

# Integrating Mali GPU with the Browser

**Wasim Abbas**  
Staff Engineer, Mali Ecosystem

# What's Coming?

- Introduction
- Mali architecture
- Optimizing for Mali
- Common pitfalls
- Questions

# Introduction

- Technical Lead of Middleware graphics team in Mali Ecosystem
  - Working with Skia, FireFox (Gecko) and WebKit teams (GPU acceleration for The Web)
  - Have worked with Mali content optimization and creation tools
  - Past expertise in the Game industry, engine development
  - Past expertise teaching Game engine architecture
- 
- Who are you?

# Graphics in the Embedded World

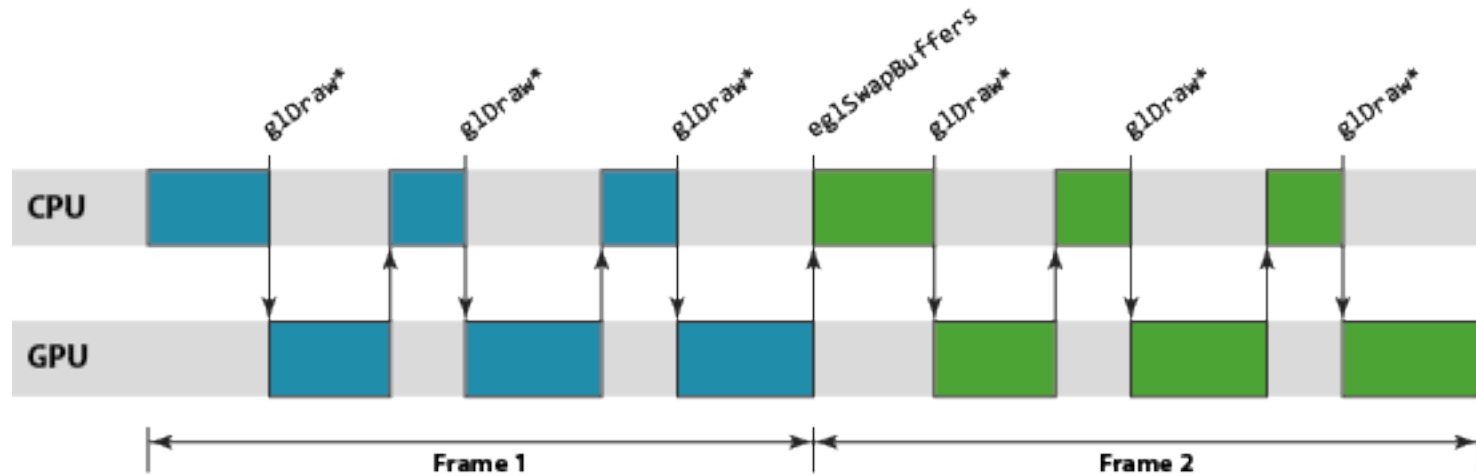
- Embedded / mobile graphics fundamentally different from PC
  - Not just a scaled down version of the desktop world
- Completely different power and bandwidth budgets
  - No dedicated GPU memory or bus – all shared with CPU, video, DSP
  - Need to last days rather than hours on a single battery charge
- Requires dedicated solutions
  - Must handle “PC use cases” with a mobile phone power budget
  - Maximize processing efficiency in an embedded memory subsystem

# Optimizing for Mali

- Understanding Mali architecture
- Understanding your browser and underlying renderer
- Understanding OpenGL ES API and what happens under the hood
- Multi-level optimizations at system and application level
- Optimizations at client level

# Abstract Rendering Machine

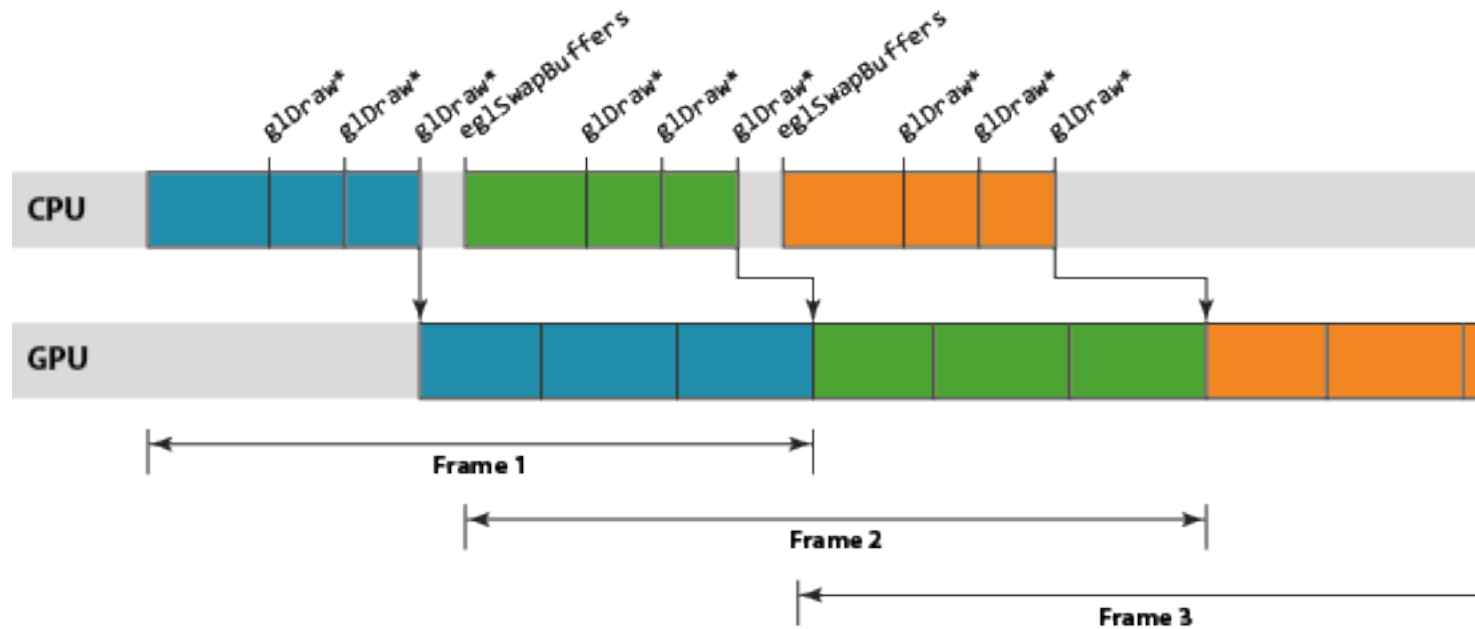
```
glDraw(1)  
glDraw(2)  
glDraw(3)  
eglSwapBuffers()
```



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-1>

# Abstract Rendering Machine

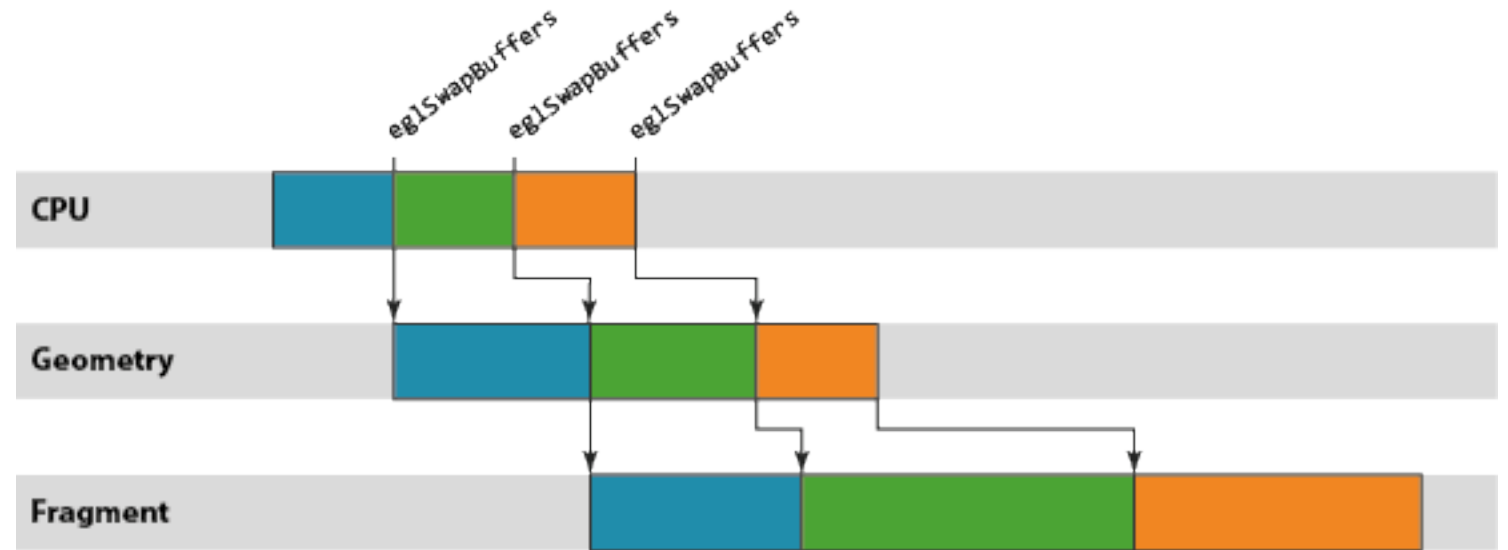
Synchronous API, asynchronous execution



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-1>

# Abstract Rendering Machine

- Mali GPU pipeline are scheduled on a per render-target basis
  - Window surface
  - Off-screen render buffer
- Two step process
  - Vertex shading
  - Fragment shading

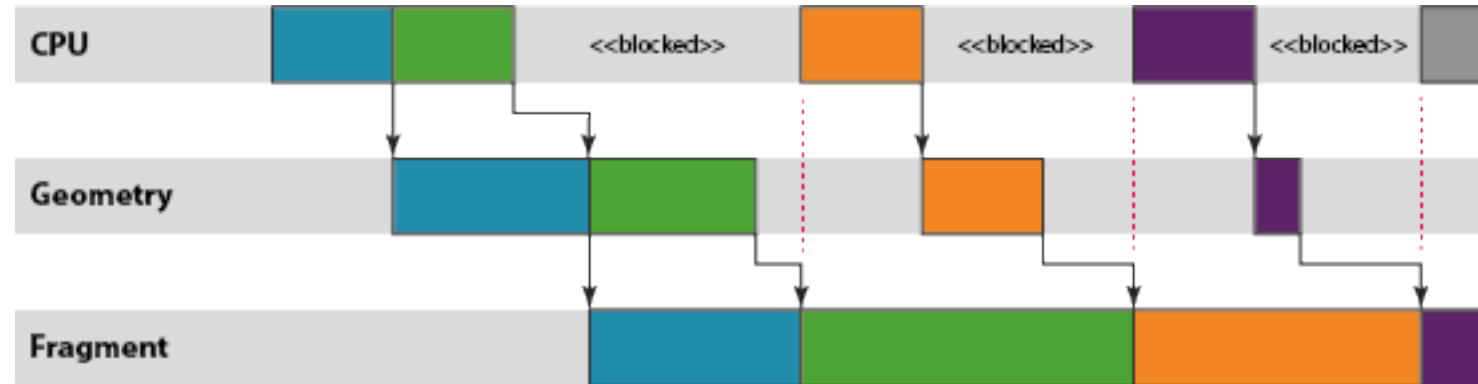


Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-1>



# Abstract Rendering Machine

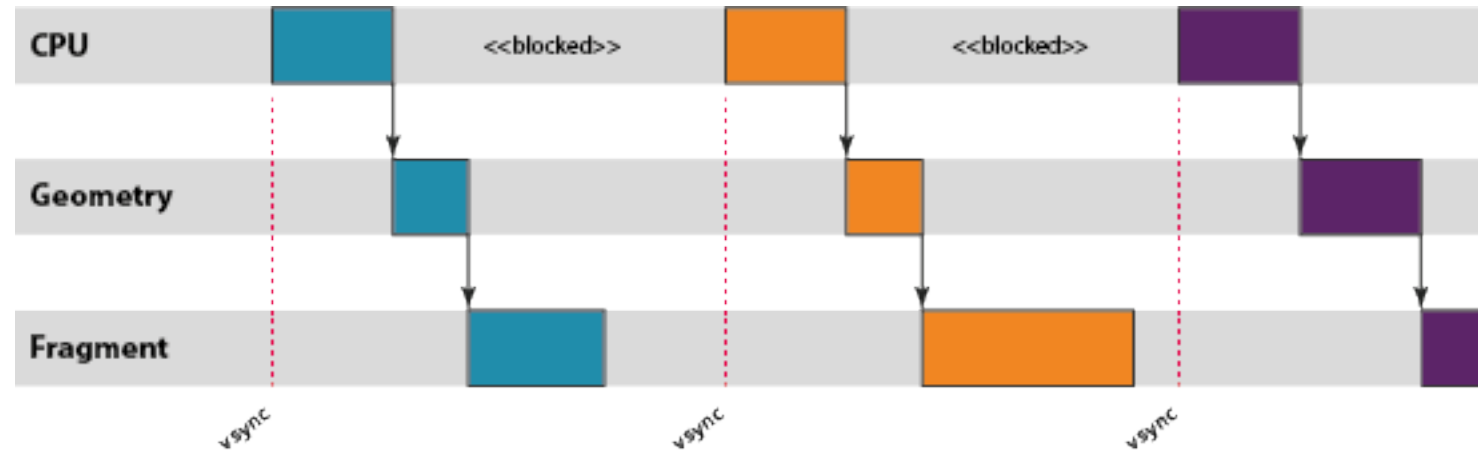
- Throttling can happen
  - Surface flinger refuses to give buffers if already issued N



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-1>

# Abstract Rendering Machine

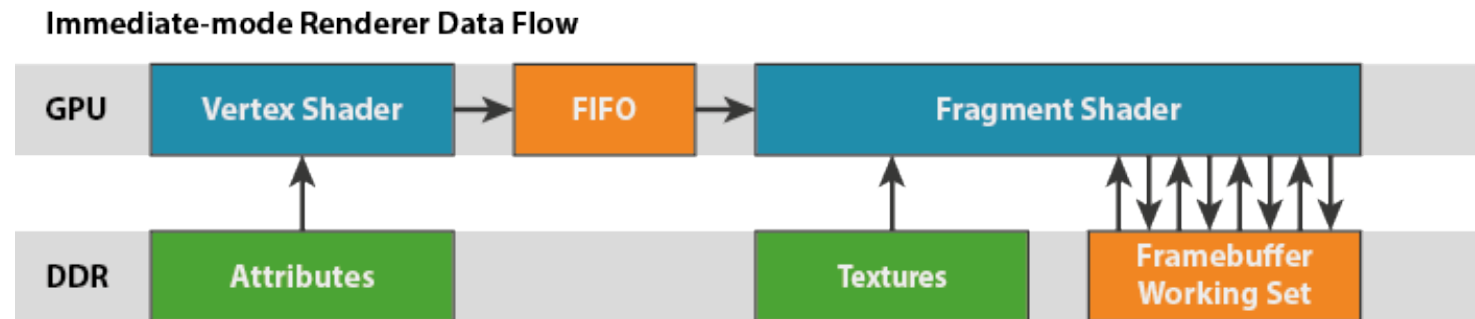
- Throttling can happen
  - Display driver can't display faster than GPU can render



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-1>

# Immediate Mode Rendering

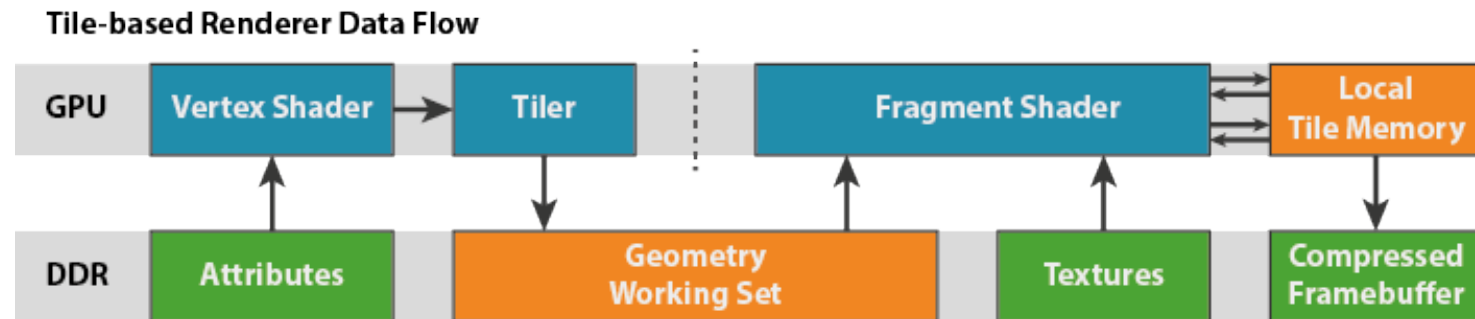
```
foreach( primitive )  
    foreach( fragment )  
        render fragment
```



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-2>

# Tile Based Rendering

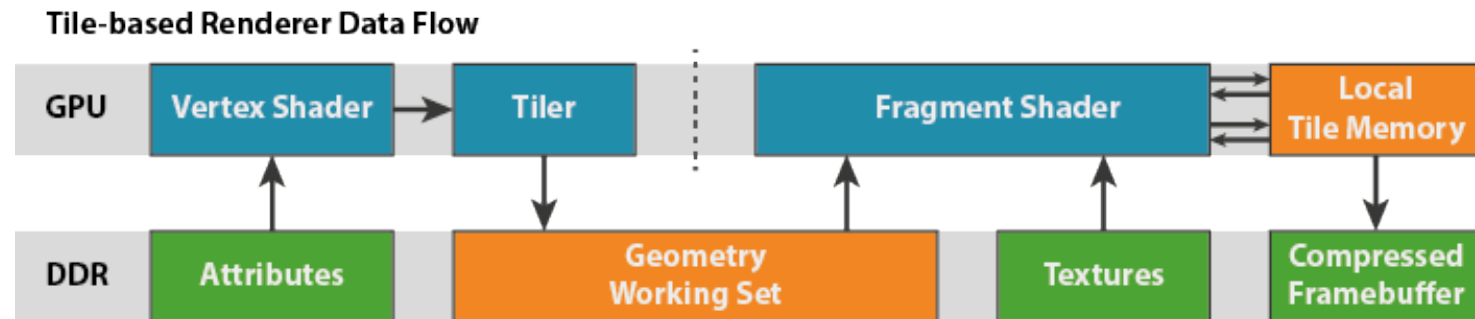
```
foreach( tile )  
    foreach( primitive in tile )  
        foreach( fragment in primitive in tile )  
            render fragment
```



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-2>

# Tile Based Rendering

- Mali processes all geometry before processing fragment shading
  - Geometry pass outputs are large, so must be written to main memory
    - Vertex shader output: varyings
    - Tiler output: polygon list



- Mali processes each 16x16 pixel tile to completion
  - Fragment shading working set is small so can keep in local memory
    - Blending and multi-sampling is inexpensive
    - Tile written back to memory when complete, allows transaction elimination and compression

Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/02/03/the-mali-gpu-an-abstract-machine-part-2>

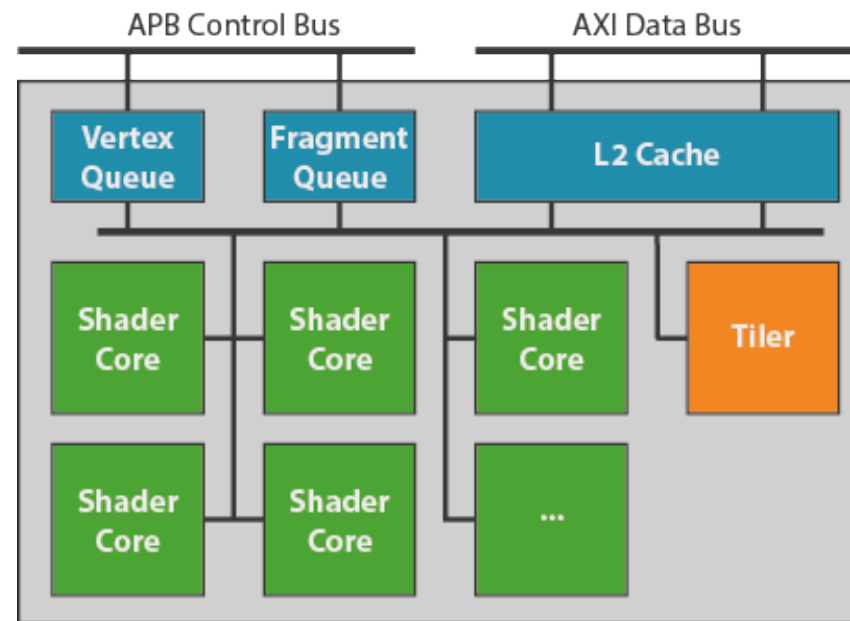
# Tile Based Rendering Advantages

- All accesses to the working set are local accesses, which is both fast and low power
- Blending is both fast and power-efficient
- A tile is sufficiently small that we can actually store enough samples locally in the tile memory to allow 4x, 8x and 16x Antialiasing
- Write out can be discarded if tile hasn't changed (Transaction Elimination)
- Output can be compressed when sending to display
- Depth and stencil can be discarded if not required

# Mali GPU

- Unified shader core architecture
  - Vertex shaders
  - Fragment shaders
  - Compute kernels

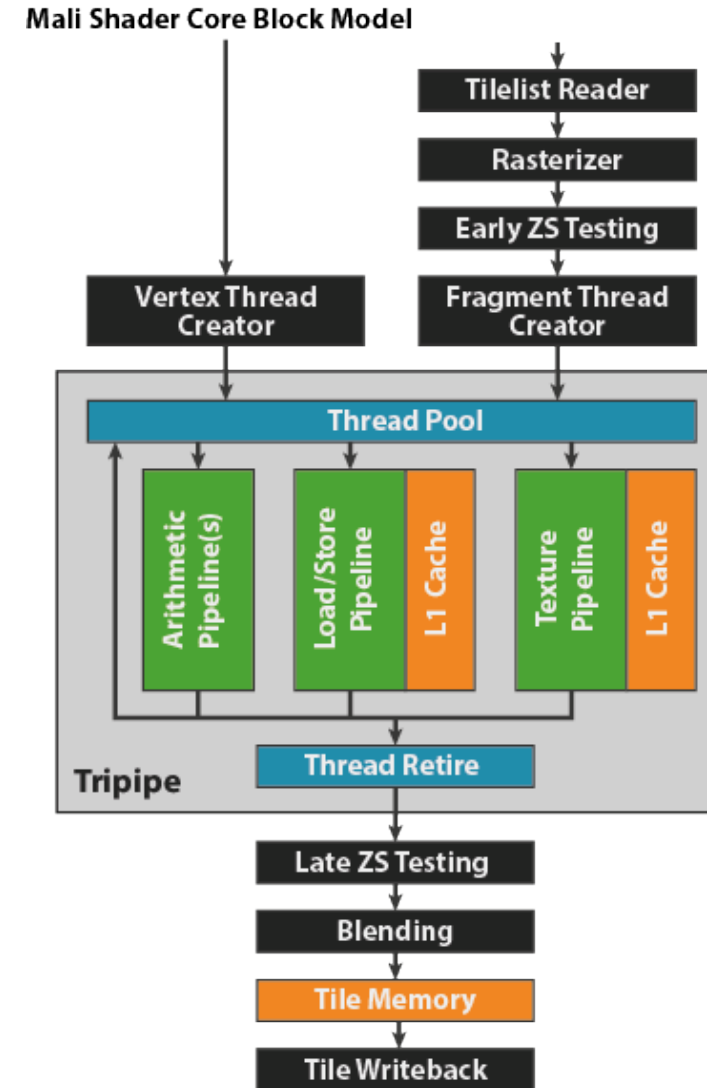
Mali GPU Block Model



Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/03/12/the-mali-gpu-an-abstract-machine-part-3--the-shader-core>

# Mali Shader core

- Fixed function units perform
  - Rasterizing triangles
  - Performing early/late depth testing
  - Blending
  - Writing back a whole tile
- Tri-pipe programmable shader core
  - ALU(s)
  - Texture Pipe
  - Load/Store

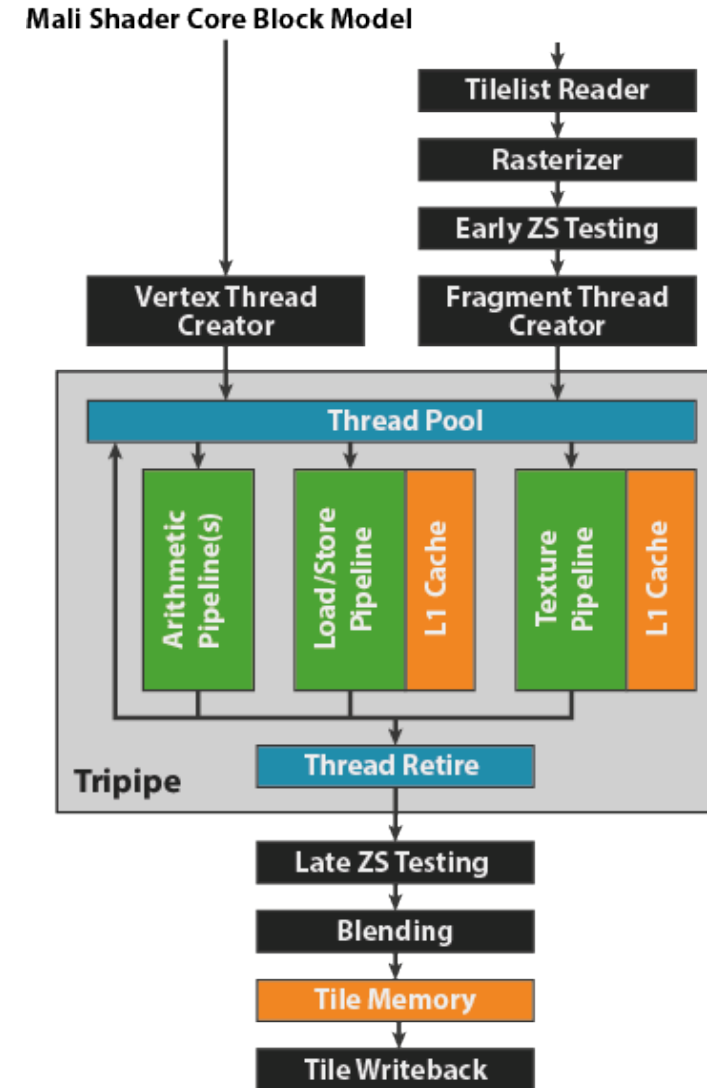


Original article from Peter Harris at <http://community.arm.com/groups/arm-mali-graphics/blog/2014/03/12/the-mali-gpu-an-abstract-machine-part-3--the-shader-core>



# Mali Shader core

- Fixed function front-end and back-end
  - Wrapped around a programmable tripipe
- Capacity for 256 threads
  - Register file supports 1024 128-bit registers
    - 256 threads @ 4 register each
    - 128 threads @ 8 registers each
    - 64 threads @ 16 registers each
- Baseline capabilities:
  - One thread spawned per clock
  - One fragment completed per clock
  - One instruction per pipe per clock
  - 4 samples blended per clock
  - One pixel written per clock



# Mali Shader core

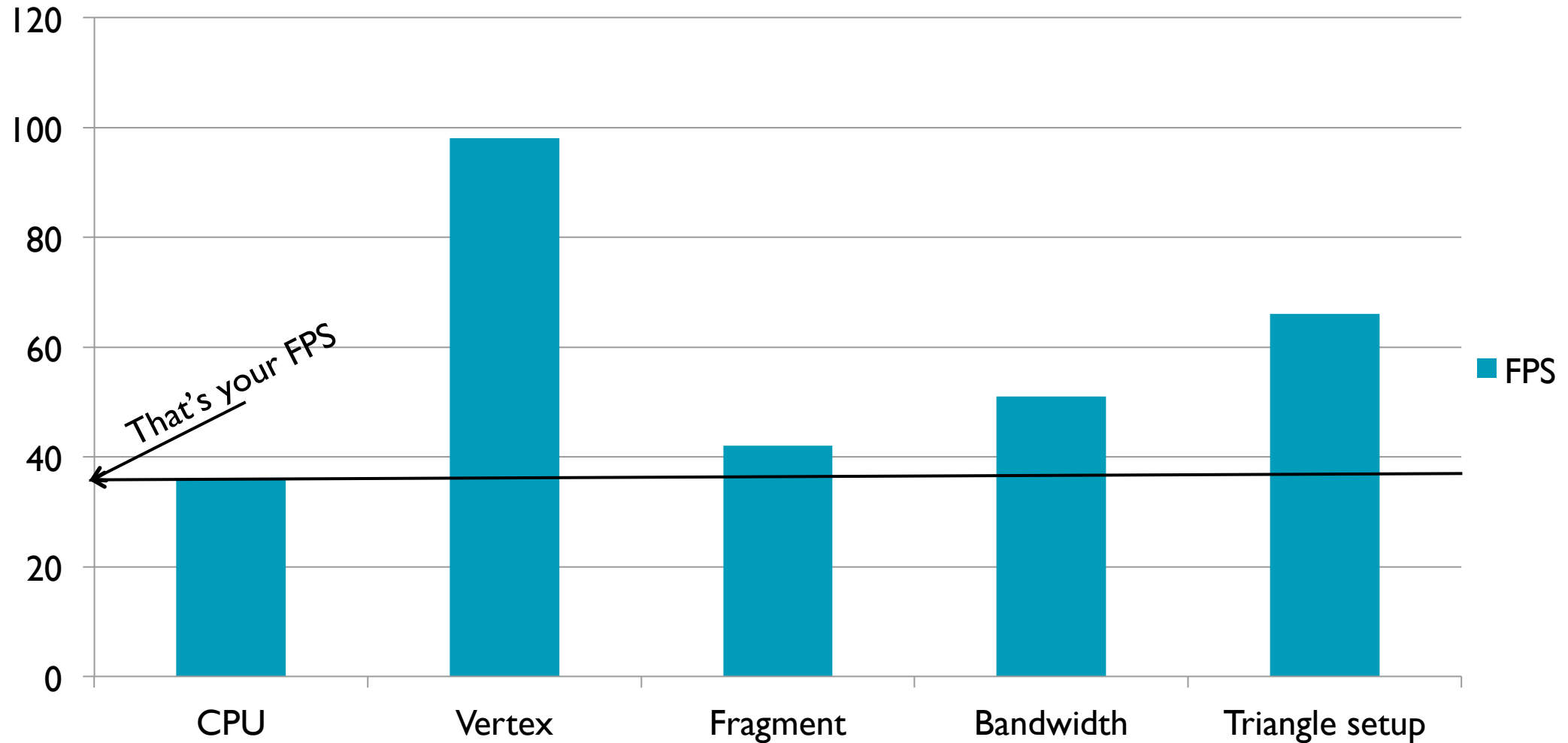
Scale this to a Mali-T760 MP8 running at 600MHz

- Fillrate:
  - 8 pixels per clock = 4.8 GPix/s
  - *That's 2314 complete 1080p frames per second!*
- Texture rate:
  - 8 bilinear texels per clock = 4.8 GTex/s
  - *That's 38 bilinear filtered texture lookups per pixel for 1080p @ 60 FPS!*
- Arithmetic rate:
  - 17 FP32 FLOPS per pipe per core = 163 FP32 GFLOPS
  - *That's 1311 FLOPS per pixel for 1080p @ 60 FPS!*
- Bandwidth:
  - 256-bits of memory access per clock = 19.2GB/s read and write bandwidth!
  - *That's 154 bytes per pixel for 1080p @ 60 FPS!*

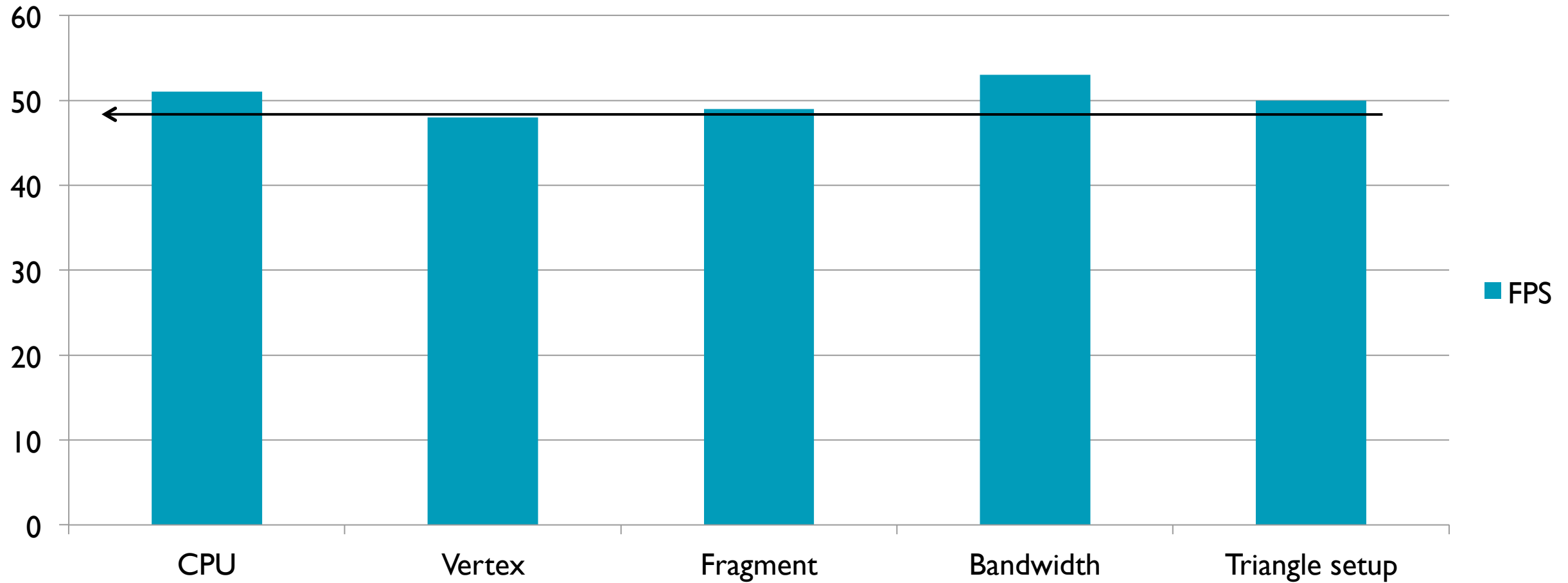
# Optimizing Application

- Five (potential) bottleneck stages:
  - CPU bound
  - Fragment processing bound
  - Bandwidth bound
  - Vertex processing bound
  - Triangle setup bound
- These stages don't stack!
  - You are always bound by at least one of them
  - The one you are bound by them determines the performance
- The goal of optimizing is to be bound by all stages simultaneously

# The weakest link decides your FPS

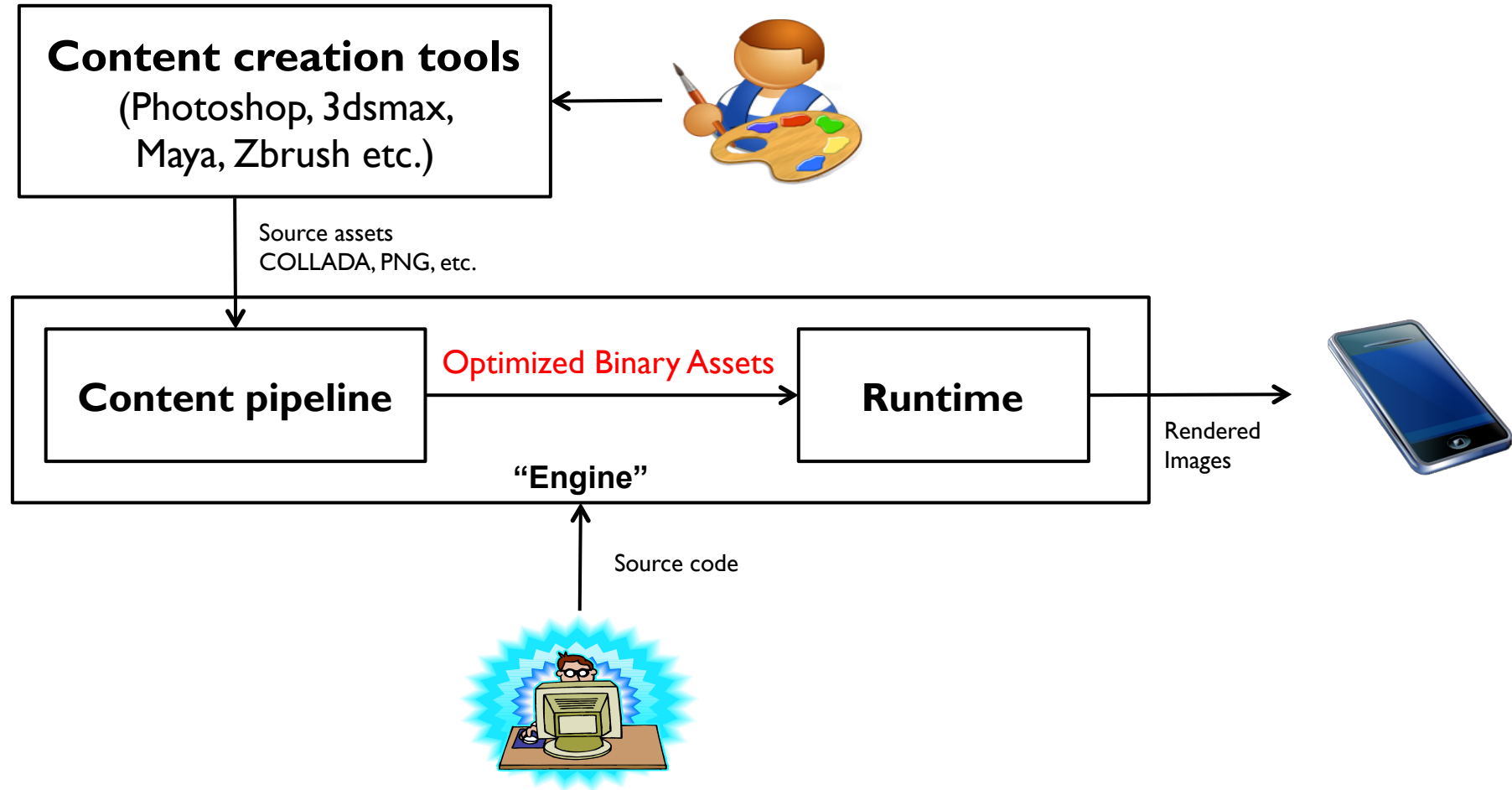


# Good apps are “multi-bound”

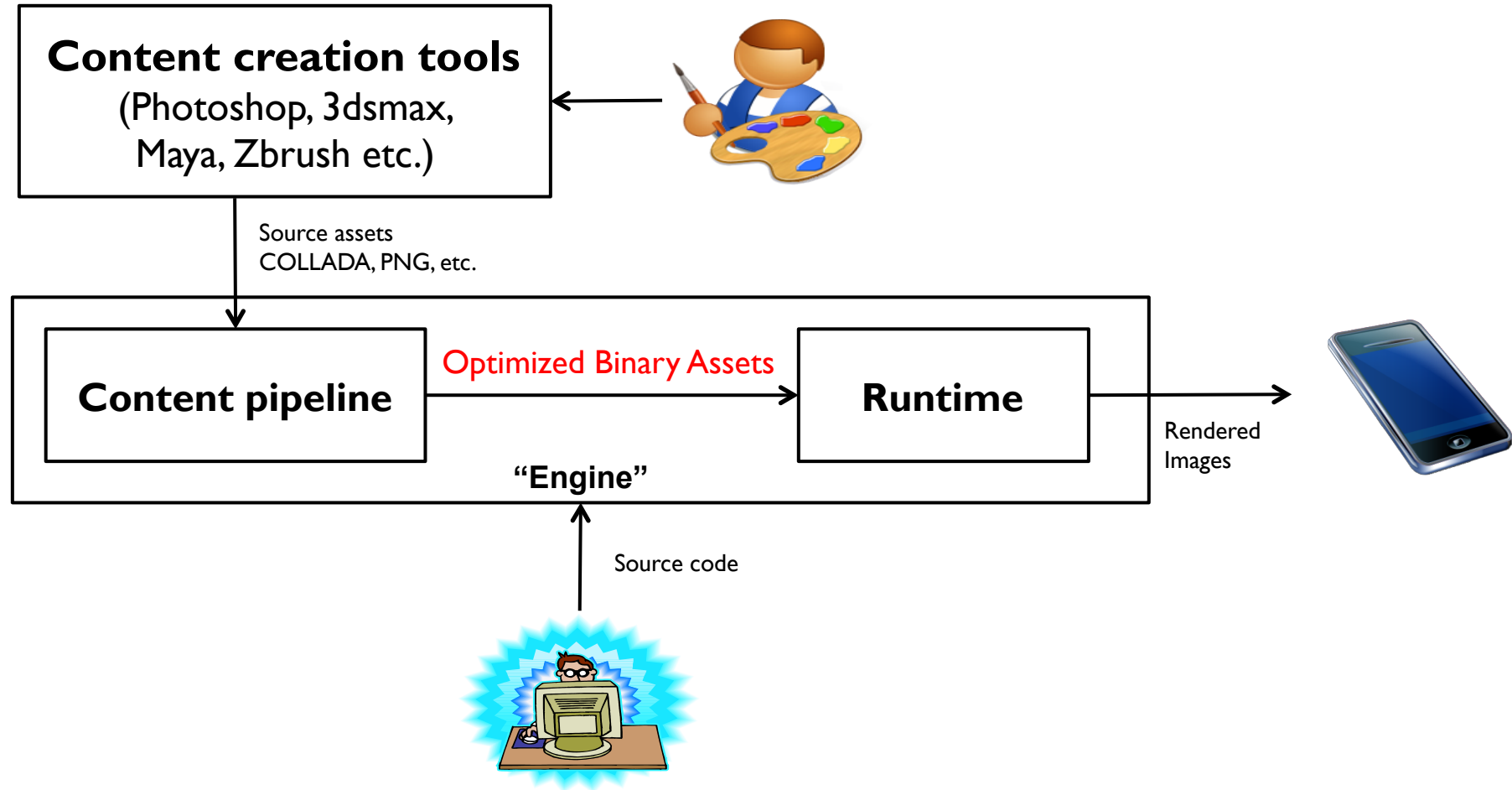


...as long as you are bound at a high FPS 😊

# Graphics app development process



# Is Browser a Graphics Application?



# Common Pitfalls

- Render target and Framebuffer management `glBindFramebuffer(0)` and `eglSwapBuffers()`
- Incremental rendering and using `EGL_BUFFER_PRESERVED`?
- `glDrawArrays` or `glDrawElements` and how many? What is batching?
- Vertex buffer packing which one is better? (PTNPTN, PPTTNN, PP,TT,NN)
- `glTexImage2D()` uploads, reusing texture IDs and mip-mapping
- `glClear(ALL_BUFFERS)` and when to clear?
- Precision lowp/mediump/highp which one should be used?
- `glReadPixels()` why is it slow and shouldn't be used?
- `glDiscardFramebufferExt()` and `glInvalidateFramebuffer()` how can they help? I have never used them.
- Why is bandwidth important?
- Can I use exotic blend modes?
- Overdraw what is it?
- Difference between Immediate vs. Deferred renderers?
- Why a desktop browser with software renderer ported to Mobile kills performance?
- etc.



# Render Target and Framebuffer Management

- On-screen window render targets
  - `eglSwapBuffer()` defined end of frame
  - Depth and Stencil can be discarded
- Off-screen framebuffer render targets
  - There is no end of frame call
  - Instead framebuffer unbind is used to infer flush
  - Application might be re-using depth and stencil so must be preserved
- Developers should bind each off-screen fbo once per frame and render to completion

# Render Target and Framebuffer Management

## Well-structured rendering sequence

- `#define ALL_BUFFERS COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT | STENCIL_BUFFER_BIT`
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 0 (window surface)
- `glBindFramebuffer( 1 )` // Switch away from FBO 0, does not trigger rendering
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 1
- `...` // Draw FBO 1 to completion
- `glBindFramebuffer(0)` // Switch to FBO 0, unbind and flush FBO 1 for rendering
- `glDraw...( ... )` // Draw something else to FBO 0 (window surface)
- `glBindFramebuffer( 2 )` // Switch away from FBO 0, does not trigger rendering
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 2
- `...` // Draw FBO 2 to completion
- `glBindFramebuffer(0)` // Switch to FBO 0, unbind and flush FBO 2 for rendering
- `glDraw...( ... )` // Draw something else to FBO 0 (window surface)
- `eglSwapBuffers()` // Tell EGL we have finished, flush FBO 0 for rendering

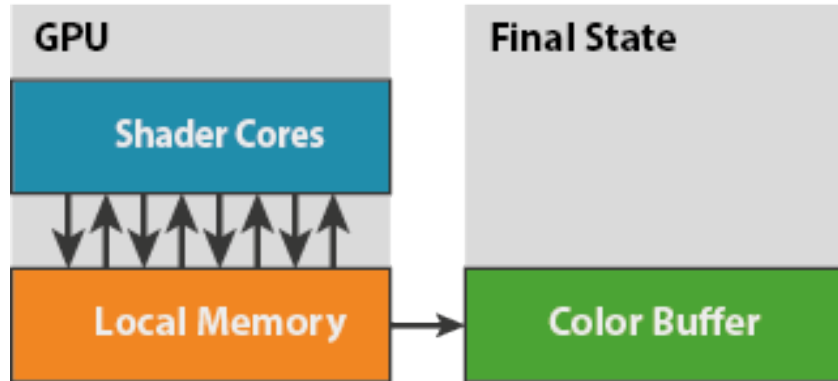
# Render Target and Framebuffer Management

## Badly-structured rendering sequence

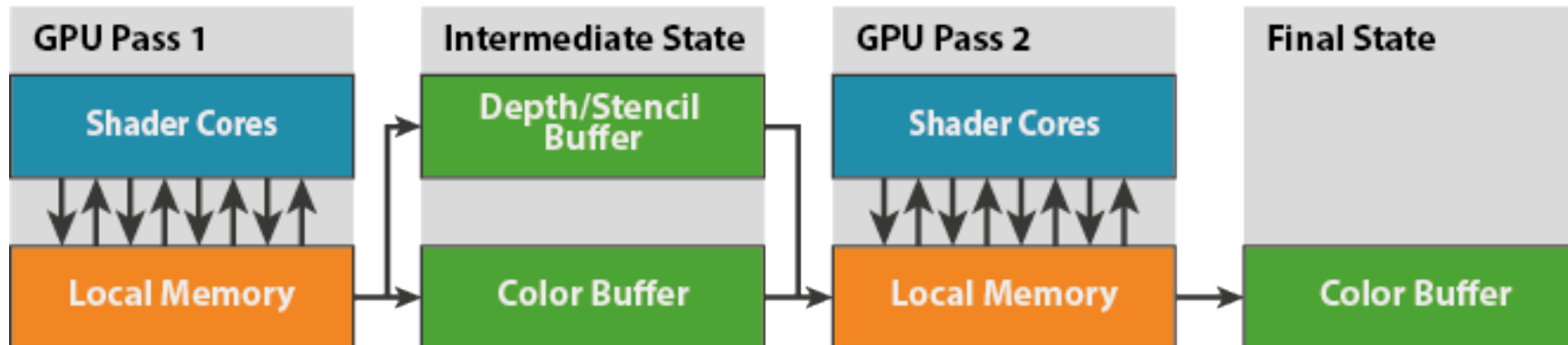
- `#define ALL_BUFFERS COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT | STENCIL_BUFFER_BIT`
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 0 (window surface)
- `glBindFramebuffer( 1 )` // Switch away from FBO 0, does not trigger rendering
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 1
- `glBindFramebuffer(0)` // Switch to FBO 0, unbind and flush FBO 1 for rendering
- `glDraw...( ... )` // Draw something else to FBO 0 (window surface)
- `glBindFramebuffer( 1 )` // Rebind away from FBO 0, does not trigger rendering of FBO  
// However, rebinding FBO 1 requires us to reload old render  
// state from memory, and write over the top of it
- `glDraw...( ... )` // Draw something to FBO 1
- `glBindFramebuffer(0)` // Switch to FBO 0, unbind and flush FBO 1 (again)
- `glDraw...( ... )` // Draw something else to FBO 0 (window surface)
- `eglSwapBuffers()` // Tell EGL we have finished, flush FBO 0 for rendering

# Render Target and Framebuffer Management

Well Behaved One-pass Render



Badly Behaved Two-pass Render



# Render Target and Framebuffer Management

## Even better-structured rendering sequence

- `#define ALL_BUFFERS COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT | STENCIL_BUFFER_BIT`
- `static const GLenum invalid_ap[2] = { GL_DEPTH_ATTACHMENT, GL_STENCIL_ATTACHMENT };`
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 0 (window surface)
- `glBindFramebuffer( 1 )` // Switch away from FBO 0, does not trigger rendering
- `glClear( ALL_BUFFERS )` // Clear initial state
- `glDraw...( ... )` // Draw something to FBO 1
- `...` // Draw FBO 1 to completion
- `glInvalidateFramebuffer( GL_FRAMEBUFFER, 2, &invalid_ap[0] );` // Only keep color
- `glBindFramebuffer(0)` // Switch to FBO 0, unbind and flush FBO 1 for rendering
- `glDraw...( ... )` // Draw something else to FBO 0 (window surface)
- `glBindFramebuffer( 2 )` // Switch away from FBO 0, does not trigger rendering
- `...`
- `eglSwapBuffers()` // Tell EGL we have finished, flush FBO 0 for rendering

# Incremental rendering

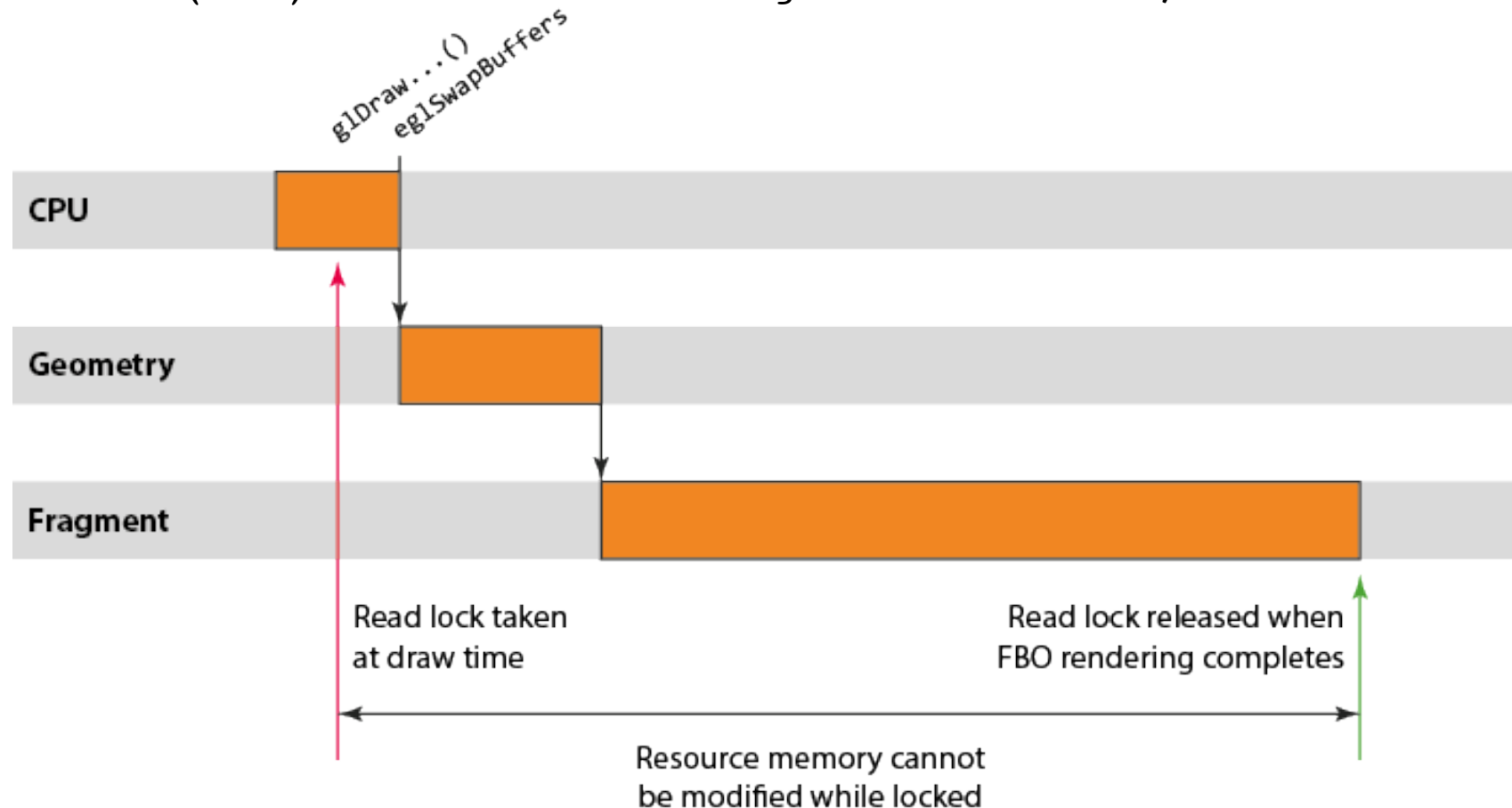
- What is EGL\_BUFFER\_PRESERVED?
  - Window surface create with EGL\_SWAP\_BEHAVIOR configured as EGL\_BUFFER\_PRESERVED
  - Default behavior is EGL\_BUFFER\_DESTROYED
- Can I only render what is changed?
  - No. Because most graphics systems are N-buffered
  - Old frame is not what you think it is
- When is it worth using?

# What is batching?

- What is a draw call
- Drawcalls take CPU cycles
  - `glDrawArrays` and `glDrawElements`
- Browsers draw small items separately

# Updating Dynamic Resources

```
glBindTexture(1)           // Bind texture 1, version 1
glDrawElements(...)         // Draw reading texture 1, version 1
glTexSubImage2D(...)        // Modify texture 1, so it becomes version 2
glDrawElements(...)         // Draw reading the texture 1, version 2
```





# What precision should I use

- ARM Mali Midgard range of GPU's:
  - Shader Cores:
    - lowp = mediump = fp16
    - highp = fp32
- ARM Mali Utgard range of GPU's:
  - Geometry Shader Core:
    - lowp = mediump = highp = fp32
  - Fragment Shader Core:
    - lowp = mediump = fp16
    - highp = not supported\*
- \*We have one special "fast path" for varyings used directly as texture coordinates which is actually fp24.

# Other Common Pitfalls

- Vertex buffer packing which one is better? (PTNPTN, PPTTNN, PP,TT,NN)
- Why is bandwidth important?
- Can I use exotic blend modes?
- Overdraw what is it?
- Difference between Immediate vs. Deferred renderers?
- Why a desktop browser with software renderer ported to Mobile kills performance?
- etc.

Questions?

# Thank You

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