Performance Analysis & Debugging Tools

Lorenzo Dal Col
Senior Software Engineer
Media Processing Division, ARM – Cambridge, UK

Agenda

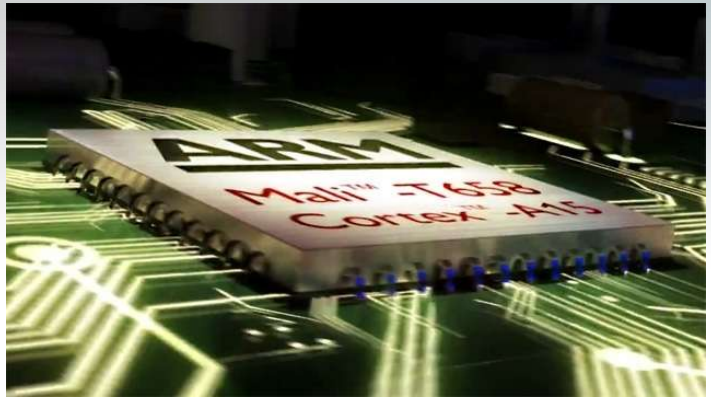
- Introduction to the ARM Mali Graphics Ecosystem
- Performance Analysis with ARM® DS-5™ and Streamline™
- Debugging with Mali™ Graphics Debugger
- Working out Limiting Factor
- Identify the problem and find out how to fix it
- Mali Developer Education
- Q & A

Problem: This is not a desktop

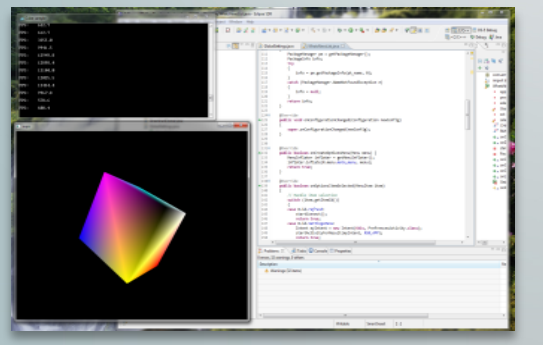
- Mobile apps require special design considerations that aren't always clear and tools to solve increasingly complex systems are limited
  - Animations and games drop frames
  - Networking, display, real time audio and video processing eat battery
  - App won't fit in memory constraints

Analysis

- Fortunately ARM is developing the world class analysis tools and solutions to these problems
- Is my app … ?
  - CPU/GPGPU bound
  - I/O or memory constrained
  - Power efficient
- What can I do to fix it?
  (short of buying everyone who runs my app
  a Quad-core ARM® Cortex™- A15 processor
  & ARM Mali™-T604 processor or Octo phone)



ARM Mali Developer Tools



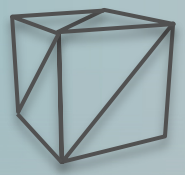
- **Software Development**
  - SDKs for OpenGL® ES & OpenCL™
  - OpenGL ES Emulators
  - Shader Development Studio
  - Shader Library
- **Asset Creation**
  - Texture Compression Tool
  - Asset Conditioning Tool
  - Binary Asset Exporter
- **Performance Analysis**
  - Streamline Performance Analyzer
  - Offline Shader Compiler
  - Mali Graphics Debugger

Mali Developer Tools Flow

Textures

Adobe Photoshop

Texture Compression Tool



Graphics Assets

Autodesk Maya, 3ds Max

Asset Conditioning Tool

Application Code

Eclipse, DS-5 or other IDE

Mali OpenGL ES SDK

Shaders

Shader Library

Shader Development Studio

Offline Shader Compiler

Game

OpenGL ES Emulators

Windows, Linux and Mac

Mali Hardware

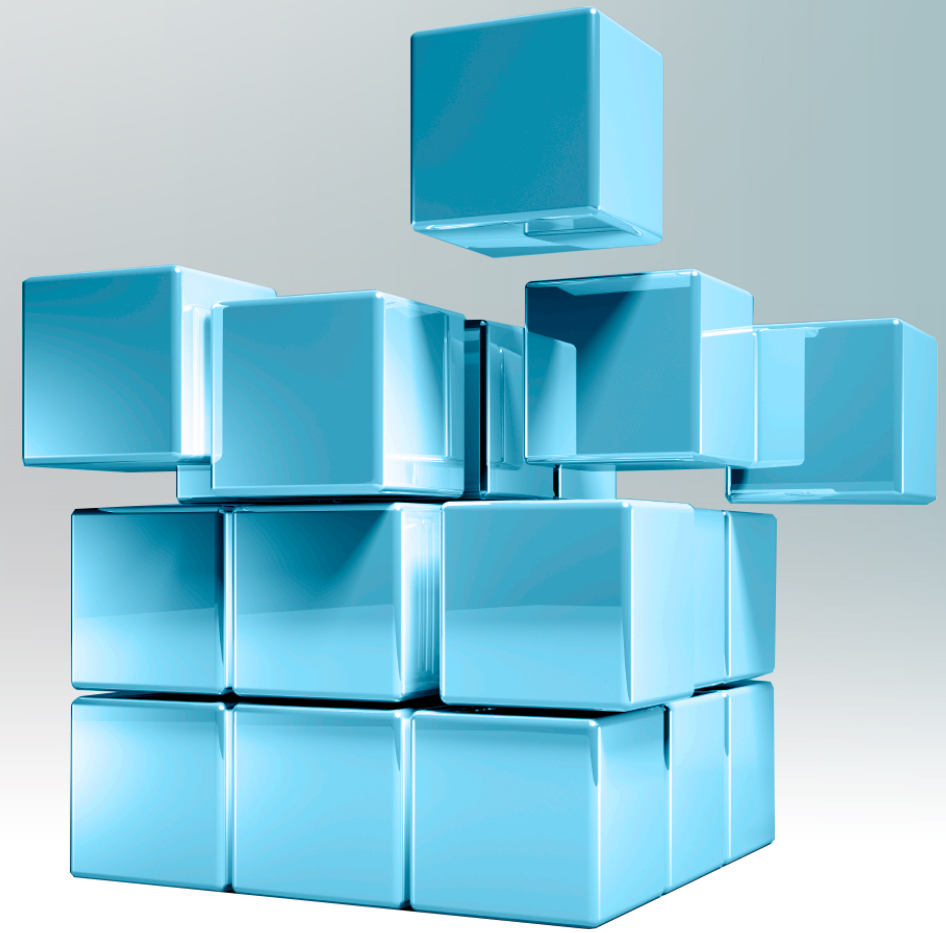
Android and Linux

ARM DS-5™ Streamline

Mali Graphics Debugger

Mali Software Development Kits

- Simplify writing, porting and optimizing OpenGL ES & OpenCL code for Mali GPU based platforms
- Demonstrate key differentiating features to developers and programmers
- **Contents**
  - Environment for quickly developing OpenCL and OpenGL ES applications
  - Tutorials and advice on developing good OpenCL & OpenGL ES code for Mali GPUs
  - Sample code

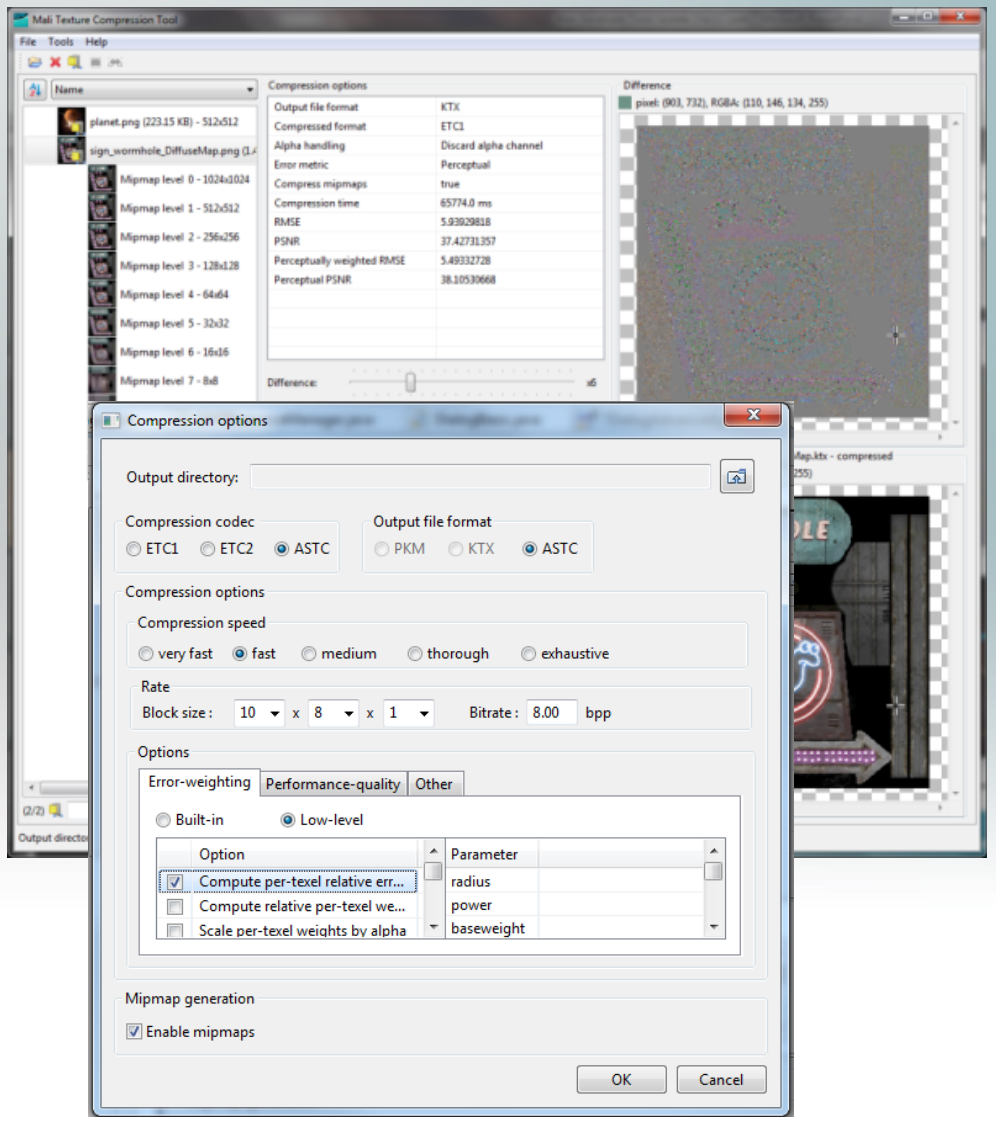


Emulation

- **OpenGL ES 1.1/2.0 Emulator**
  - Khronos Conformant
  - MESA software rendering support
- **OpenGL ES 3.0 Emulator**
  - Khronos Conformance Test Submitted
  - ATSC support
- Includes EGL emulator
- Run OpenGL ES content on desktop systems
  - Easier setup/running/debugging
  - "WYSIWYG"

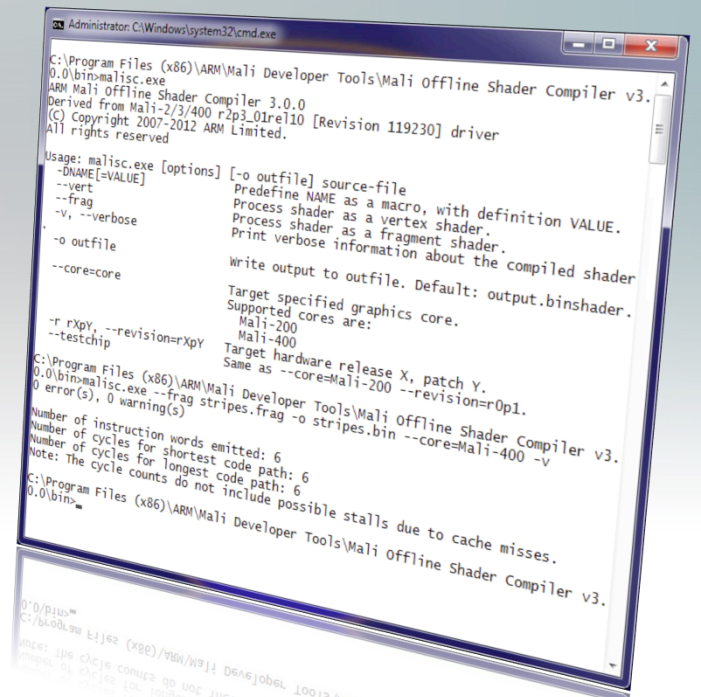
Texture Compression Tool

- ETC1 Texture compression
  - **600x** speed up compared to existing reference encoder
- ETC2 / EAC texture compression
  - Multiple new formats and support for alpha channel
- ASTC Texture Compression
  - LDR and HDR image support
  - Bitrates from 0.89 bits/pixel to 8bits/pixel in fine steps
- Visualization of compressed output
- Reporting of compression statistics
- Automatic Mipmap generation



Offline Shader Compiler

- Compiles shader code written in OpenGL ES Shading Language (ESSL) offline
- Provides verbose shader performance & error messages for optimization and debug
- Support for:
  - Mali-300
  - Mali-400
  - Mali-450
  - Mali-T604
  - Mali-T628
  - Integration with Shader Development Studio
  - Integration with Mali Graphics Debugger



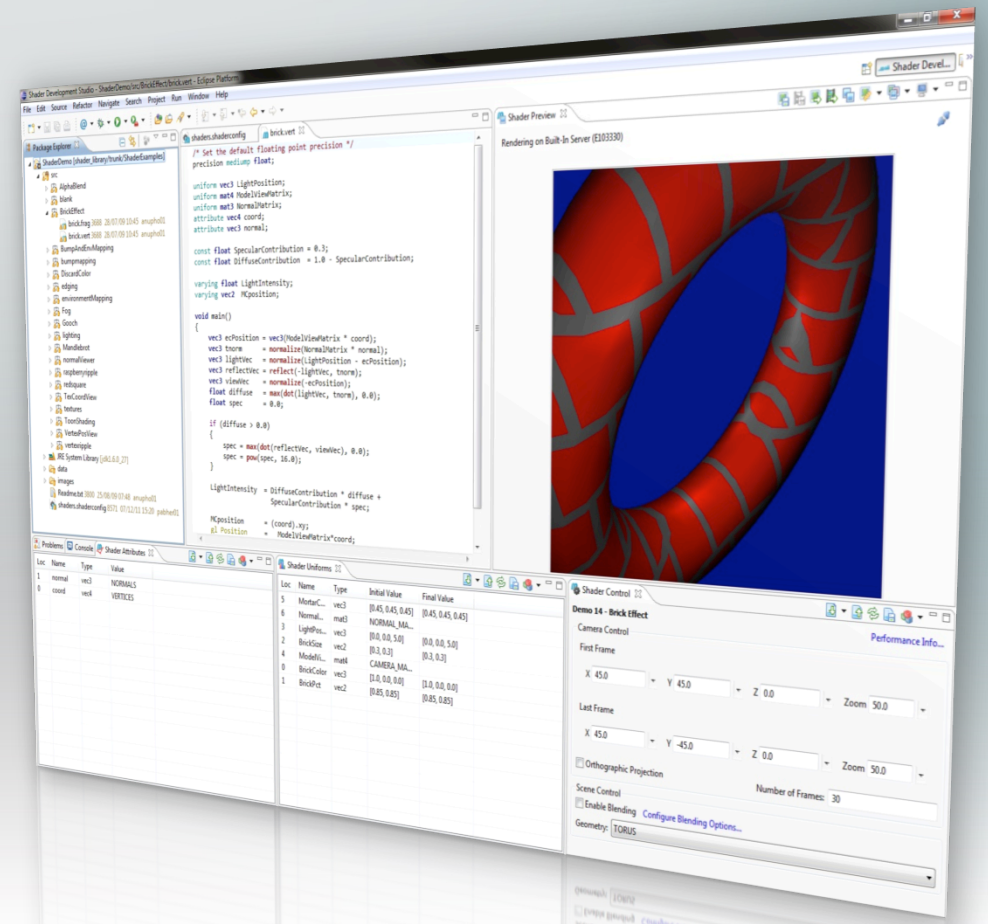
Mali Offline Shader Compiler

- Command-line interface: Easy integration into regression build and test systems
- Offline compilation of GLSL ES vertex & fragment shaders to Mali GPU binary
- Detailed output of shader performance

```
C:\Program Files (x86)\ARM\Mali Developer Tools\Mali Offline Shader Compiler v4.
0.0\bin>malisc.exe -v --frag --core=Mali-T600 "C:\Documents\Presentations\Own\gd
c\Example_FresnelFp.glsles.OLD"
0 error(s), 0 warning(s)

2 work registers used, 1 uniform registers used

Pipelines:                                  A / L / T / Overall
Number of instruction words emitted:        10 + 3 + 3 = 16
Number of cycles for shortest code path:    4.5 / 3 / 3 = 4.5 (A bound)
Number of cycles for longest code path:     4.5 / 3 / 3 = 4.5 (A bound)
Note: The cycle counts do not include possible stalls due to cache misses.
```

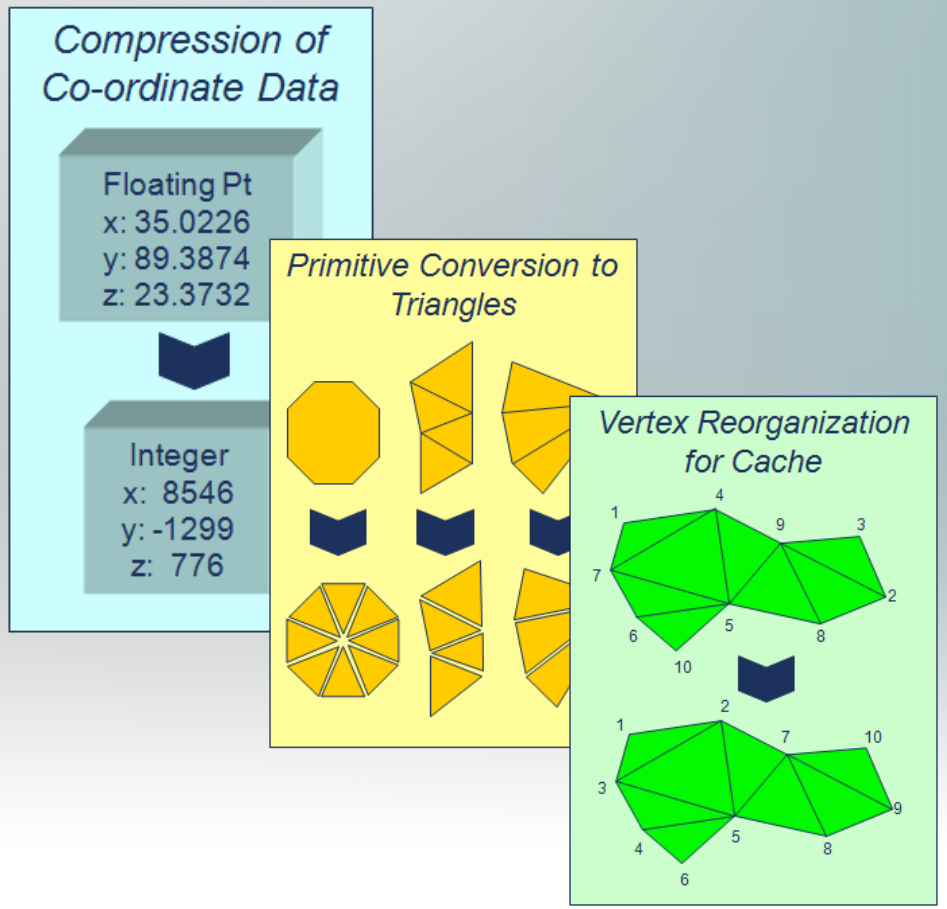
Shader Development

- **Shader Development Studio**
  - Rapid prototyping environment for shader development
  - Extensive Library of shader examples
  - Real-time preview on host and on target



Asset Conditioning Tool

- Optimization of geometry data for Mali GPU-based devices
- Conversion of unsupported primitives to supported types
- Vertex reorganization for efficient cache utilization



Timbuktu Demo

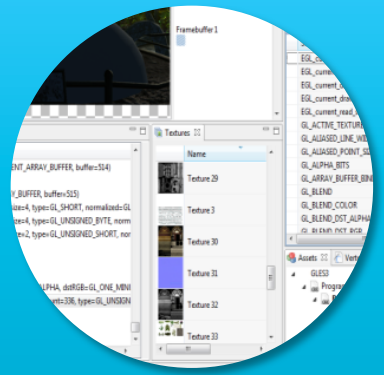
Importance of Analysis & Debug

- Mobile Platforms
  - Expectation of amazing console like graphics and playing experience
  - Screen resolution beyond HD
  - Limited power budget
- Solution
  - Mali GPU 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

Performance Analysis & Debug

DS-5™ Streamline

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



Mali Graphics Debugger

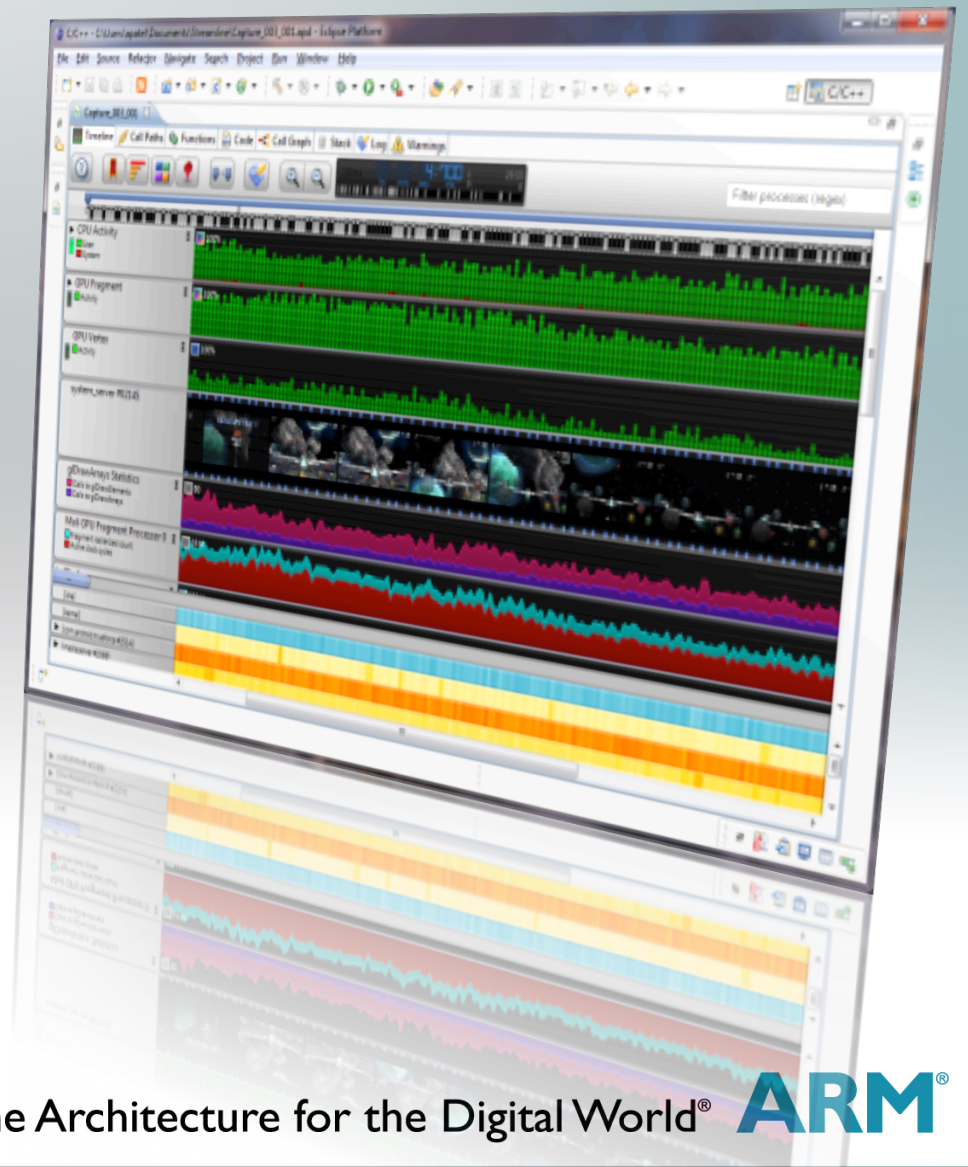
- API Trace & Debug Tool
- Understand graphics and compute issues at the API level
- Debug and improve performance at frame level
- Support for OpenGL ES 1,1, 2.0, 3.0 and OpenCL 1.1

Offline Compilers

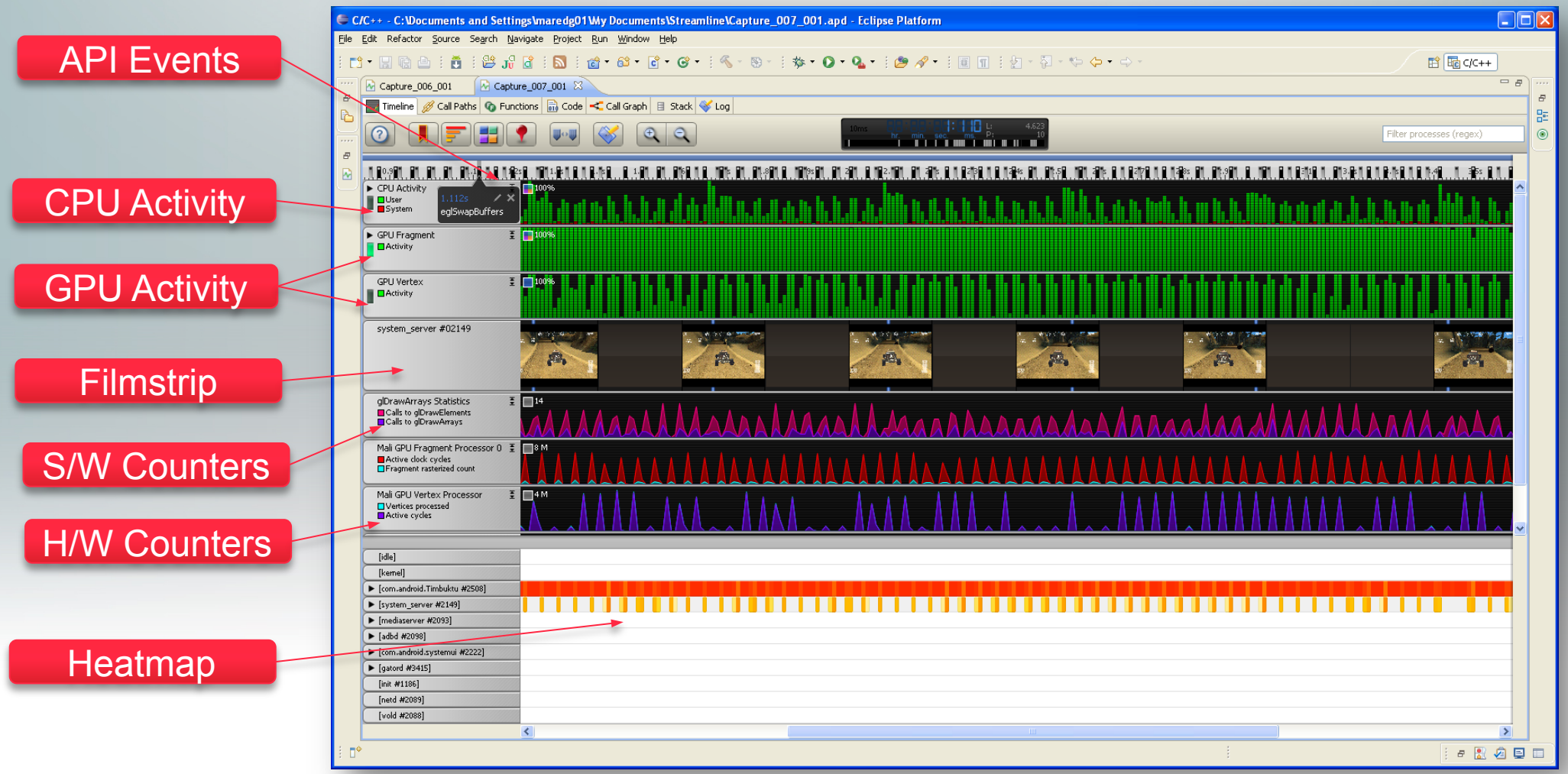
- Understand complexity of GLSL shaders and CL kernels
- Support for Mali-4xx and Mali-T6xx GPU families

Streamline Performance Analyzer

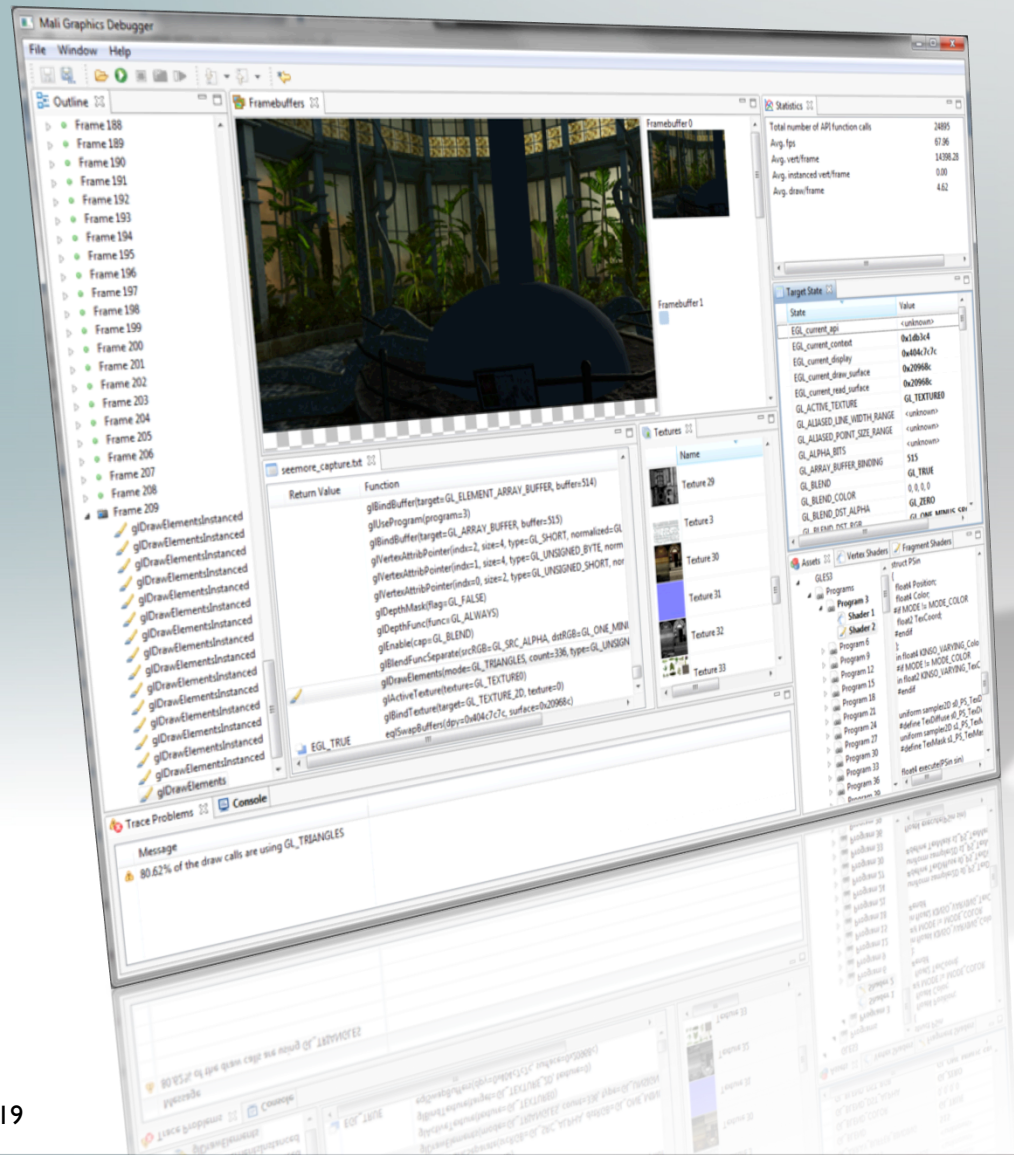
- **System Wide performance Analysis**
  - Simultaneous visibility across ARM Cortex® processors & Mali GPUs
  - Support for graphics and GPU compute performance analysis on Mali-T604
  - Timeline profiling of hardware counters for detailed analysis
  - Software counter support for OpenGL ES 2.0 and OpenCL 1.1
  - Custom counters
  - Per-core/thread/process granularity
  - Frame buffer capture and display
- **ARM DS-5 toolchain** with support for ARM Mali GPUs
- **Optimize performance and power efficiency** of gaming applications across the system



Streamline Performance Analyzer



Mali Graphics Debugger



Mali Graphics Debugger

- Graphics debugging for content developers
- API level tracing
- Understand issues and causes at frame level
- Support for OpenGL® ES 2.0, 3.0, EGL & OpenCL™ 1.1
- Complimentary to DS-5 Streamline

Analyzing and Debugging on Device

API Calls Affect the State

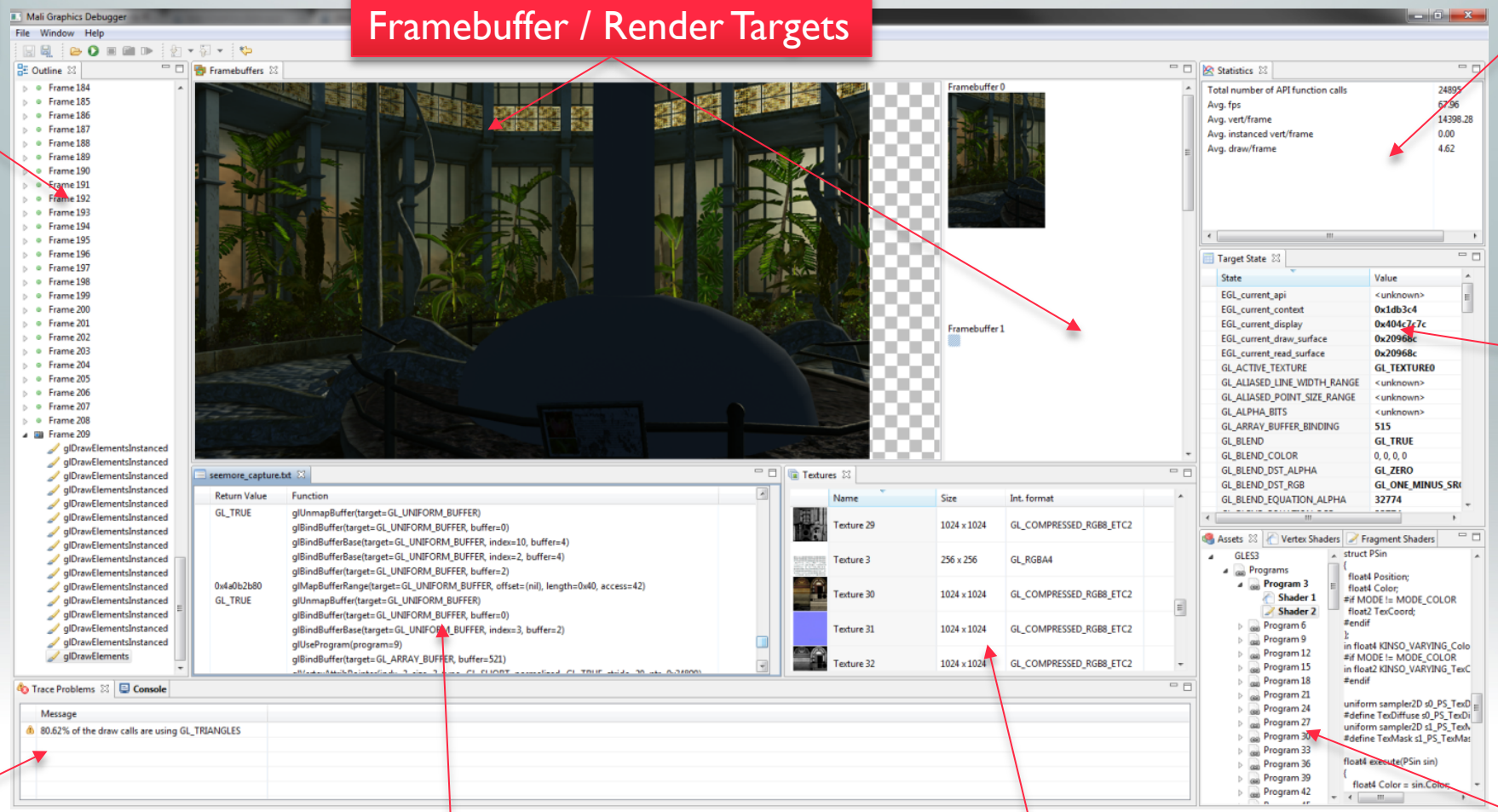
Application

```
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);
...
```

Mali Graphics Debugger



Frame Outline

Framebuffer / Render Targets

Frame Stats

State View

Dynamic Help

API Trace

Textures

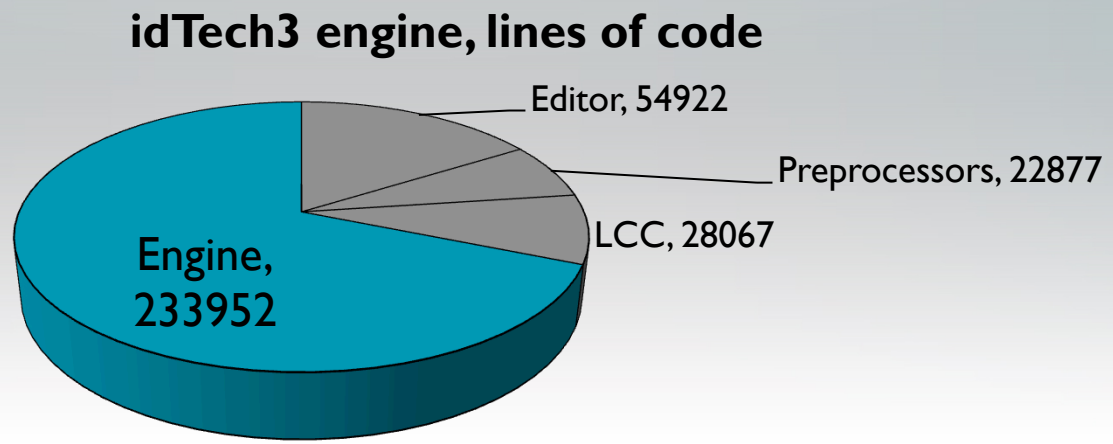
Asset/Shader View

The Architecture for the Digital World®

ARM®

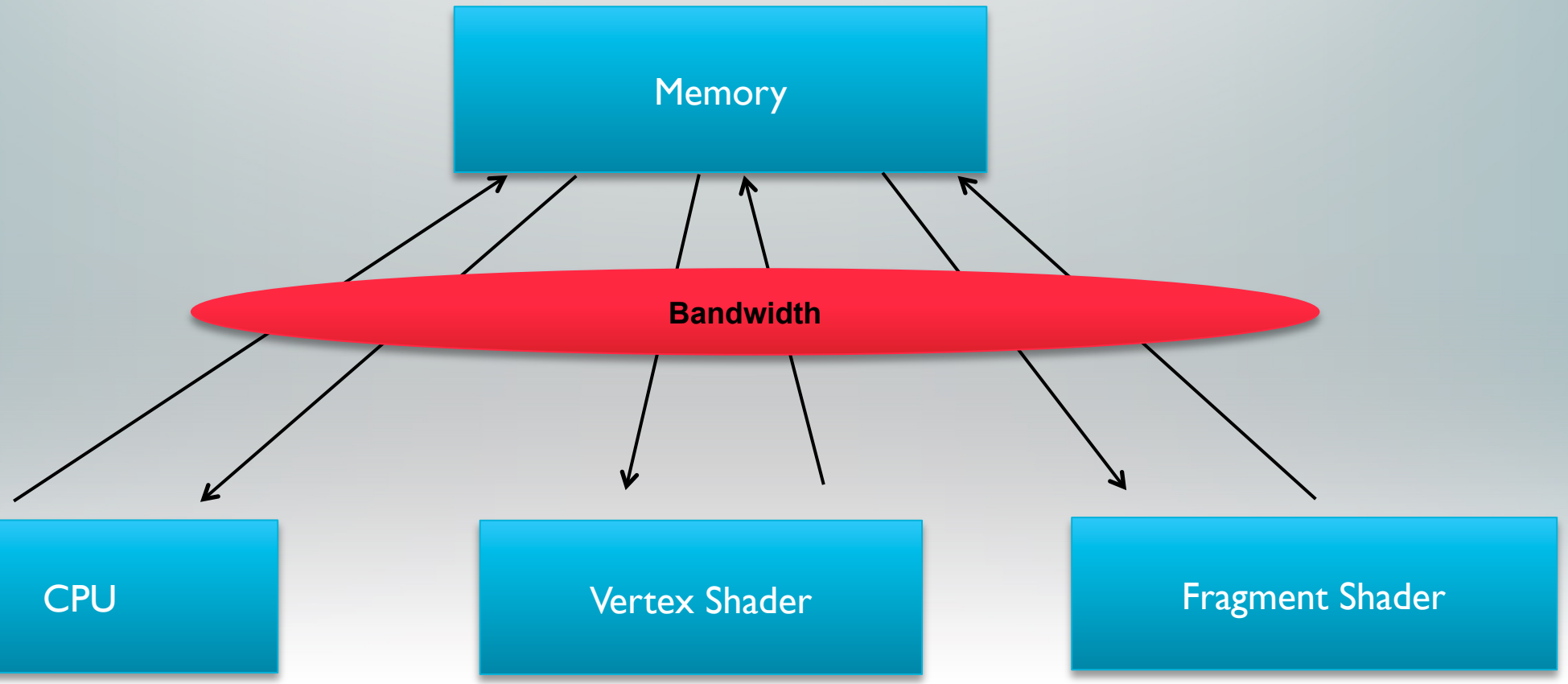
Analyzing 3D Applications

- 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://fabiensanglard.net/quake3 Updated: Feb, 02, 2013.

Bandwidth Vertex Fragment CPU

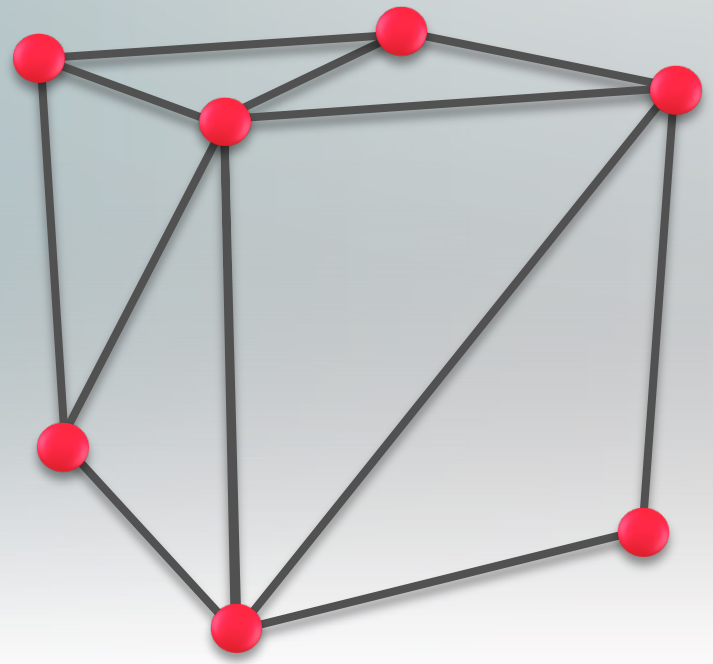


CPU doing too much and stalling GPU?

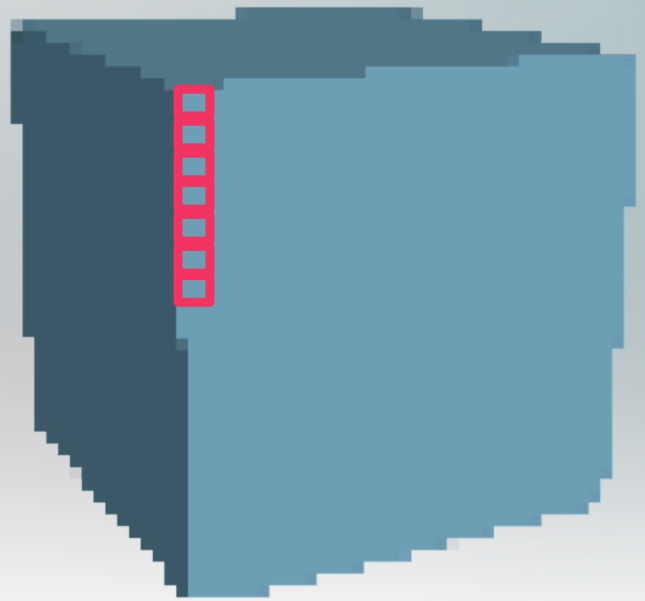
Vertex shader operating on too many vertices stalls fragment shader and CPU?

Fragment shader trying to do a lot of fancy effects stalling CPU and vertex shader?

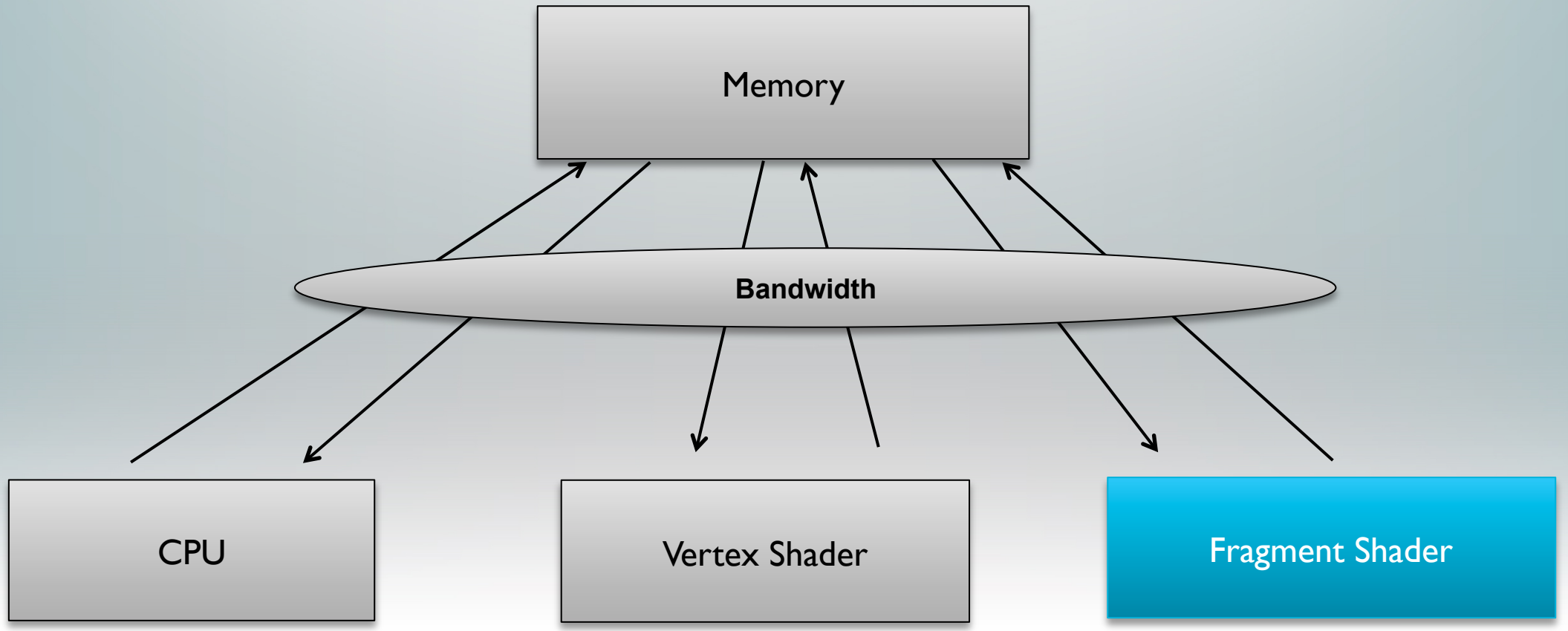
Shaders



Vertex Shader



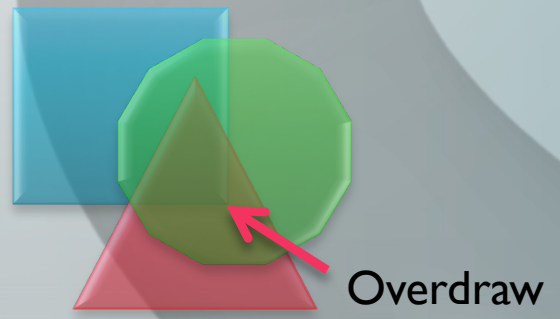
Fragment Shader



Fragment shader trying to do a lot of fancy effects stalling CPU and vertex shader?

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 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

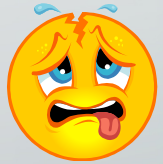
Nexus 10 Native Resolution
- 2560 x 1600 = 4,096,000 pixels

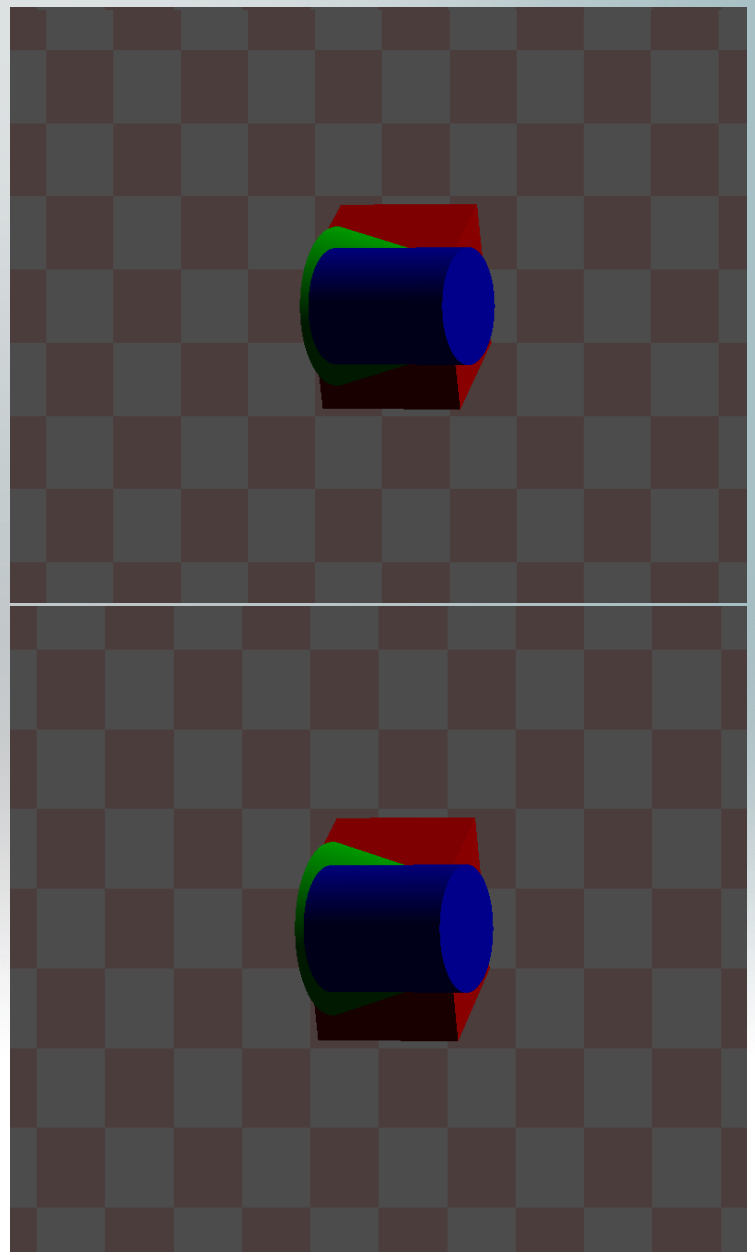
Quad Core GPU 533Mhz
- 520 Cycles per pixel Approx.

Targeting 30 FPS
- 17 Cycles in your shader

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





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

DS-5 Streamline: Fragment Bound

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

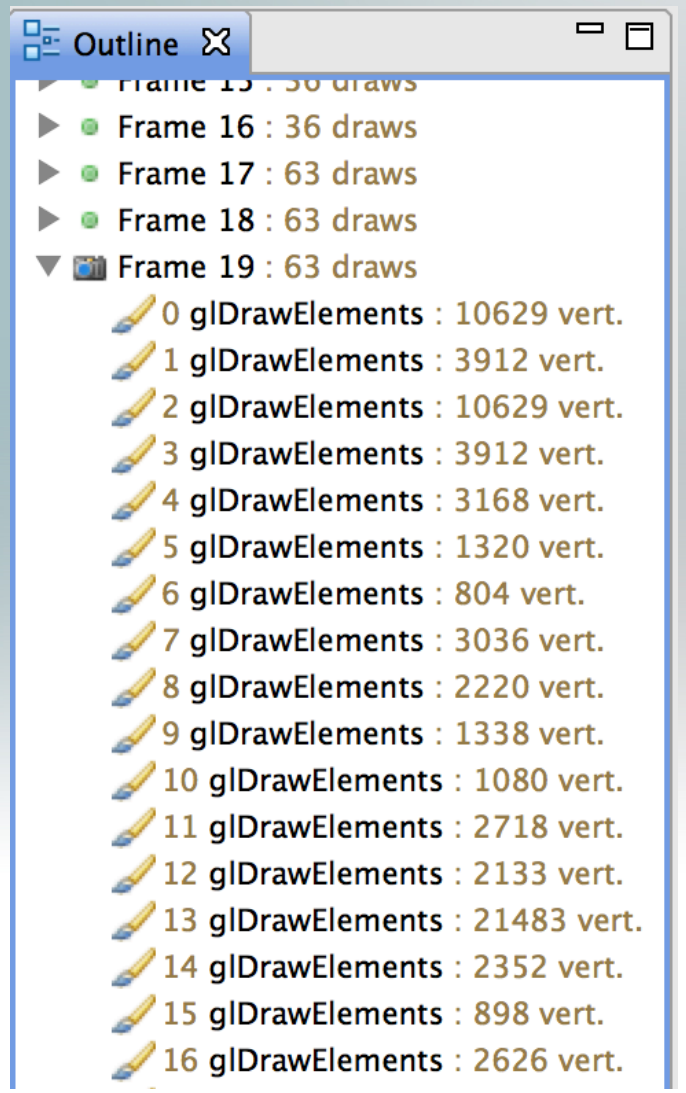
Fragment Percentage = (Job Slot 0 active / Frequency) *100

Fragment Percentage = 84%

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

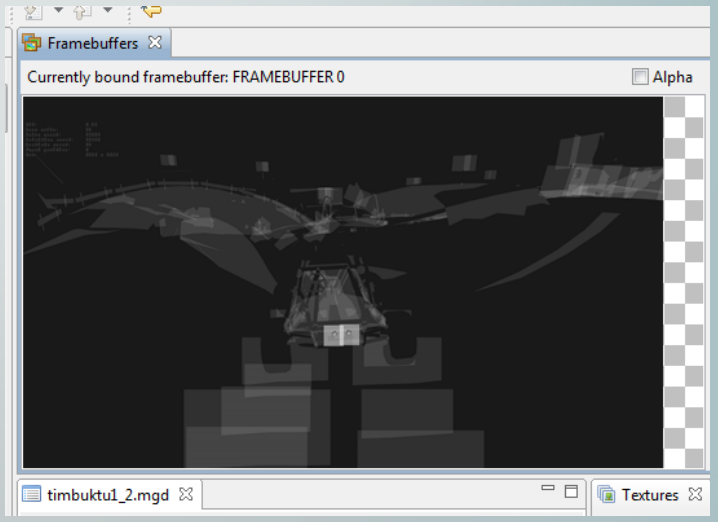
Overdraw = 3.9

Mali Graphics Debugger: inspect the draw calls

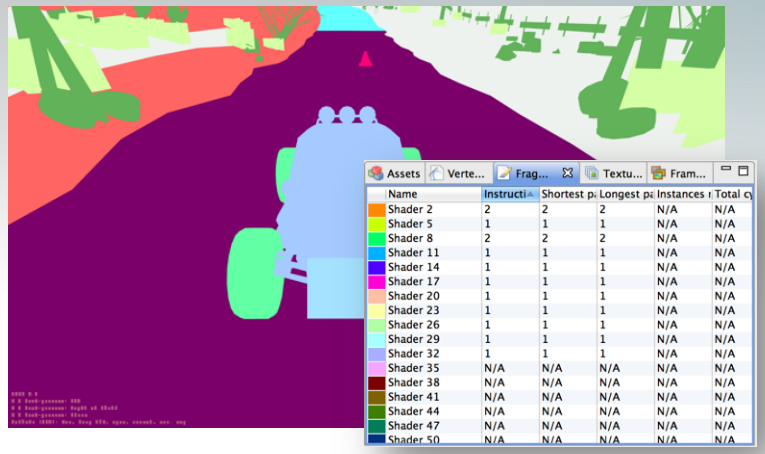


Overdraw and Shader Utilization

- **Overdraw Map**
  - Display render targets with increasing levels of intensity for amount of overdraw
  - High amounts of overdraw displayed in white
- **Shader Utilization**
  - Sorting of Vertex shaders and Fragment shader by utilization
  - Vertex Shader : Shader Cycles x Vertex Count
  - Fragment Shader : Shader Cycles x Fragments affected
  - Visual Map of fragment shader usage
- **Available Nov'13**

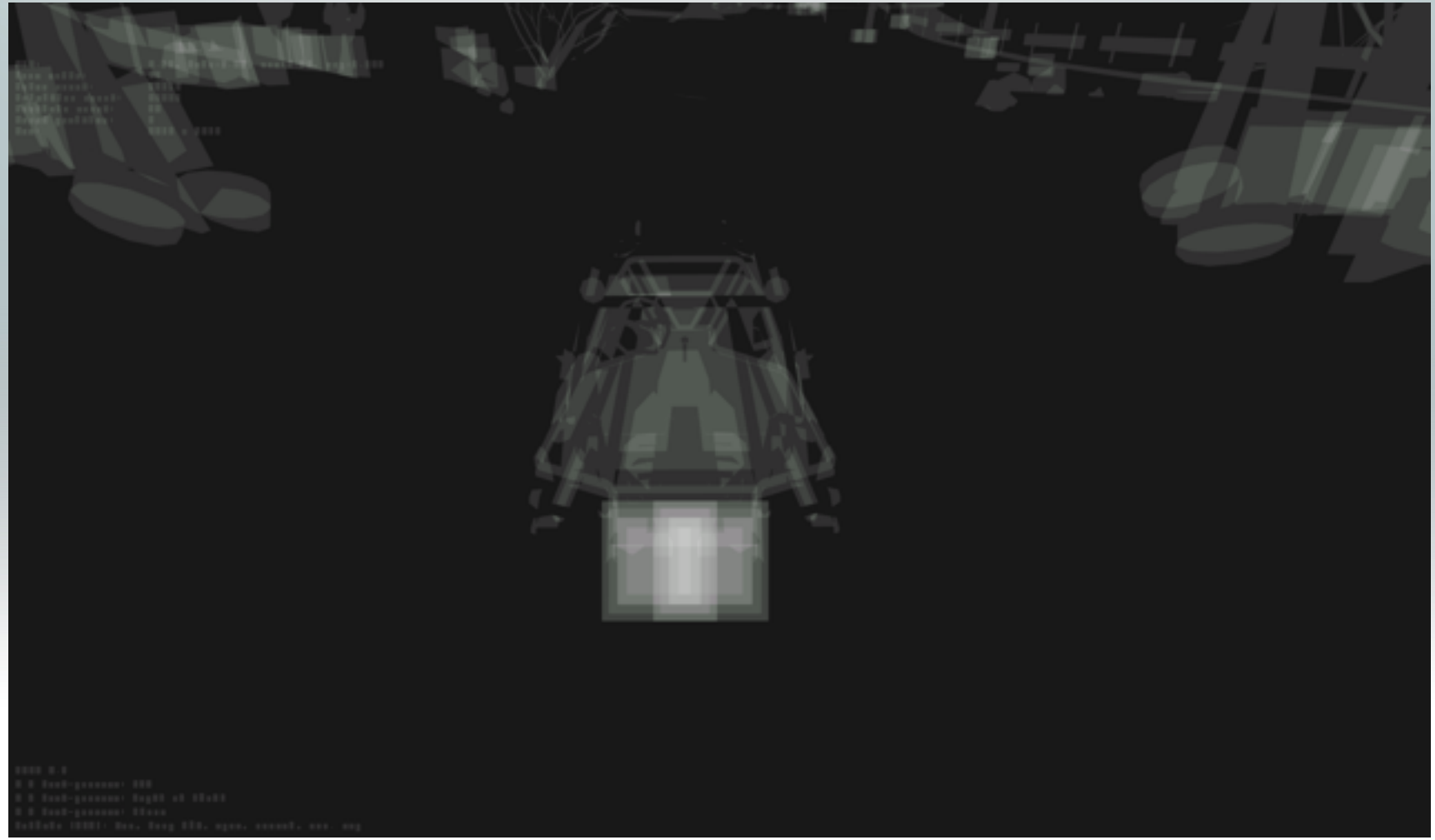


Overdraw Map

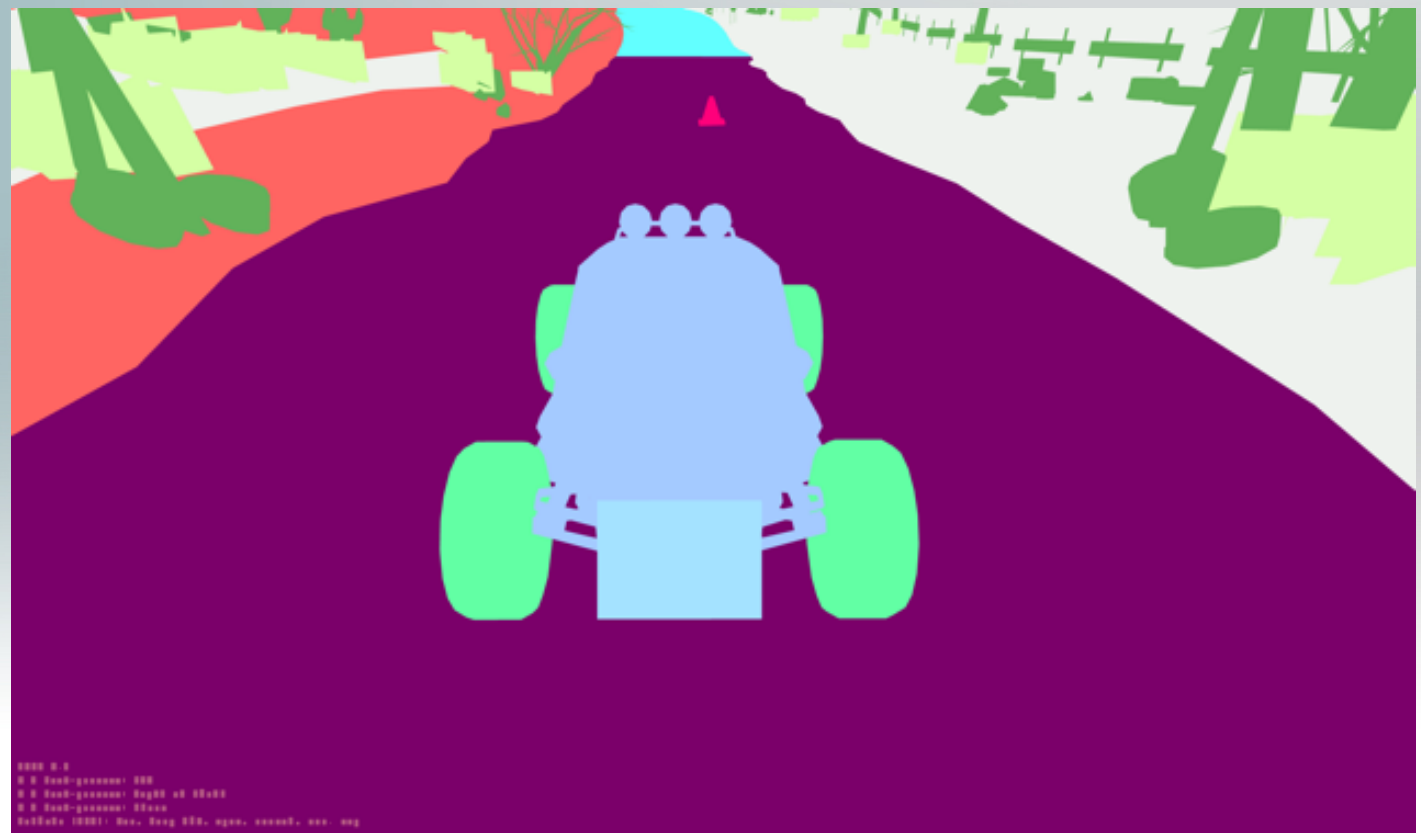


Shader Utilization Map

Overdraw Map



Shader Utilization Map



Assets | Verte... | Frag... | Textu... | Fram...

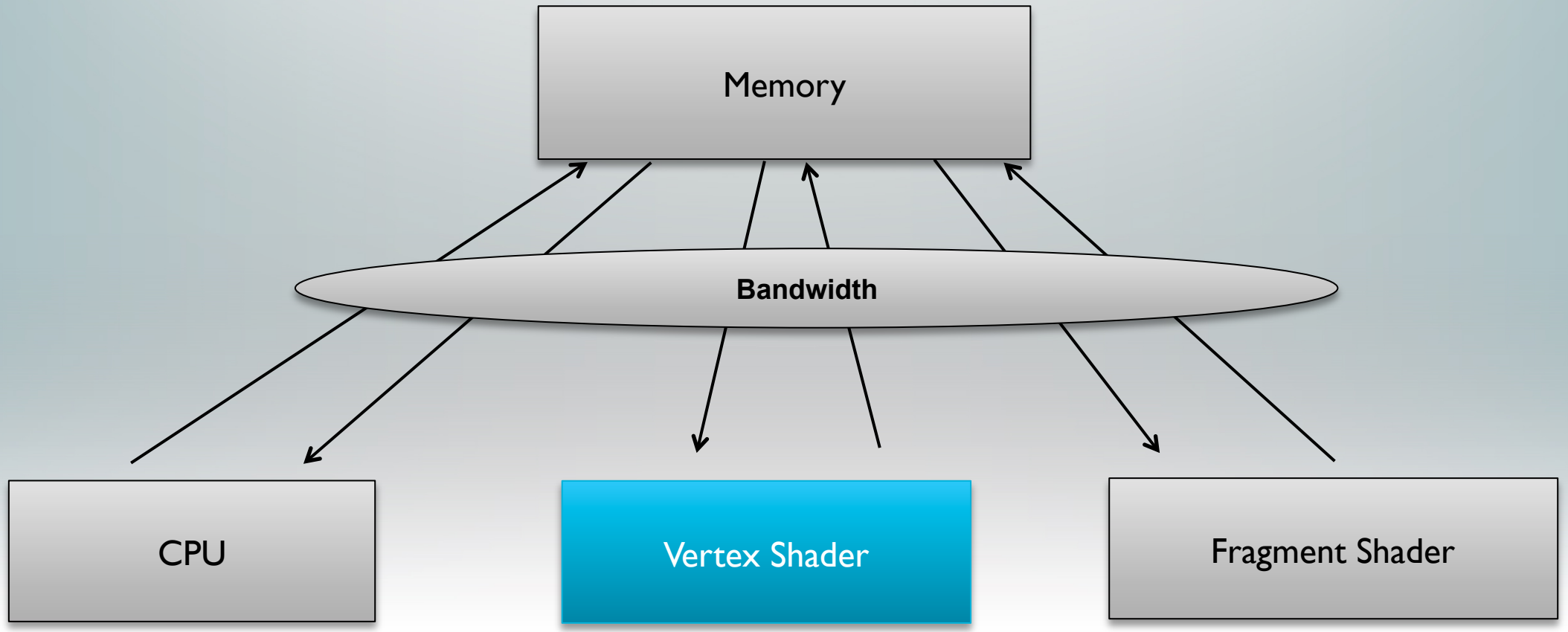
Name	Instructi	Shortest pa	Longest pa	Instances r	Total cy
Shader 2	2	2	2	N/A	N/A
Shader 5	1	1	1	N/A	N/A
Shader 8	2	2	2	N/A	N/A
Shader 11	1	1	1	N/A	N/A
Shader 14	1	1	1	N/A	N/A
Shader 17	1	1	1	N/A	N/A
Shader 20	1	1	1	N/A	N/A
Shader 23	1	1	1	N/A	N/A
Shader 26	1	1	1	N/A	N/A
Shader 29	1	1	1	N/A	N/A
Shader 32	1	1	1	N/A	N/A
Shader 35	N/A	N/A	N/A	N/A	N/A
Shader 38	N/A	N/A	N/A	N/A	N/A
Shader 41	N/A	N/A	N/A	N/A	N/A
Shader 44	N/A	N/A	N/A	N/A	N/A
Shader 47	N/A	N/A	N/A	N/A	N/A
Shader 50	N/A	N/A	N/A	N/A	N/A

Example Shader Optimization

```
11  uniform sampler2D noiseMap;
12  uniform samp
13  uniform samp
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                                          og * noiseScale;
25      final +=
26
27      float fr                                          , fresnelPower);
28
29      vec4 ref
30      vec4 ref
31
32      gl_FragC
33  }
34
```

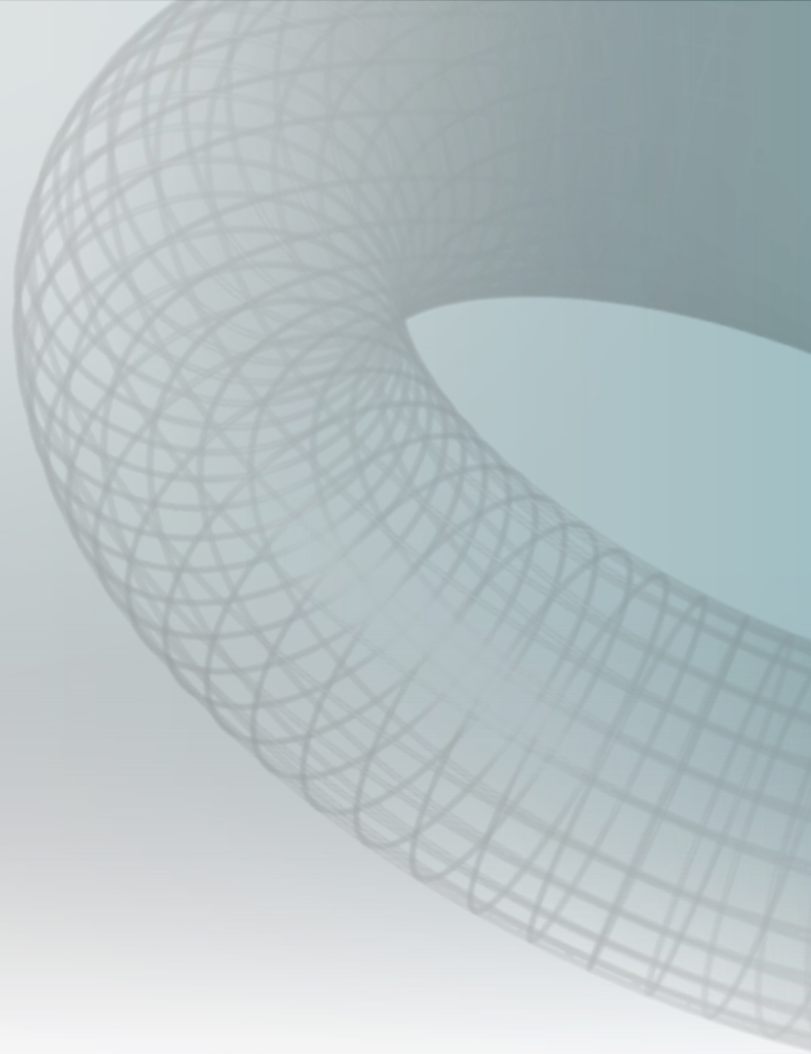
Assets | Vertex Shad... | Fragment Sh... | Textures | Framebuffers

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



Vertex shader operating on too many vertices stalls fragment shader and CPU?

Vertex Bound

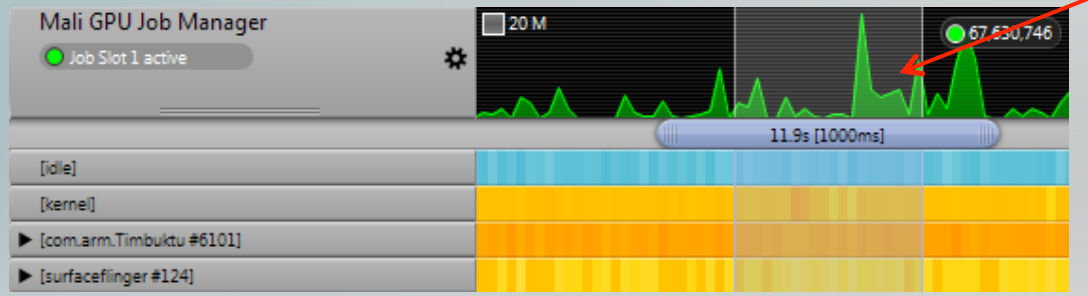


- Too many vertices in geometry
  - Get your artist to remove unnecessary vertices
    - A lot of artists still generate content for high end desktop content
    - Impose some budgeting and limits
  - LOD Switching
    - Only objects near the camera need to be in high detail
    - Objects that are further 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

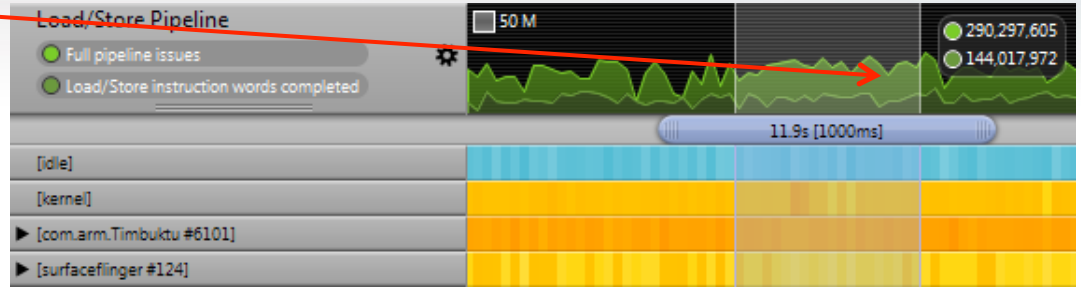
- Involves just 1 counter and the frequency of the GPU
  - Job Slot 1 Active

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



Vertex Percentage = 13%

Load Store CPI = Full Pipeline issues / Load Store Instruction Words Completed

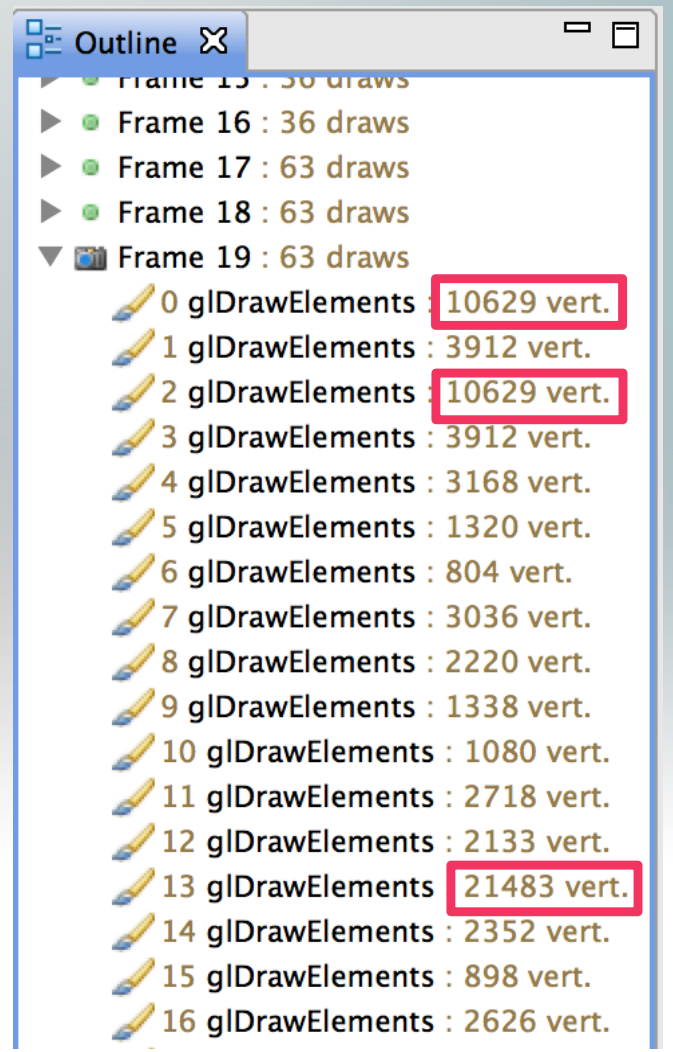


Load Store CPI = 2.02

Mali Graphics Debugger: Vertices Count

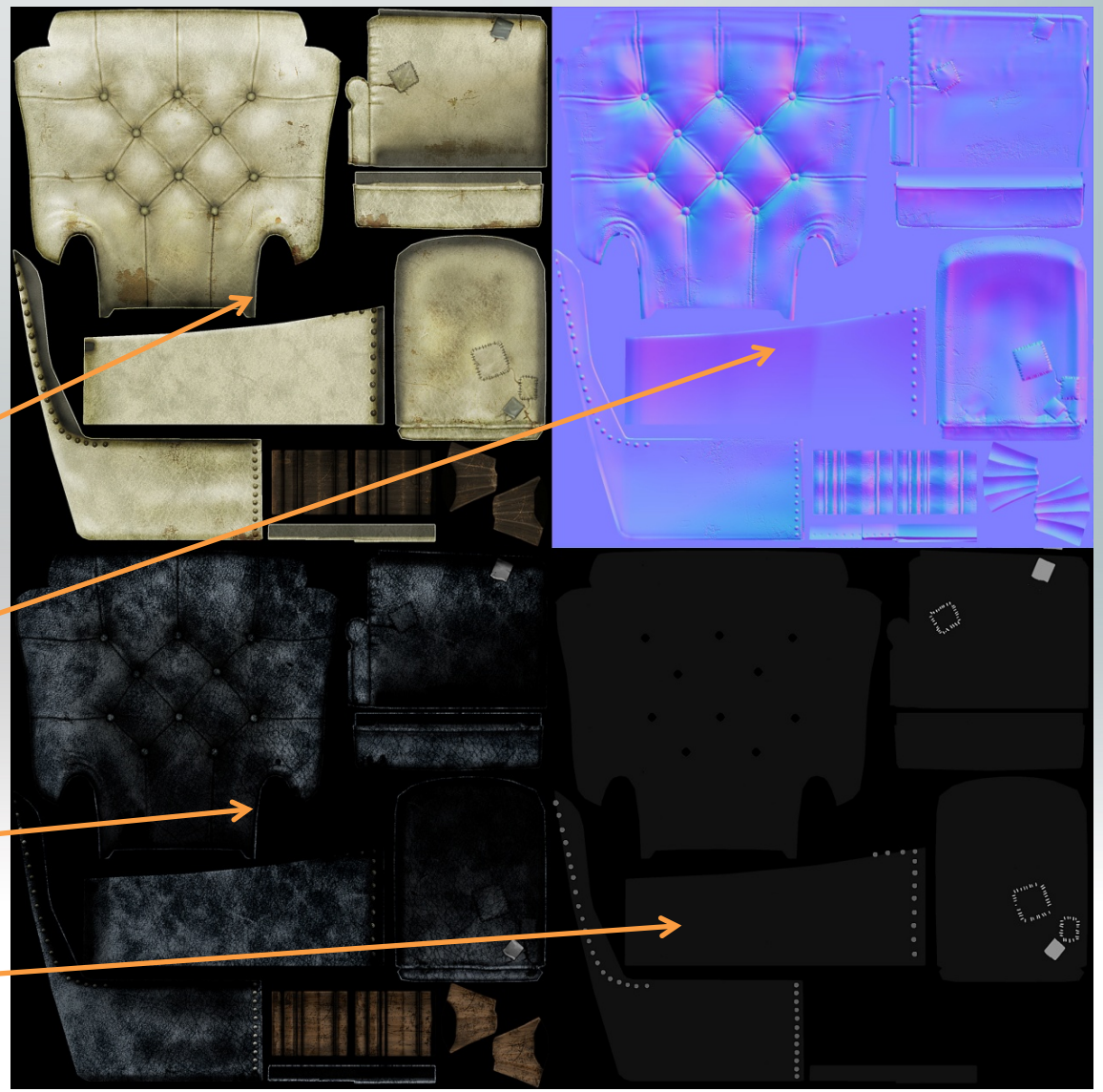
- Analyze the trace in Mali Graphics Debugger
- Find the draw calls with a high number of vertices

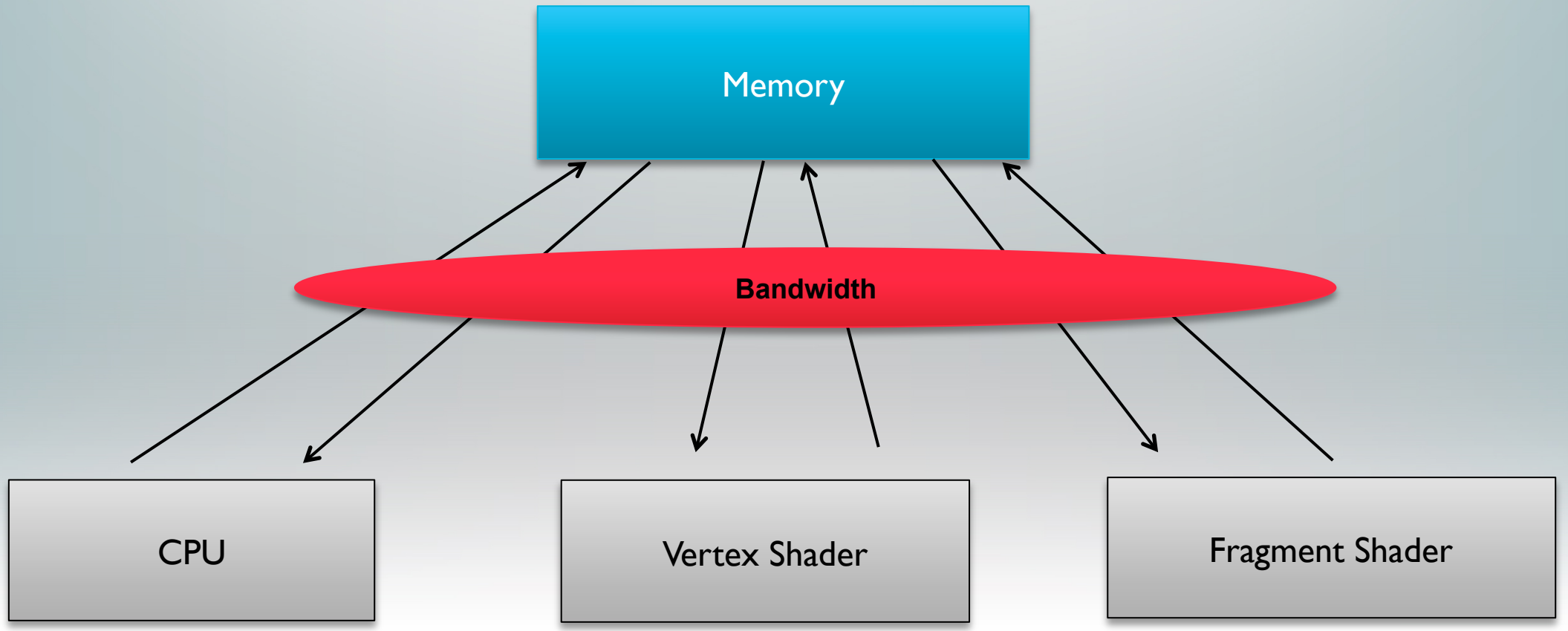
- Shader Statistics
  - Find the vertex shaders with a high number of instructions



Lighting

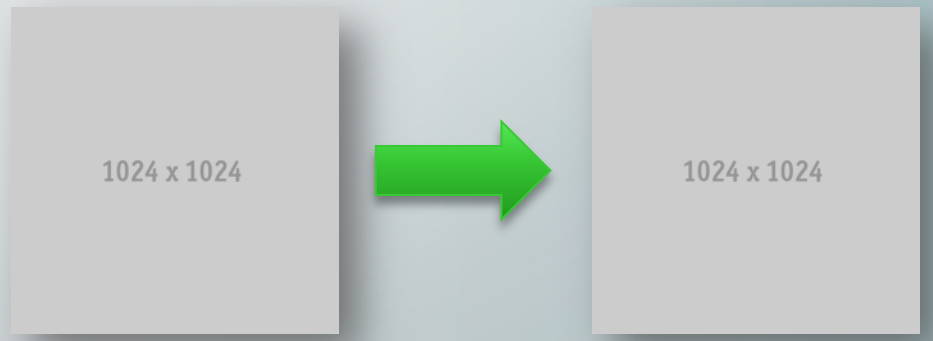
- Often embed light-related information in textures
- Can make low polygon models very realistic!




Memory
Bandwidth
CPU
Vertex Shader
Fragment Shader

Bandwidth Bound

- 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
- Use texture compression
  - The main popular format is ETC Texture Compression
  - ASTC (Adaptive Scalable Texture Compression) < 1 bit per pixel



~ 4 MB
(32 bit per pixel)

~ 0.5 MB
(4 bits per pixel)

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

DS-5 Streamline: Bandwidth Counters

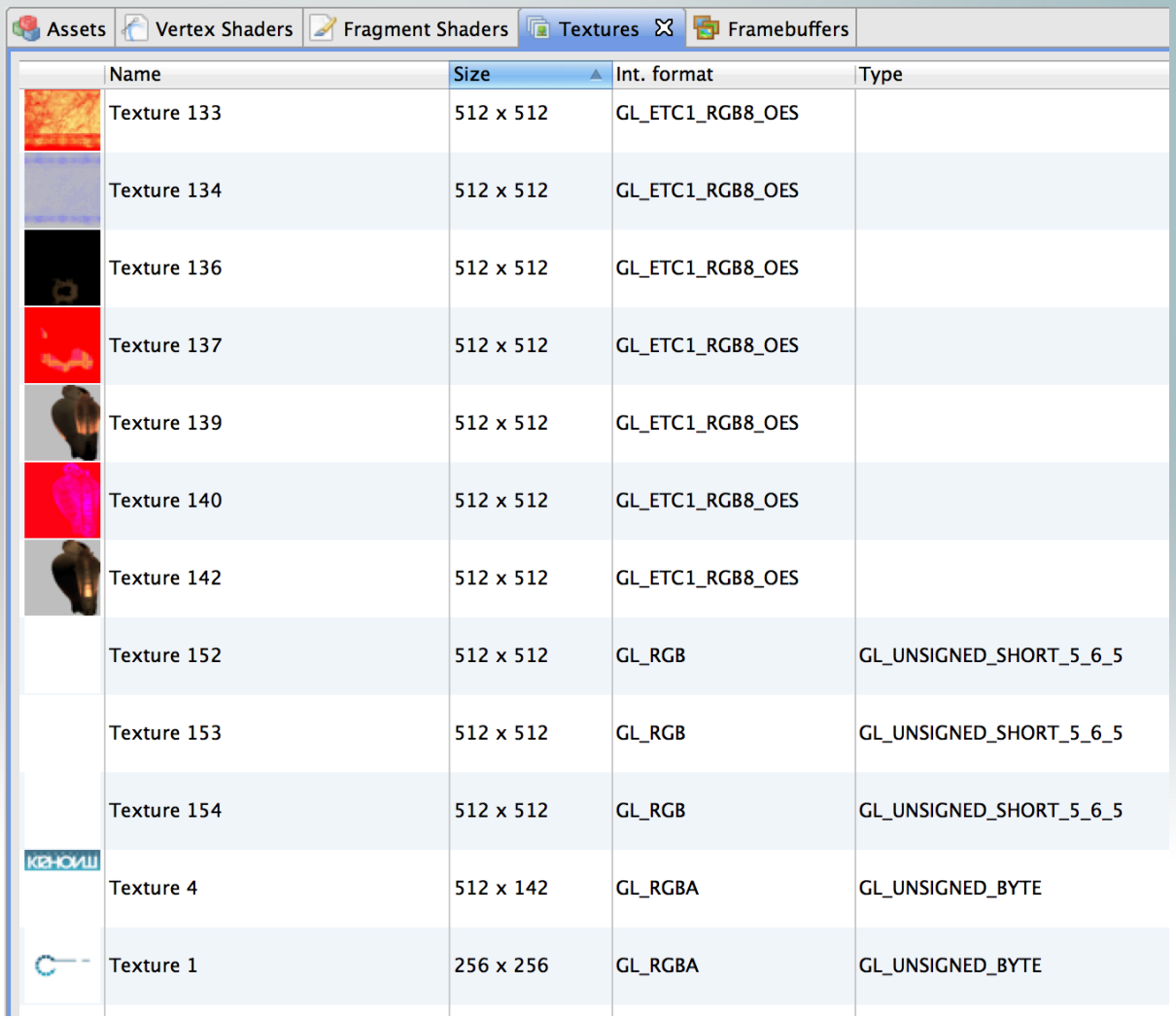
- Involves just 2 Streamline Counters
  - External Bus Read Beats
  - External Bus Write Beats

Bandwidth = 967 MB/S

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

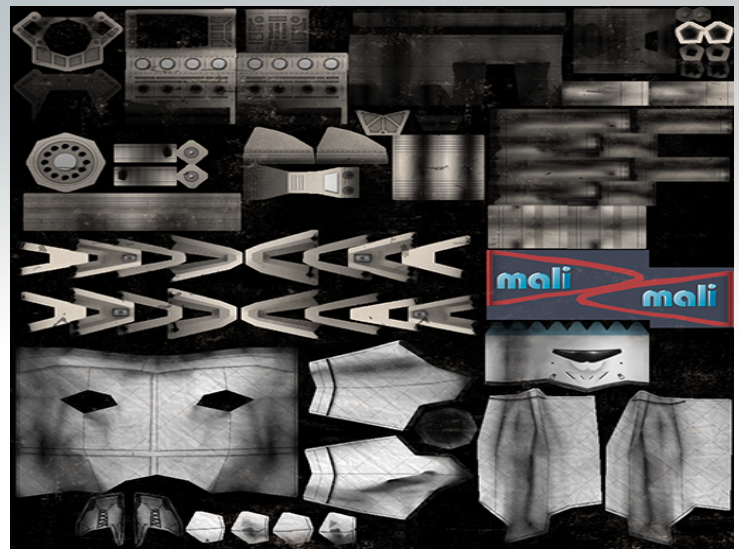
Mali Graphics Debugger: Textures

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



Batching

- Try to combine as many of your draw calls 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....
```

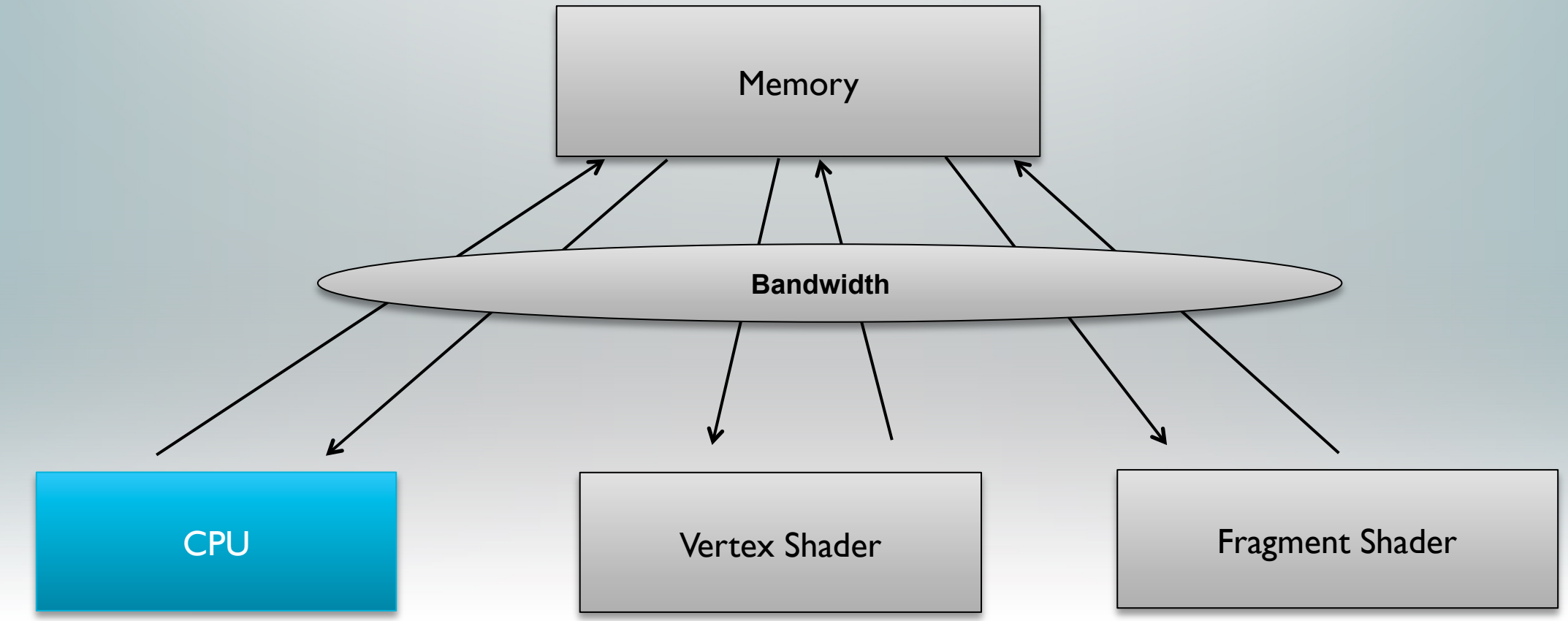
Vertex Buffer Objects

- 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

```
glGenBuffers(1, VertexVBOID);
glBindBuffer(GL_ARRAY_BUFFER, VertexVBOID);
glBufferData(GL_ARRAY_BUFFER, (sizeof(GLFloat)*3)* numVert, &pvertex[0], GL_STATIC_DRAW);
```

```
glVertexAttribPointer(vertexID,3, GL_FLOAT, false, 0, 0)
```

Must pass an offset here instead of a pointer



CPU doing too much and stalling GPU?

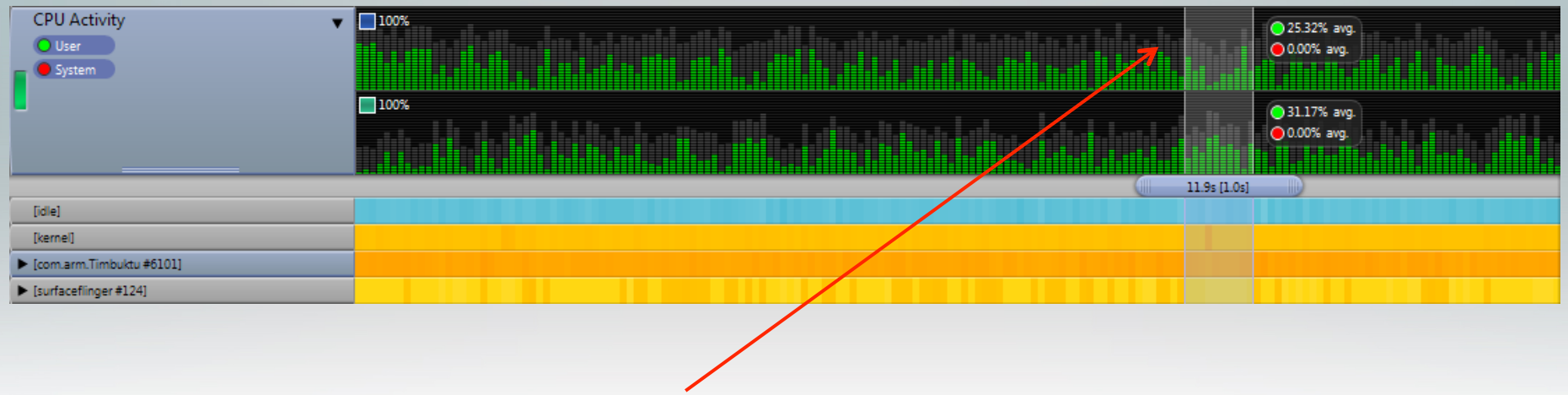
CPU Bound

- 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

Other Streamline Features

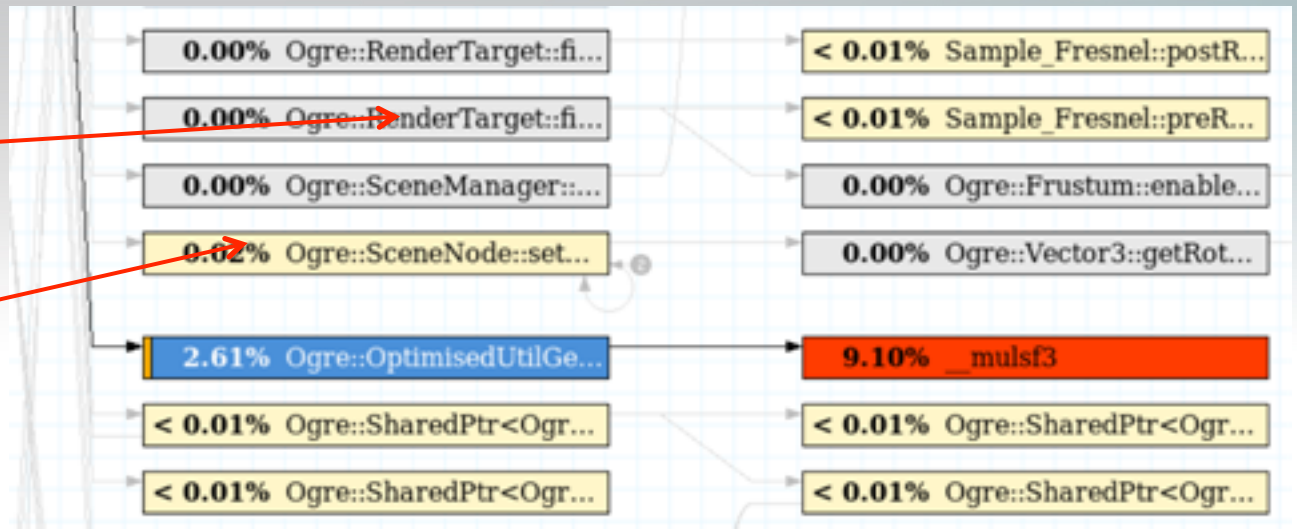
Self	% Self	Instances	Function Name	Location	Image
54,163	71.44%	1	[idle]	<anonymous>	<anonymous>
12,428	16.39%	6	[kernel]	<anonymous>	<anonymous>
1,773	2.34%	3	I_FinishUpdate	i_video.c:472	prboom
1,316	1.74%	10	R_DrawColumn8_PointUV_PointZ	r_drawcolumn.inl:113	prboom
1,021	1.35%	1	R_DrawSpan8_PointUV_PointZ	r_drawspan.inl:80	prboom
823	1.09%	1	R_FlushWhole8	r_drawflush.inl:100	prboom
748	0.99%	7	[libc-2.9.so]	<anonymous>	<anonymous>
502	0.66%	1	R_FlushQuad8	r_drawflush.inl:212	prboom
320	0.42%	3	R_DrawColumn8_PointUV	r_drawcolumn.inl:113	prboom
313	0.41%	7	R_StoreWallRange	r_segs.c:476	prboom

See which functions are the most intensive in your code

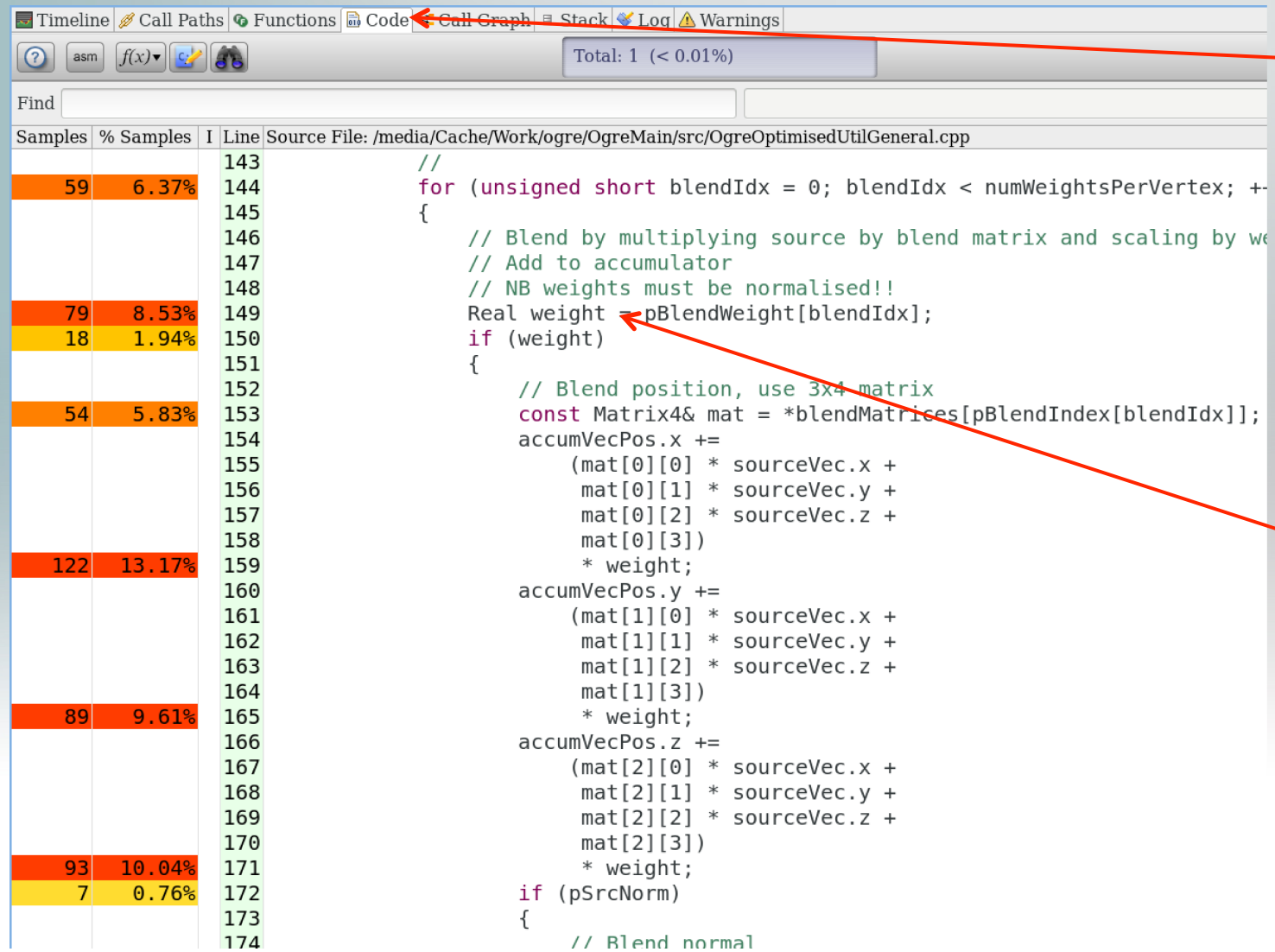
Supply symbols for your code to get more detailed information

Look at the call graph of your application to enable you to follow your program flow

Even when the function calls itself



Other Streamline Features 2



Switch to code view so you can see in your code where all the time is spent

Useful to see how efficient your algorithms are

Mali Offline Shader Compiler

- Command-line interface: Easy integration into regression build and test systems
- Offline compilation of GLSL ES vertex & fragment shaders to Mali GPU binary
- Detailed output of shader performance

```
C:\Program Files (x86)\ARM\Mali Developer Tools\Mali Offline Shader Compiler v4.
0.0\bin>malisc.exe -v --frag --core=Mali-T600 "C:\Documents\Presentations\Own\gd
c\Example_FresnelFp.glsles.OLD"
0 error(s), 0 warning(s)

2 work registers used, 1 uniform registers used

Pipelines:                                  A / L / T / Overall
Number of instruction words emitted:     10 + 3 + 3 = 16
Number of cycles for shortest code path: 4.5 / 3 / 3 = 4.5 (A bound)
Number of cycles for longest code path:  4.5 / 3 / 3 = 4.5 (A bound)
Note: The cycle counts do not include possible stalls due to cache misses.
```

Example Output of Compiler

```
// Fragment program for distorting a texture using a 3D noise texture
void main()
{
    // Do the tex projection manually so we can distort _after_
    vec2 final = projectionCoord.xy / projectionCoord.w;

    // Noise
    vec3 noiseNormal = (texture2D(noiseMap, (noiseCoord.xy / 5.0)).rgb - 0.5).rbg * noiseScale;
    final += noiseNormal.xz;

    // Fresnel
    //normal = normalize(normal + noiseNormal.xz);
    float fresnel = fresnelBias + fresnelScale * pow(1.0 + dot(eyeDir, oNormal), fresnelPower);

    // Reflection / refraction
    vec4 reflectionColour = texture2D(reflectMap, final);
    vec4 refractionColour = texture2D(refractMap, final) + tintColour;

    // Final colour
    gl_FragColor = mix(refractionColour, reflectionColour, fresnel);
}
```

```
C:\Program Files (x86)\ARM\Mali Developer Tools\Mali Offline Shader Compiler v4.
0.0\bin>malisc.exe -v --frag --core=Mali-T600 "C:\Documents\Presentations\Own\gd
c\Example_FresnelFp.glsles.New"
0 error(s), 0 warning(s)

2 work registers used, 2 uniform registers used

Pipelines:                                  A / L / T / Overall
Number of instruction words emitted:        7 + 3 + 3 = 13
Number of cycles for shortest code path:    3 / 4 / 3 =  4 (L bound)
Number of cycles for longest code path:     3 / 4 / 3 =  4 (L bound)
Note: The cycle counts do not include possible stalls due to cache misses.
```

Visit Us at Developer Events

malideveloper.arm.com

Enable developers to interact at different levels

Learn about Mali

- Introduction to Mali, latest news, latest devices…

Develop for Mali

- All the dev tools and resources…

Engage with Mali

- Promotion of partners, registration for developer program, events..

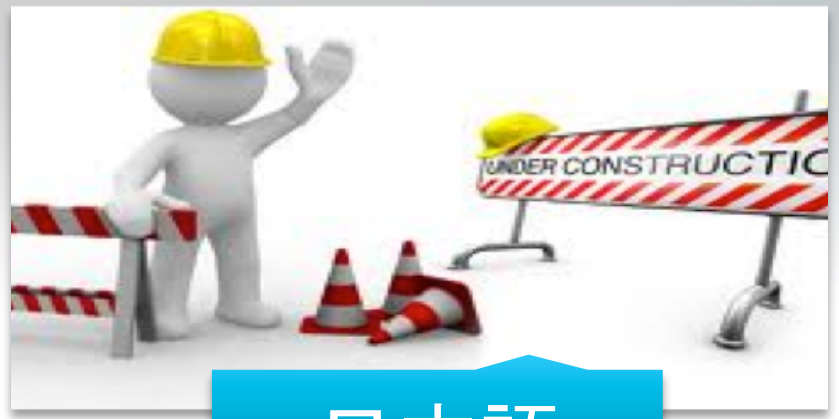
Lots of opportunities to promote Mali partner activities

malideveloper.arm.com

ENGLISH

中文

한국어



日本語

Thank you

malideveloper.arm.com

How to setup MGD

Architecture of the Mali Graphics Debugger - Android

- Android (4.2+) uses a monolithic library which includes GLES and EGL
- The suffix of the library to load is written in /system/lib/egl/egl.cfg
  - Usually it's "`0 1 mali`"
  - We have to change it to "`0 0 mgd`"
- libGLES_mgd.so will dynamically load libGLES_mali.so

Install Steps (Android Nexus 10)

1. Copy Interceptor and Daemon to writable area

```
adb push libGLES_mgd.so /sdcard/
adb push mgddaemon /sdcard/
```

2. Gain A

```
adb s
su
mount
```

Revert to factory GL ES configuration

```
echo "0 0 mali" > /system/lib/egl/egl.cfg
```

3. Reloca

```
cp /sdcard/libGLES_mgd.so /system/lib/egl/
cp /sdcard/mgddaemon /system/bin/
chmod 777 /system/bin/mgddaemon
```

4. Modify EGL configuration to point to interceptor

```
echo "0 0 mgd" > /system/lib/egl/egl.cfg
```

Starting Capture

1. Forward TCP/IP Port 5002 to device

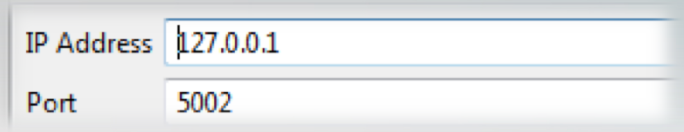
```
adb forward tcp:5002 tcp:5002
```

2. Start MGD Daemon

```
adb shell /system/bin/mgddaemon
```

3. Launch Mali Graphics Debugger Tool

4. Set target IP and Port (File > Set target IP…)



5. Connect to target and start tracing