

Analyzing Performance of Mobile Games

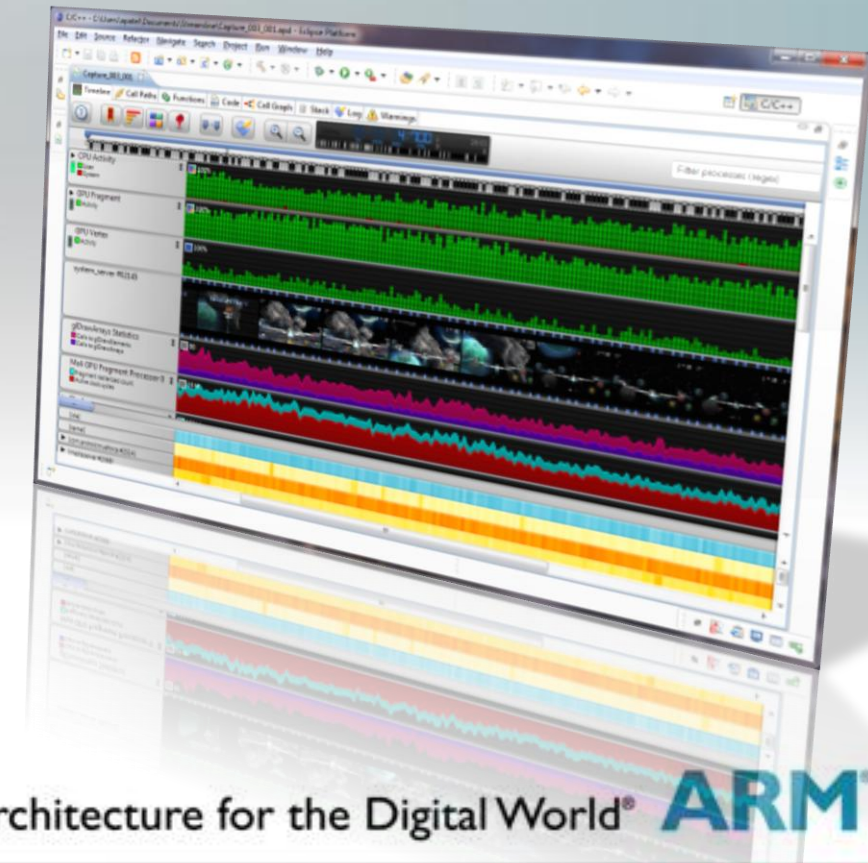
Stephen Barton
Software Engineer, Media Processing Division

Agenda

- Introduction to Streamline and Performance Capture
- Working out Limiting Factor
 - Fragment Bound
 - Vertex Bound
 - Bandwidth Bound
 - CPU Bound
- Other Streamline Features
- Offline Shader Compiler
- Techniques
- Summary

ARM DS-5 Toolchain & Streamline

- **ARM® DS-5™ toolchain** with support for ARM Mali™ GPUs
- **System wide performance analysis**
Simultaneous visibility across ARM Cortex™ processors + Mali GPUs
- **Technology leadership** with first available system level tool in mobile
- **Optimize performance and power efficiency** of gaming applications across the system



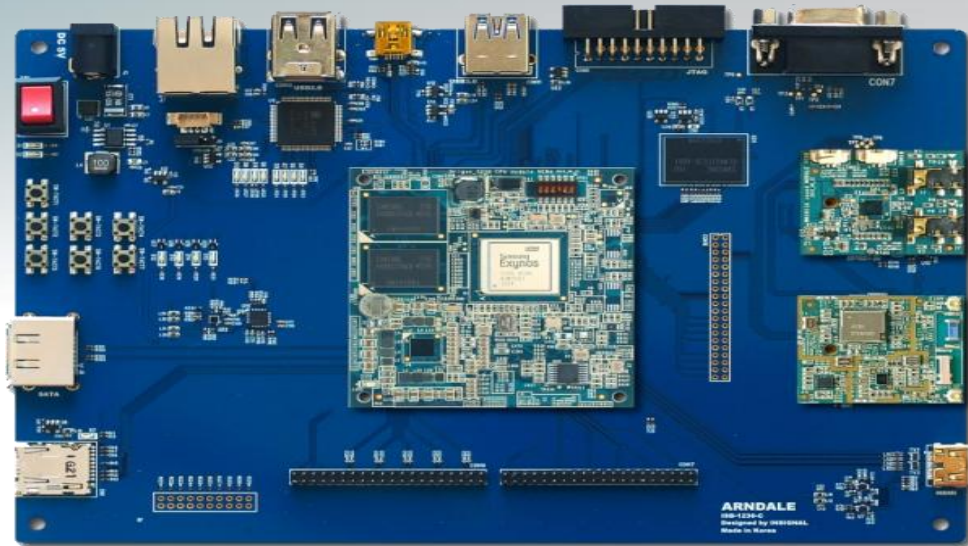
DS-5 Streamline Performance Analyzer

- Best –in-class suite of software development tools for all ARM processors
- Eclipse-based IDE provides code development, debug, performance analysis and compatibility with 3rd party plug-ins
- Thousands of developers using DS-5 toolchain today
- **System Wide performance Analysis**
 - Support for graphics and GPU compute performance analysis on Mali-T604/Mali-T658
 - Timeline profiling of hardware counters for detailed analysis
 - Software counter support for OpenGL[®] ES 2.0 and OpenCL[™] 1.1
 - Per-core/thread/process granularity
 - Frame buffer capture and display

Nexus 10 Out of Box

■ Nexus 10

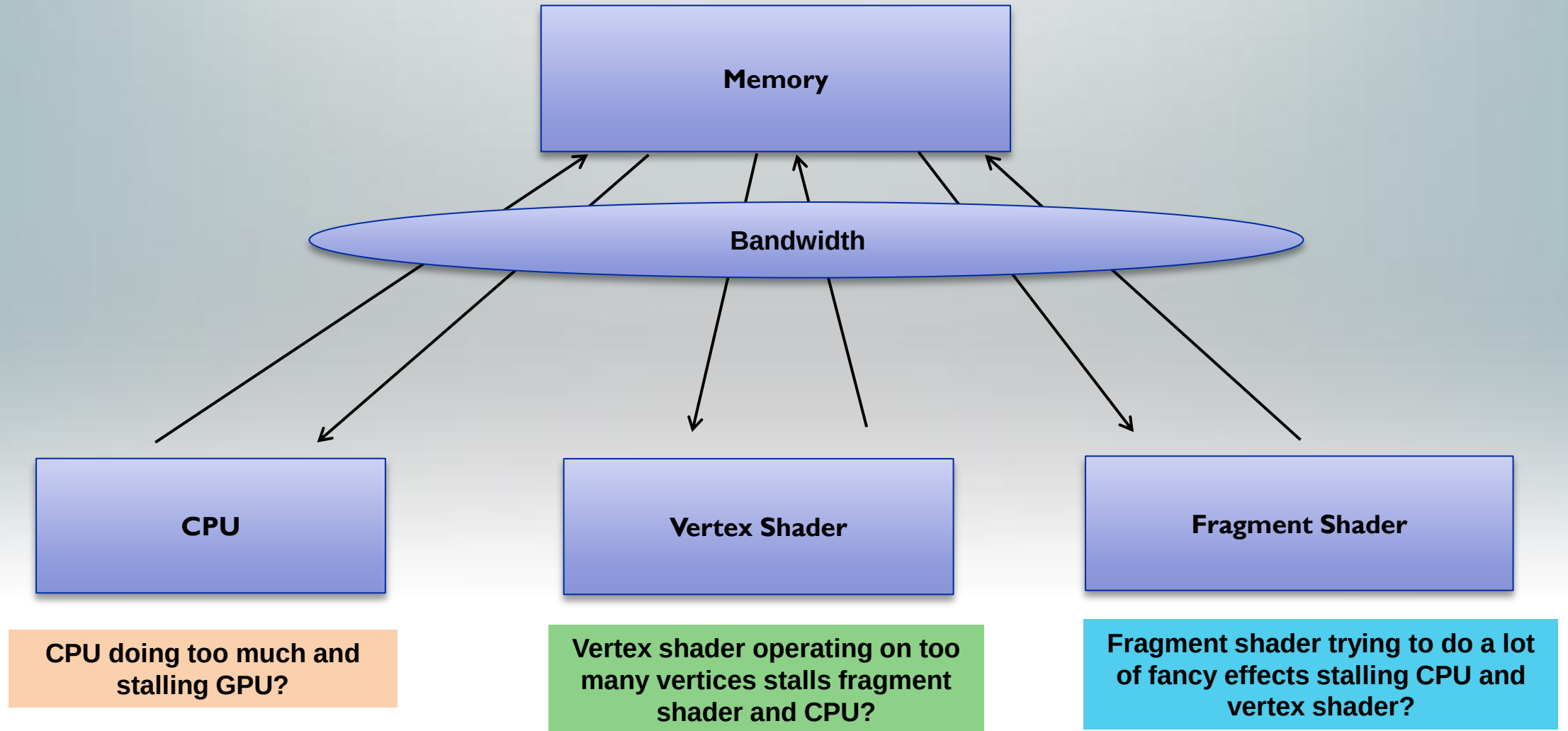
- Dual-core Cortex-A15 processor @1.7GHz
- Mali-T604 @533MHz
- 4 MP resolution (2560×1600)
- Stock Android™ 4.2.1 with DS-5 support



■ Arndale

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

Bandwidth Vertex Fragment CPU



Fragment Bound

- **Overdraw**

- This is when you draw to each pixel on the screen more than once
- Drawing your objects front to back instead of back to front reduces overdraw
- Also limiting the amount of transparency in the scene can help

- **Resolution too high or too many effects or cycles in shader**

- Every effect that you add to your scene. Every light 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 = $2560 \times 1600 = 4,096,000$ pixels

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

Targeting 30 FPS = 17 Cycles in your shader

Fragment Bound Streamline

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

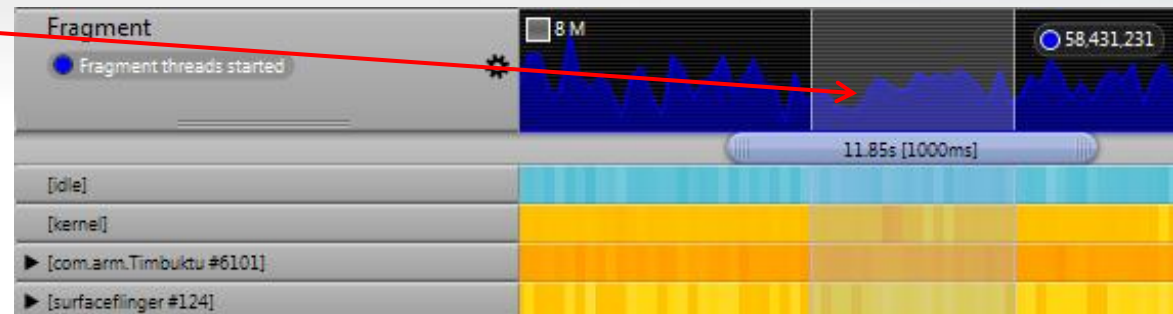


Fragment Percentage = (Job Slot 0 active / Frequency) * 100

Fragment Percentage = 84%

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

Overdraw = 3.9

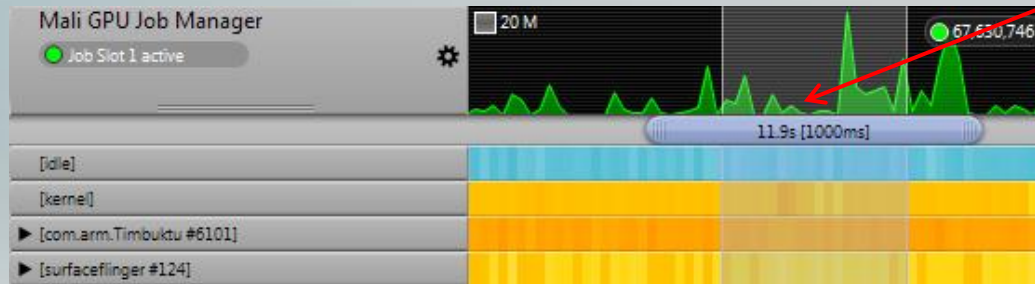


Vertex Bound

- 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
 - Use LOD Switching
 - Only objects that take up a lot of screen space 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

Vertex Bound Streamline

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

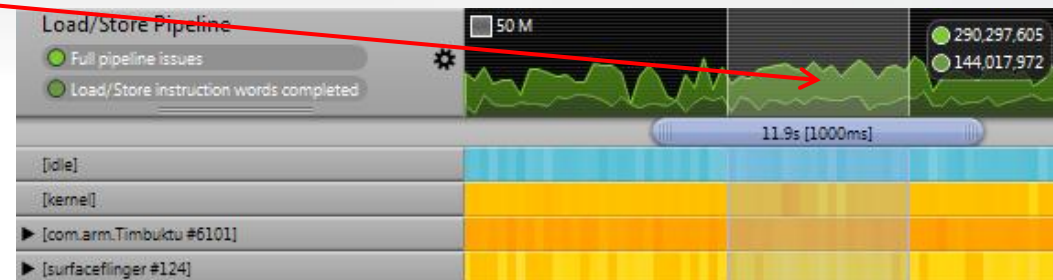


$$\text{Vertex Percentage} = (\text{Job Slot I active} / \text{Frequency}) * 100$$

Vertex Percentage = 13%

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

Load Store CPI = 2.02



Bandwidth Bound

- When creating embedded graphics applications. Bandwidth is a scarce resource
 - A typical embedded device can handle $\approx$ 5.0 Gigabytes a second of bandwidth
 - A typical desktop GPU can do in excess of 100 Gigabytes a second
- One way to reduce bandwidth is to use texture compression
 - The main popular format is ETC Texture Compression
 - This can help reduce your 32 bits per pixel texture into a 4 bits per pixel texture
 - Mali Texture Compression Tool can help convert your textures for you
- Another way to reduce bandwidth is to use 16 bit textures instead of 32
 - You won't often notice the difference

Bandwidth Bound Streamline

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

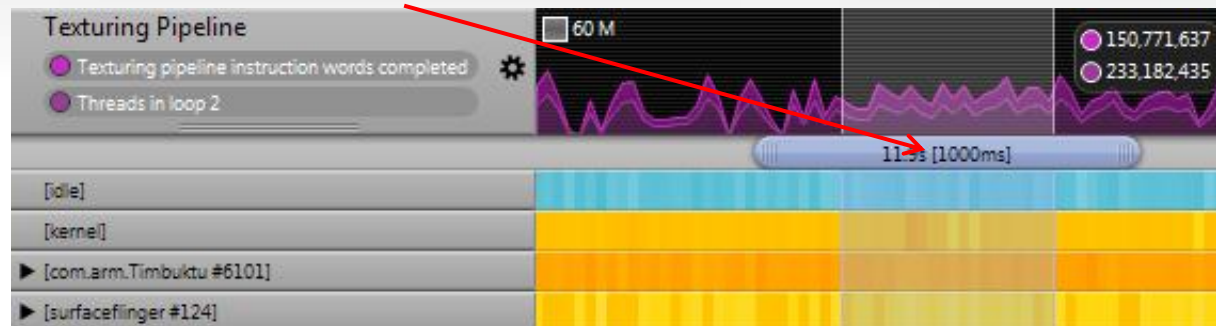


Bandwidth in Bytes = (External Bus Read Beats + External Bus Write Beats) * Bus Width

Bandwidth = 967 MB/S

Texture Pipeline CPI = Threads in Loop 2 / Texturing Pipeline instruction words completed

Texture Pipeline CPI =
1.55

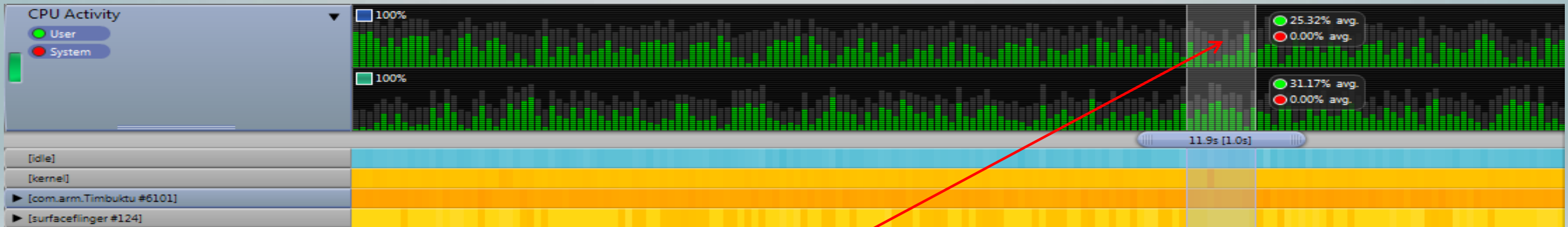


CPU Bound

- 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

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
- Available on malideveloper.arm.com

```
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\Ppresentations\Own\gd
c\Example_FresnelFp.glsls.OLD"
0 error(s), 0 warning(s)

2 work registers used, 1 uniform registers used

Pipelines:
Number of instruction words emitted:      A / L / T / Overall
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).rgb * 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.
```

Other Streamline Features

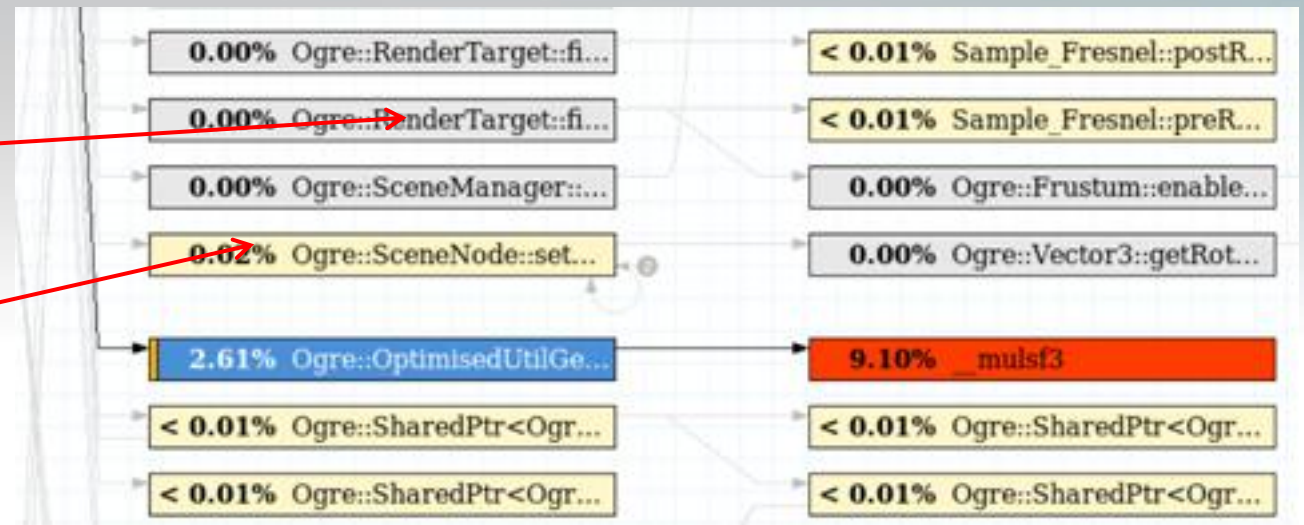
Self ▼	% Self	Instances	Function Name ▲	Location	Image
54,163	71.44%	1	[idle]	<anonymous>	<anonymous>
12,428	16.39%	6	[kernel]	<anonymous>	<anonymous>
1,773	2.34%	3	I_FinishUpdate	i_video.c:472	prboom
1,316	1.74%	10	R_DrawColumn8_PointUV_PointZ	r_drawcolumn.inl:113	prboom
1,021	1.35%	1	R_DrawSpan8_PointUV_PointZ	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

Timeline Call Paths Functions Code Call Graph Stack Log Warnings

asm f(x) Total: 1 (< 0.01%)

Find

Samples	% Samples	I	Line
59	6.37%		143
			144
			145
			146
			147
79	8.53%		148
18	1.94%		149
			150
			151
			152
54	5.83%		153
			154
			155
			156
			157
			158
122	13.17%		159
			160
			161
			162
			163
			164
89	9.61%		165
			166
			167
			168
			169
			170
93	10.04%		171
7	0.76%		172
			173
			174

Source File: /media/Cache/Work/ogre/OgreMain/src/OgreOptimisedUtilGeneral.cpp

```
//  
for (unsigned short blendIdx = 0; blendIdx < numWeightsPerVertex; ++blendIdx)  
{  
    // Blend by multiplying source by blend matrix and scaling by weight  
    // Add to accumulator  
    // NB weights must be normalised!!  
    Real weight = pBlendWeight[blendIdx];  
    if (weight)  
    {  
        // Blend position, use 3x4 matrix  
        const Matrix4& mat = *blendMatrices[pBlendIndex[blendIdx]];  
        accumVecPos.x +=  
            (mat[0][0] * sourceVec.x +  
             mat[0][1] * sourceVec.y +  
             mat[0][2] * sourceVec.z +  
             mat[0][3])  
            * weight;  
        accumVecPos.y +=  
            (mat[1][0] * sourceVec.x +  
             mat[1][1] * sourceVec.y +  
             mat[1][2] * sourceVec.z +  
             mat[1][3])  
            * weight;  
        accumVecPos.z +=  
            (mat[2][0] * sourceVec.x +  
             mat[2][1] * sourceVec.y +  
             mat[2][2] * sourceVec.z +  
             mat[2][3])  
            * weight;  
        if (pSrcNorm)  
        {  
            // Blend normal
```

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

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**



Batching

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



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

Summary

- Introduction to Streamline and Performance Capture
- Working out Limiting Factor
 - Fragment Bound
 - Vertex Bound
 - Bandwidth Bound
 - CPU Bound
- Other Streamline Features
- Offline Shader Compiler
- Techniques