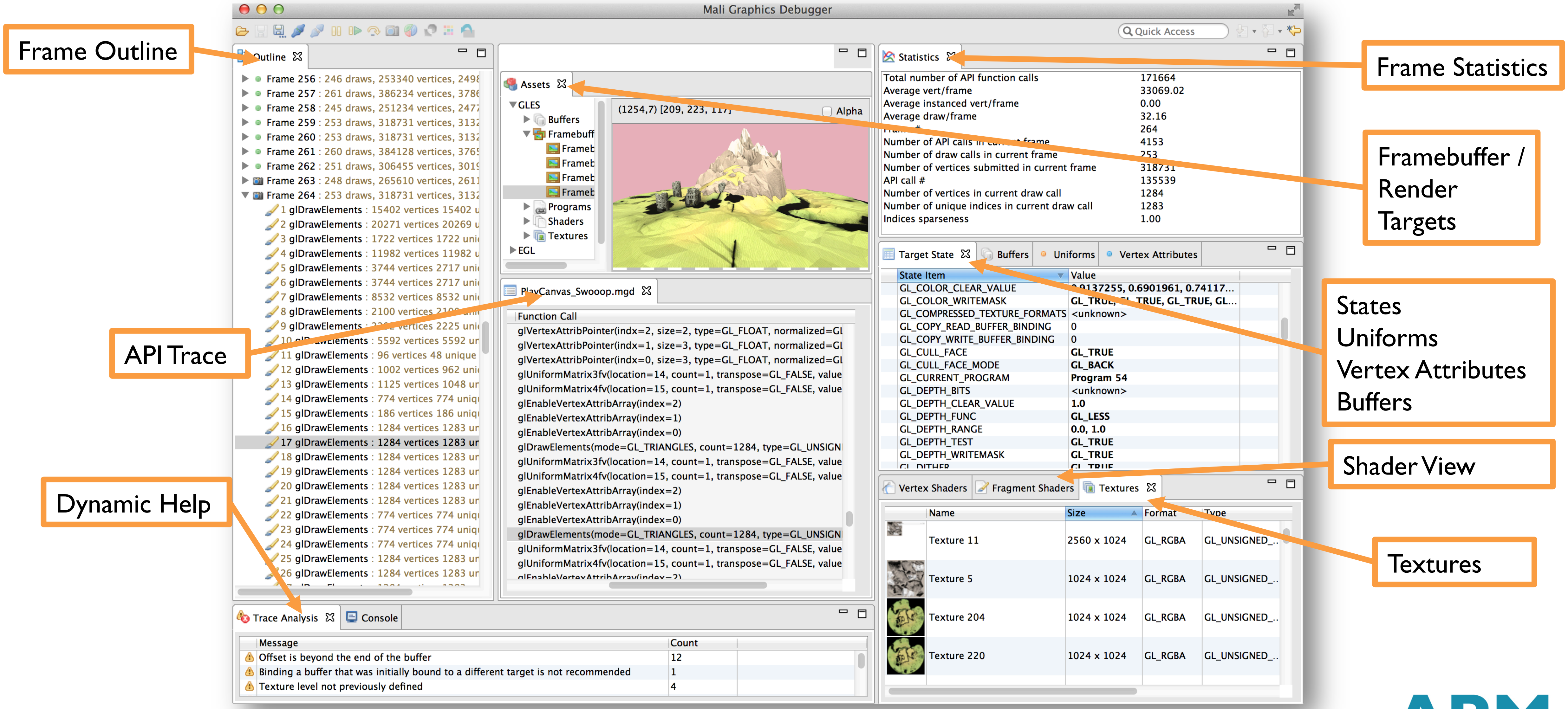
- Lightweight sample profiling
  - Time- or event*-based sampling
  - Process to C/C++ source code profiler
  - Low probe effect; <5% typically

- Multiple data sources
  - CPU, GPU and Interconnect hardware counters
  - Software counters and kernel tracepoints
  - User defined counters and instrumented code
  - Power/energy measurements



* Event-based sampling is available on kernels 3.0 or later

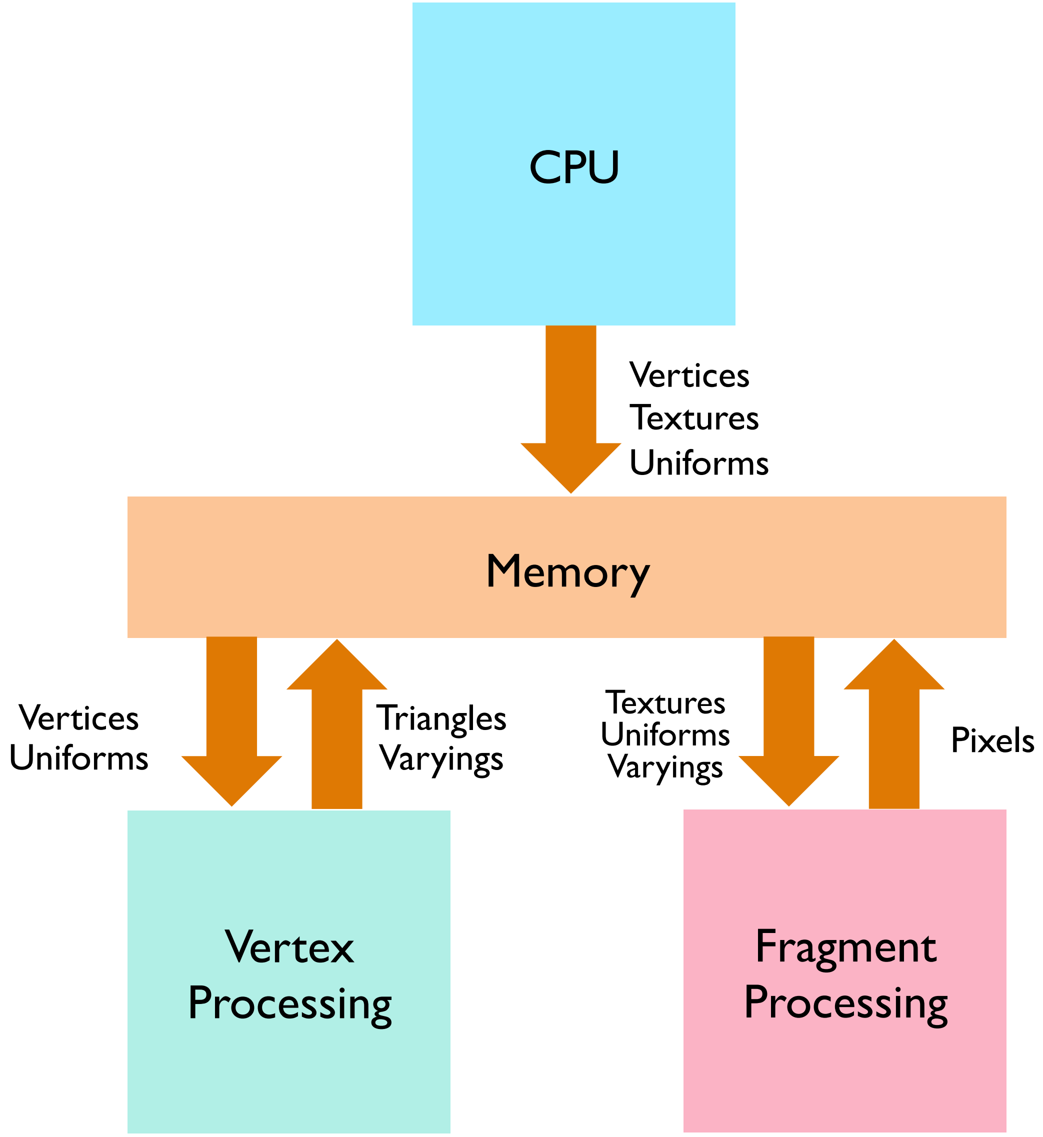
ARM® Mali™ Graphics Debugger



Main Bottlenecks (1)

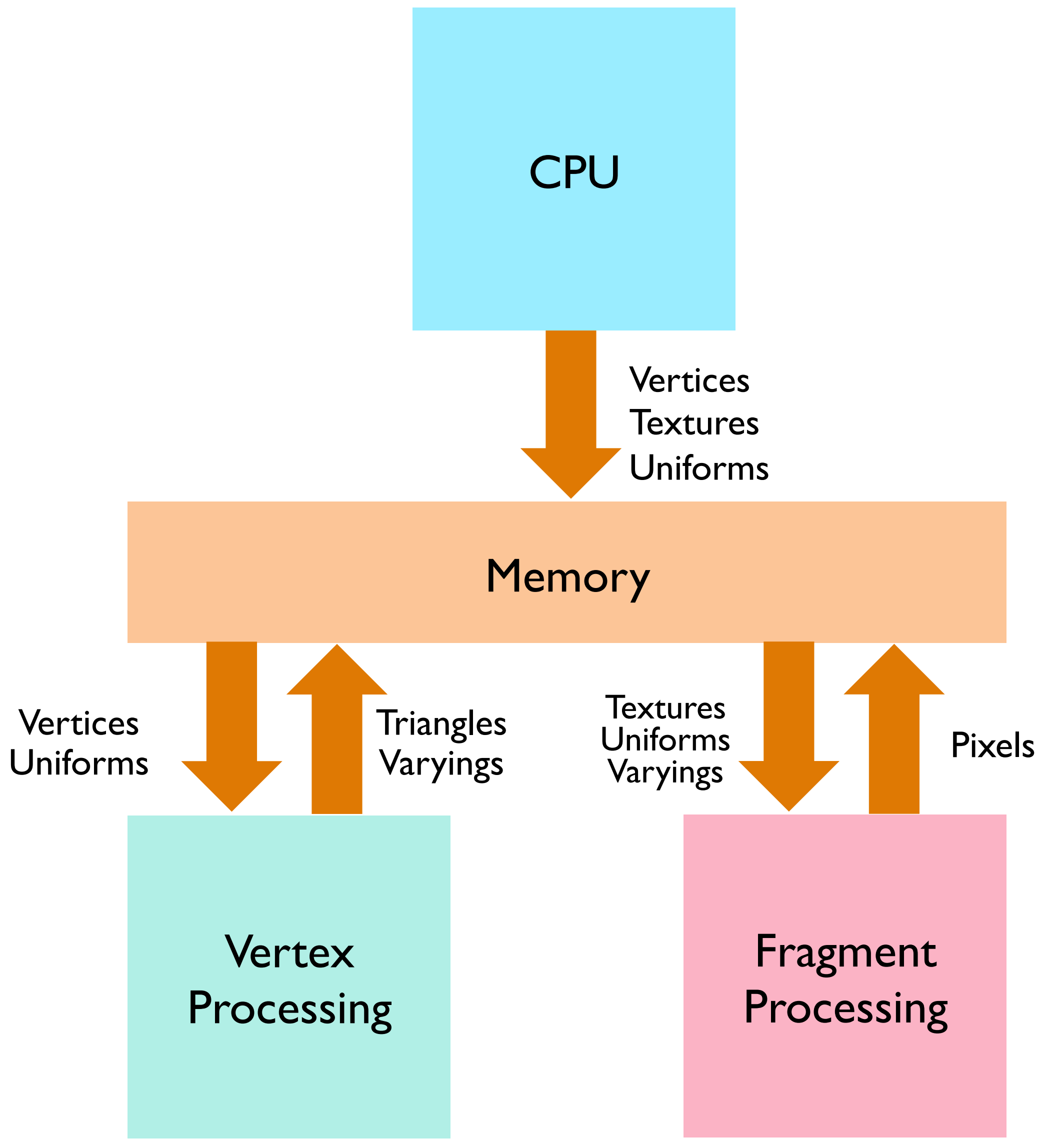
- The frame rate of a particular WebGL™ application could be limited by:
  - CPU
  - Vertex Processing
  - Fragment Processing
  - Bandwidth

- Fortunately we have tools to understand which one is the culprit



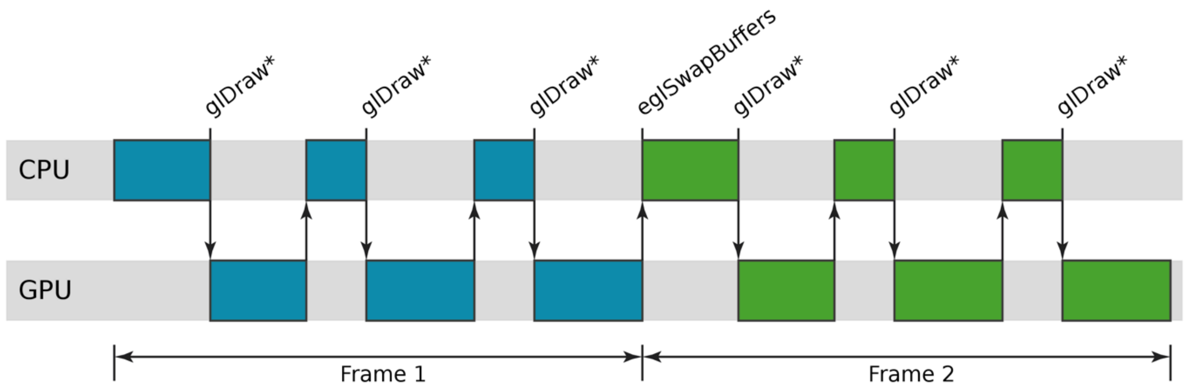
Main Bottlenecks (2)

- **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

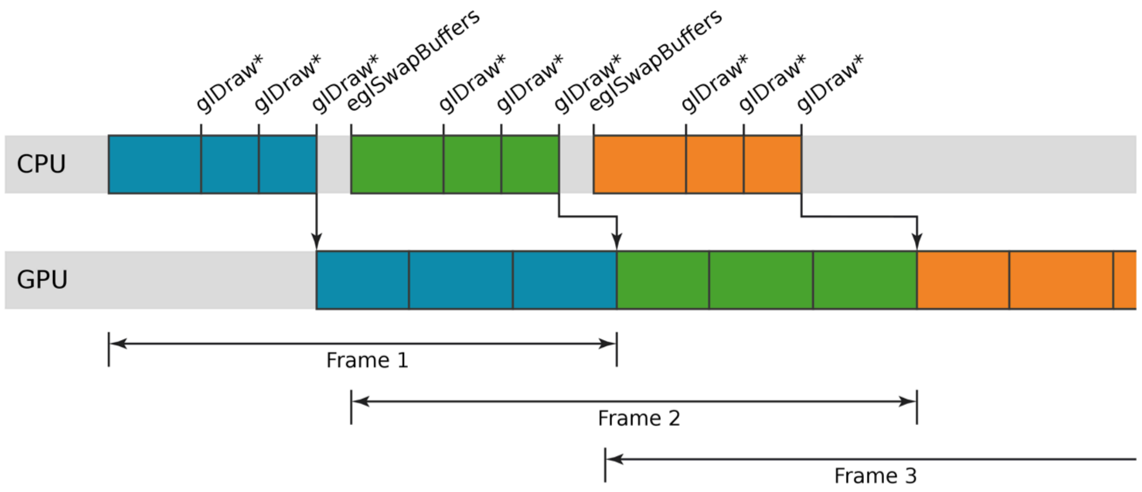


Frame Rendering Time

Synchronous Rendering



Deferred Rendering

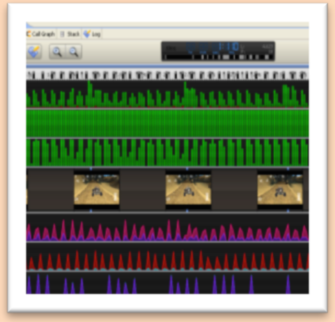



```
// THIS DOES NOT MEASURE GPU RENDERING

var start = new Date().getTime();
gl.drawElements(gl.TRIANGLE, …);
var time = new Date().getTime() - start;
```

```
// THIS FORCES SYNCHRONOUS RENDERING
// (BAD PRACTICE)

var start = new Date().getTime();
gl.drawElements(gl.TRIANGLE, …);
gl.finish(); // or gl.readPixels…
var time = new Date().getTime() - start;
```


ARM

Workflow

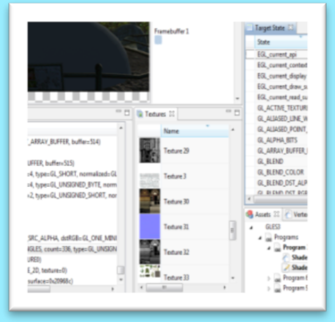


ARM® DS-5 Streamline Performance Analyzer

What kind of problem do we have?

Detailed analysis

Validate



ARM Mali Graphics Debugger

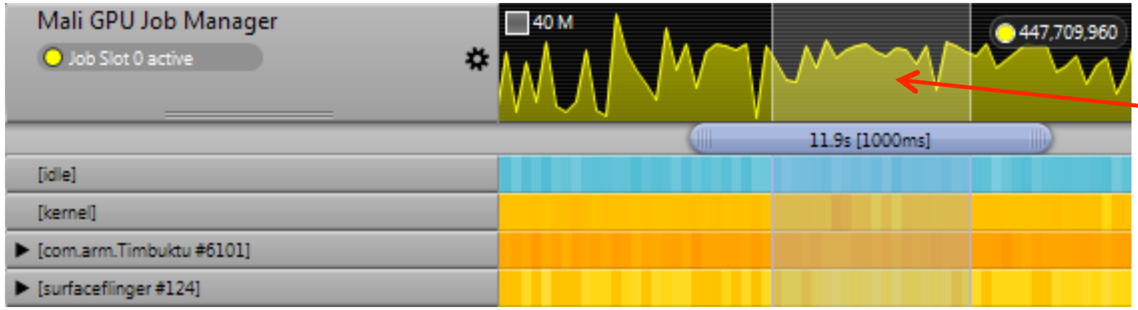
What's causing the problem?

How can I fix it?

Fragment Bound

ARM DS-5 Streamline: Fragment Bound

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



Fragment Percentage = (Job Slot 0 active / Frequency) *100

Fragment Percentage = 84%

Overdraw = Fragment Threads Started * Number of Cores / Resolution * FPS

Overdraw = 3.9

Fragment Bound

- Resolution too high or too many effects or cycles in the 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

Nexus 10 Native Resolution

- 2560 x 1600 = 4,096,000 pixels

Quad Core GPU 533Mhz

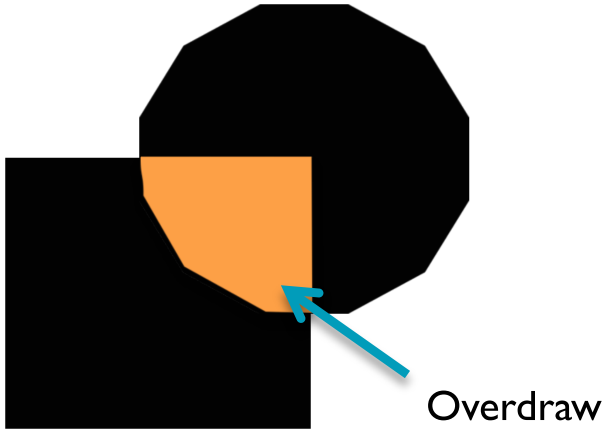
- 520 Cycles per pixel Approx.

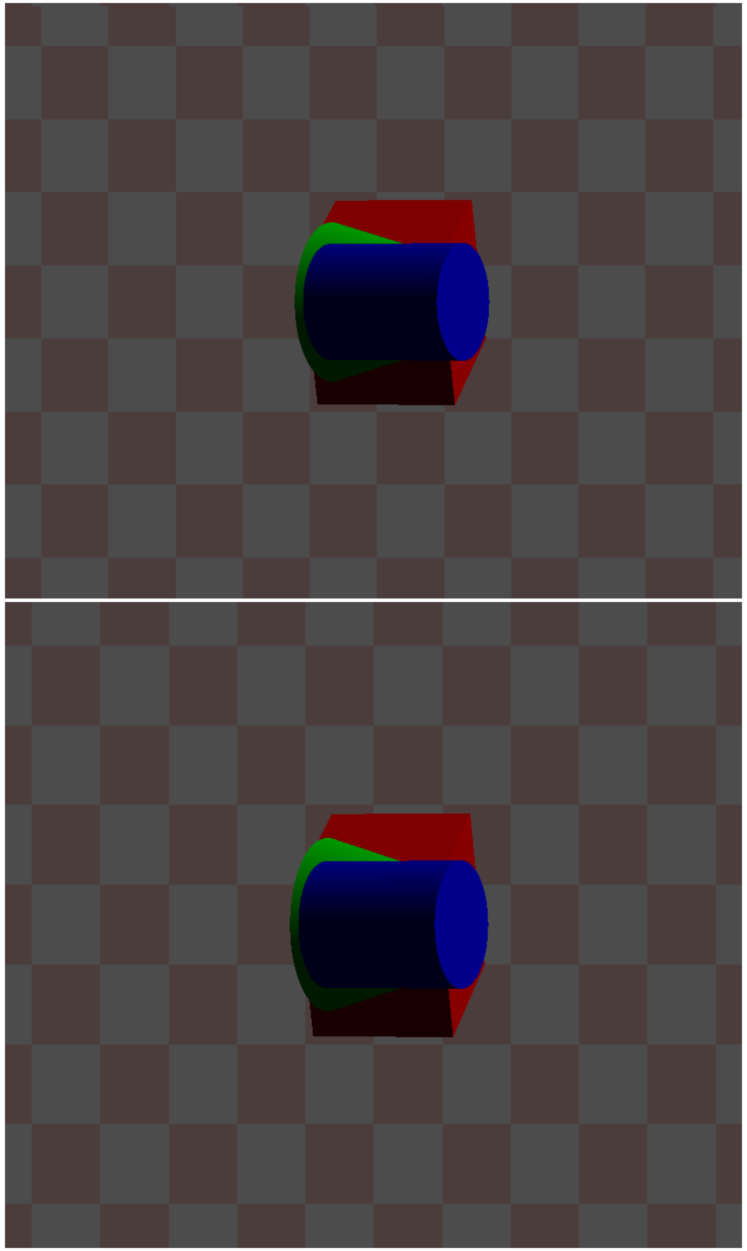
Targeting 30 FPS

- 17 Cycles in your shader

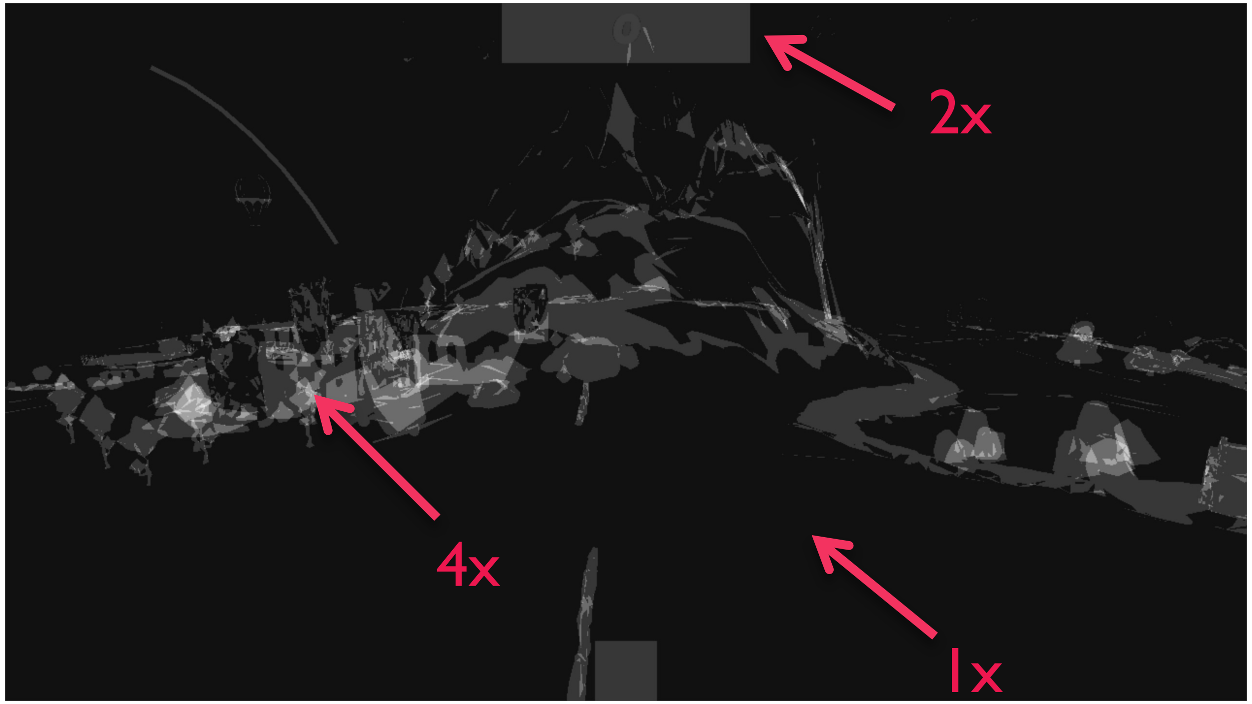
Overdraw

- 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
- Limiting the amount of transparency in the scene can help



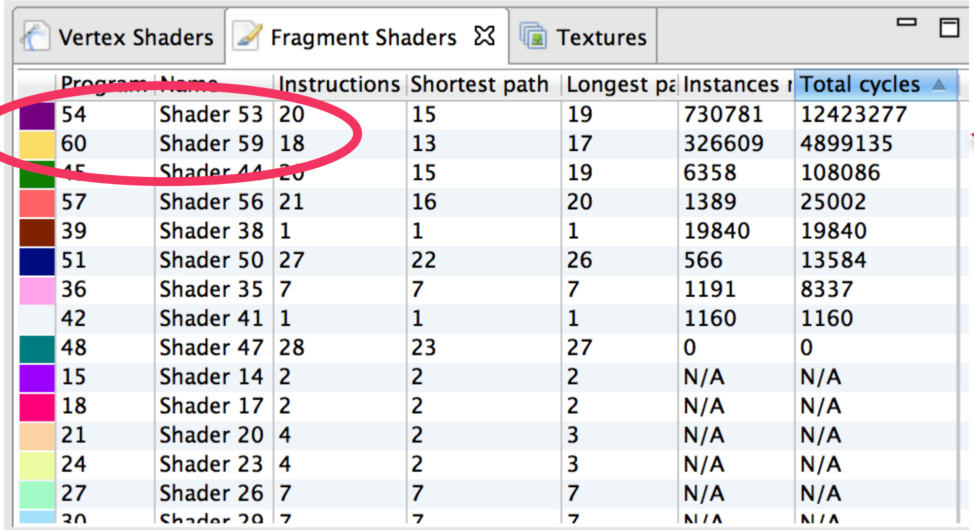


ARM Mali Graphics Debugger: Overdraw



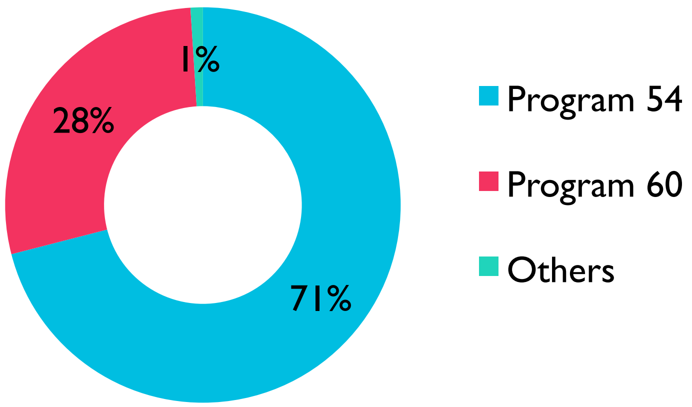
ARM

ARM Mali Graphics Debugger: Shader Map & Fragment Count



	Program	Name	Instructions	Shortest path	Longest pa	Instances r	Total cycles
	54	Shader 53	20	15	19	730781	12423277
	60	Shader 59	18	13	17	326609	4899135
	45	Shader 44	20	15	19	6358	108086
	57	Shader 56	21	16	20	1389	25002
	39	Shader 38	1	1	1	19840	19840
	51	Shader 50	27	22	26	566	13584
	36	Shader 35	7	7	7	1191	8337
	42	Shader 41	1	1	1	1160	1160
	48	Shader 47	28	23	27	0	0
	15	Shader 14	2	2	2	N/A	N/A
	18	Shader 17	2	2	2	N/A	N/A
	21	Shader 20	4	2	3	N/A	N/A
	24	Shader 23	4	2	3	N/A	N/A
	27	Shader 26	7	7	7	N/A	N/A
	30	Shader 29	7	7	7	N/A	N/A

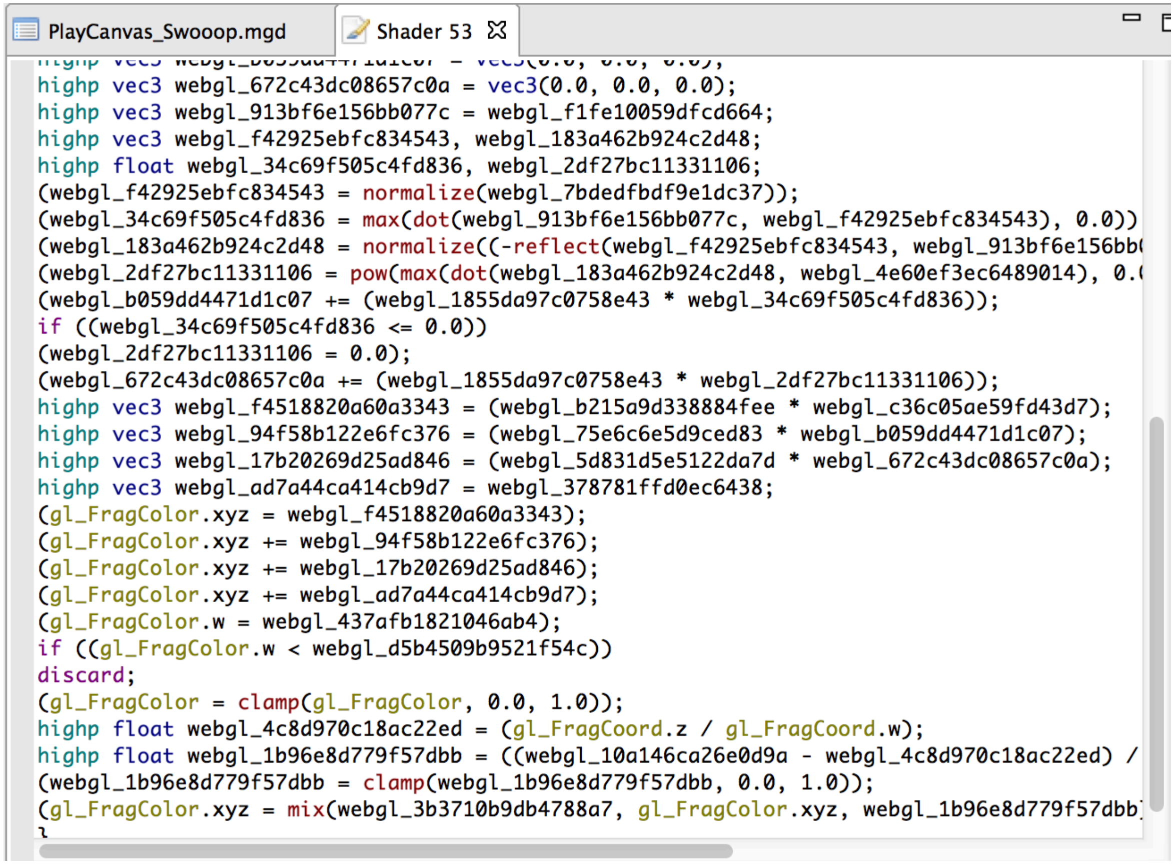
Total Cycles Per Program



Shader Optimization

- Depending on the arithmetic workload, we could reduce the number of uniforms and varyings and calculate them on-the-fly
- Reduce their size
- Reduce their precision: all the varyings, uniforms and local variables are highp, is that really necessary?
- Use the ARM® Mali™ Offline Shader Compiler!

  http://malideveloper.arm.com/develop-for-mali/tools/analysis-debug/mali-gpu-offline-shader-compiler/



General Tips

Fragment Bound

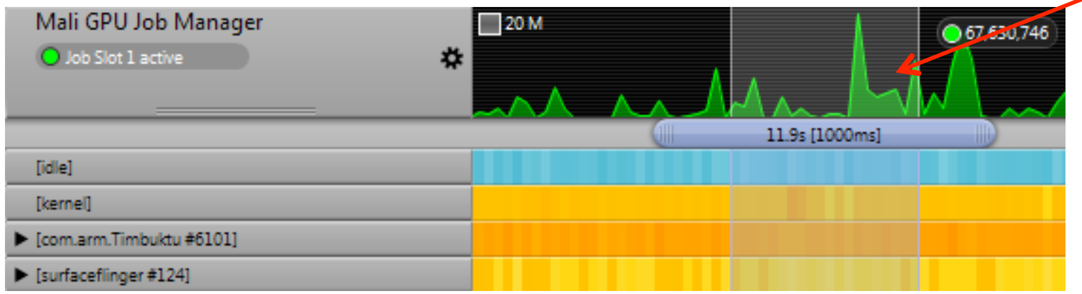
- Render to a smaller framebuffer
  - This will upscale the rendered frame to the size of the HTML canvas
- Move computation from the fragment to the vertex shader (use HW interpolation)
- Drawing your objects front to back instead of back to front reduces overdraw
- Reduce the amount of transparency in the scene

Vertex Bound

ARM DS-5 Streamline: Vertex Bound

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

Vertex Percentage = (Job Slot 1 active / Frequency) *100



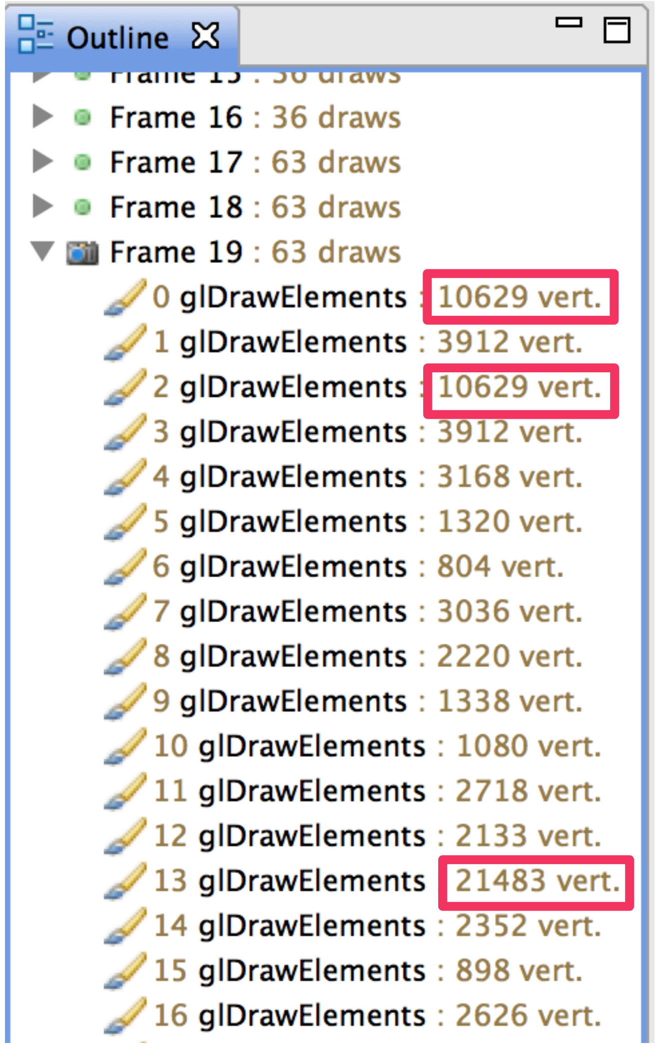
Vertex Percentage = 13%

ARM

ARM Mali Graphics Debugger: 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



ARM Mali Graphics Debugger: Frame Capture

ARM

General Tips

Vertex Bound

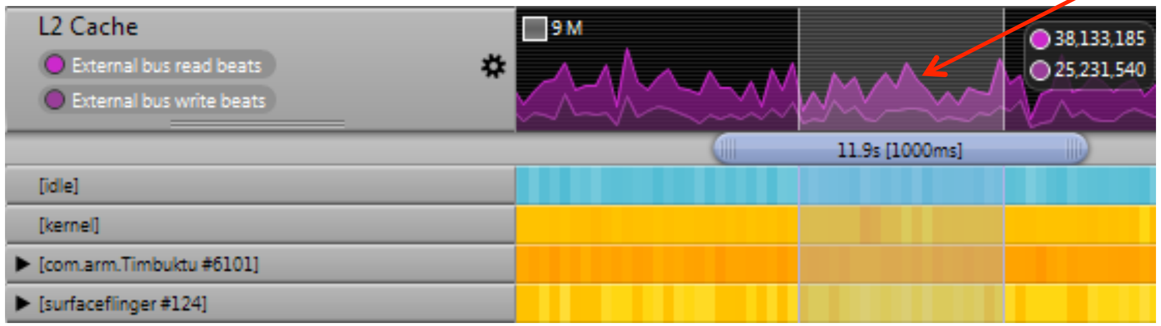
- Get your artist to remove unnecessary vertices
- LOD switching
  - Only objects near the camera need to be in high detail
- Use culling
- Too many cycles in the vertex shader

Bandwidth Bound

ARM DS-5 Streamline: Bandwidth Counters

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

Bandwidth = 967 MB/S



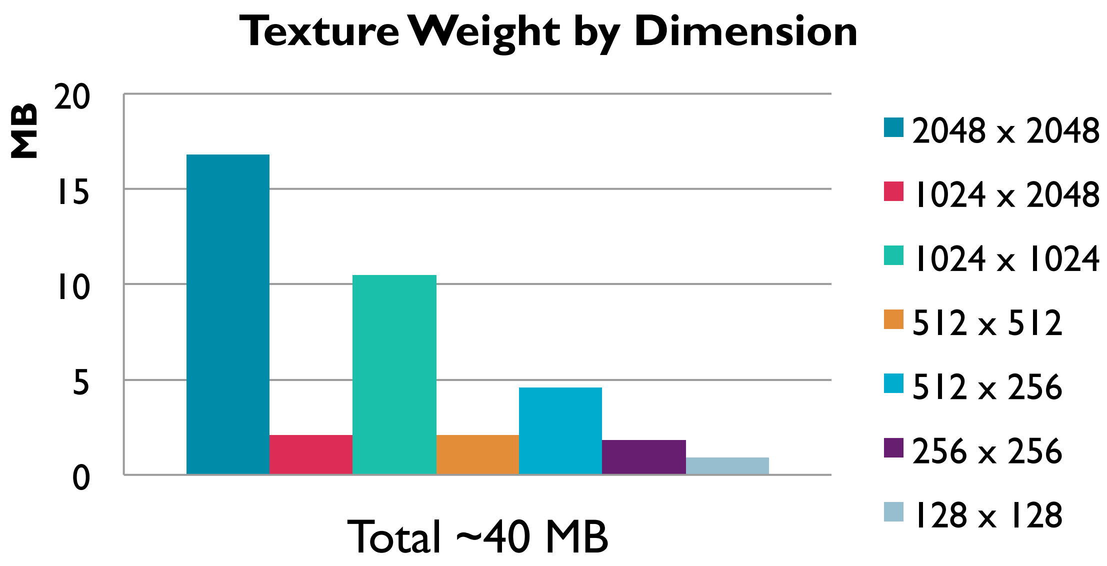
Bandwidth in Bytes = (External Bus Read Beats + External Bus Write Beats) * Bus Width

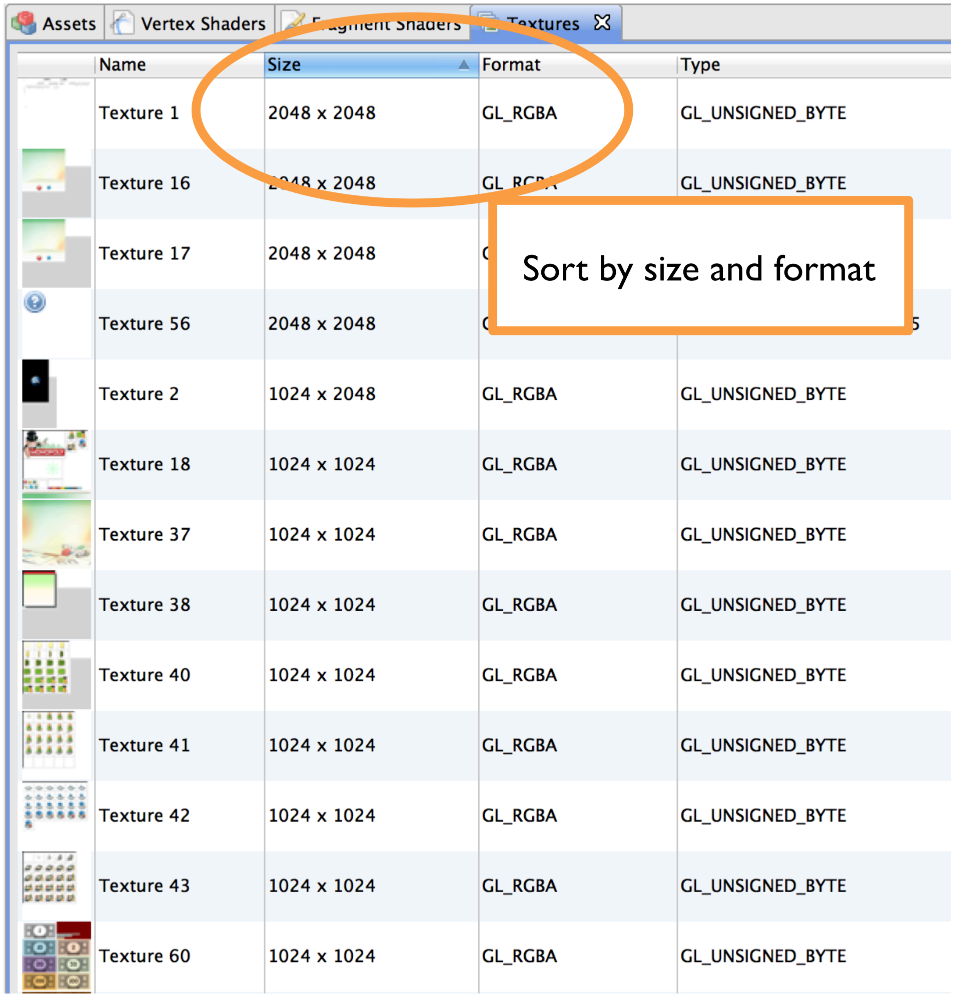
ARM

ARM Mali Graphics Debugger: Textures

Save memory and bandwidth with texture compression

- The current most popular format is ETC Texture Compression
- But ASTC (Adaptive Scalable Texture Compression) can deliver < 1 bit/pixel



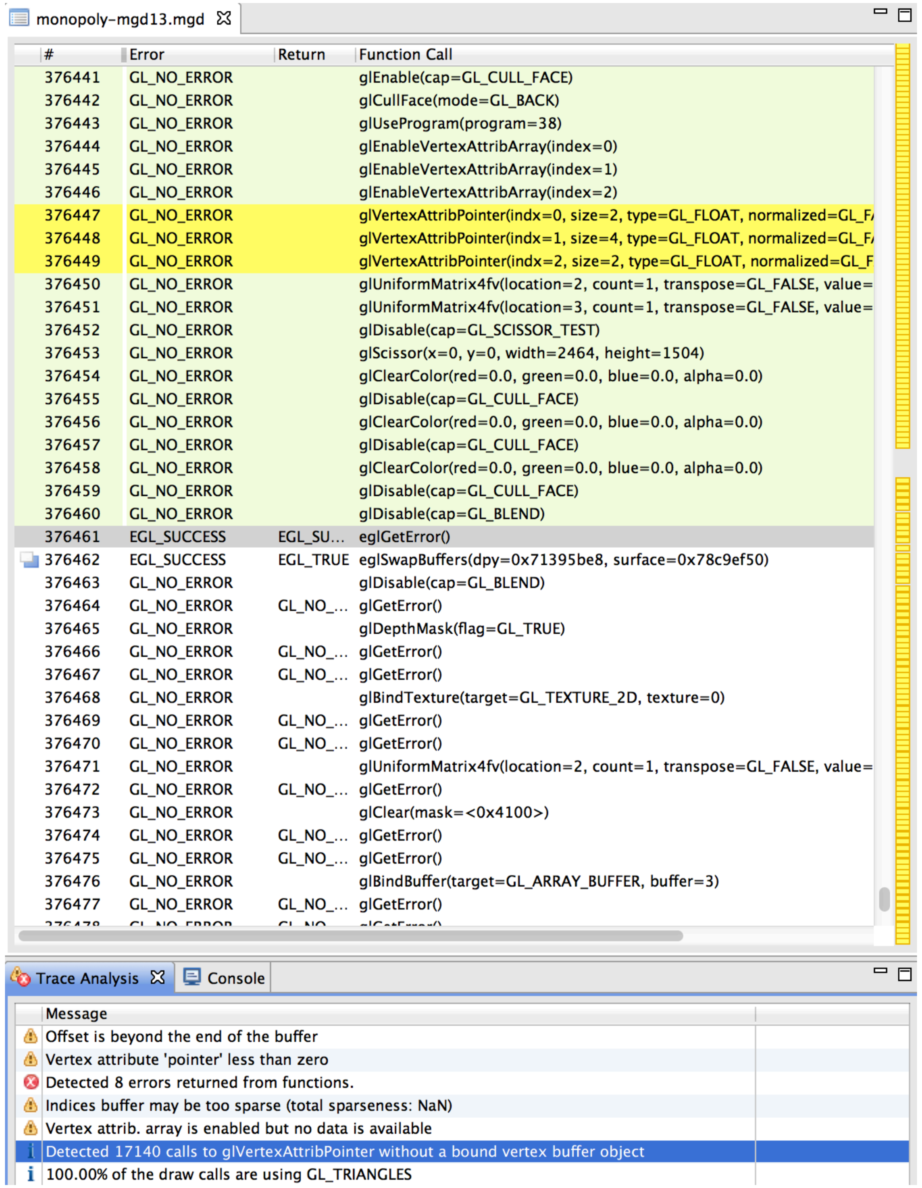


With texture compression, the total amount could be just ~6.4 MB (ASTC 5x5 blocks)

ARM

ARM Mali Graphics Debugger: Vertex Buffer Objects

- Using Vertex Buffer Objects (VBOs) 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



ARM

General Tips

Bandwidth Bound

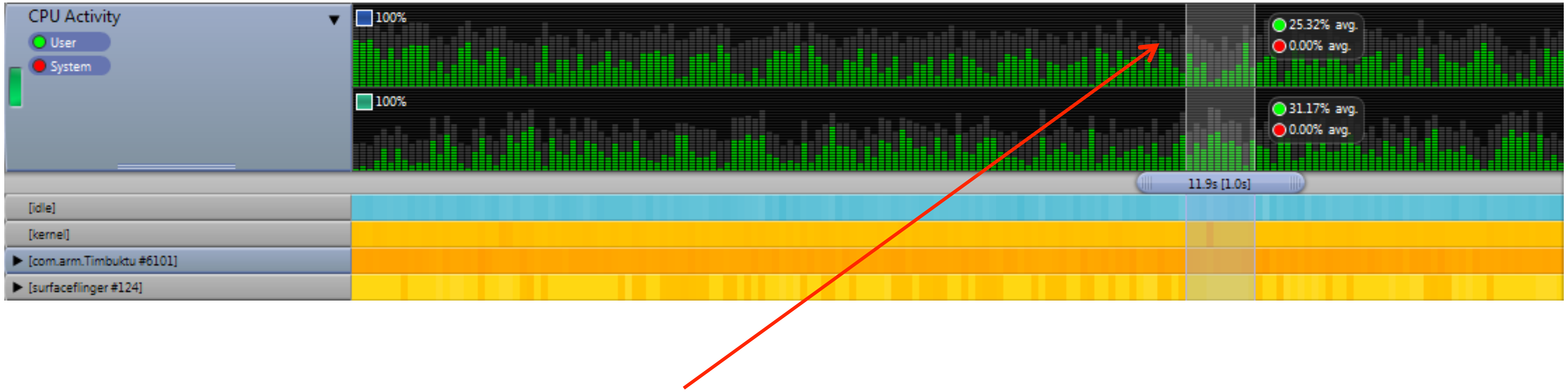
- Use texture compression
- Enable texture mipmapping
- Reduce the number of vertices and varyings
- Interleave vertices, normals, texture coordinates
- Reduce the size of the textures
- Use VBOs

This will also cause a better cache utilization.

CPU Bound

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 process CPU activity

General Tips

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

Summary

- Covered today:
  - Introduction to WebGL
  - PlayCanvas
  - Profiling with ARM DS-5 Streamline
  - Debugging with the ARM Mali Graphics Debugger

- ARM:
  - www.malideveloper.arm.com
  - www.ds.arm.com
  - www.community.arm.com

- WebGL & PlayCanvas:
  - http://www.khronos.org/webgl/
  - https://playcanvas.com
  - https://github.com/playcanvas/engine
  - http://swooop.playcanvas.com
  - https://playcanvas.com/playcanvas/swooop

ARM

Thank You

Any Questions?