Best Coding Practices for Mobile Platforms

Roberto Lopez Mendez
Senior Software Engineer, ARM

ARM Game Developer Day – London
03/12/2015

Agenda

- Introduction
  - Current mobile device capabilities
  - ARM® Mali™ GPU Architecture
- Best practises to overcome the different types of bottleneck:
  - CPU
  - Vertex processor
  - Fragment processor
  - Bandwidth
- Summary

Two Approaches When Developing for Mobile Platforms

- No easy way to make the most from mobile GPUs

The hard way

Design the app with no restrictions:
- Reflections, shadow, fanciness

Make lots of painful compromises during implementation

End up with something unattractive and underperforming

The wise way

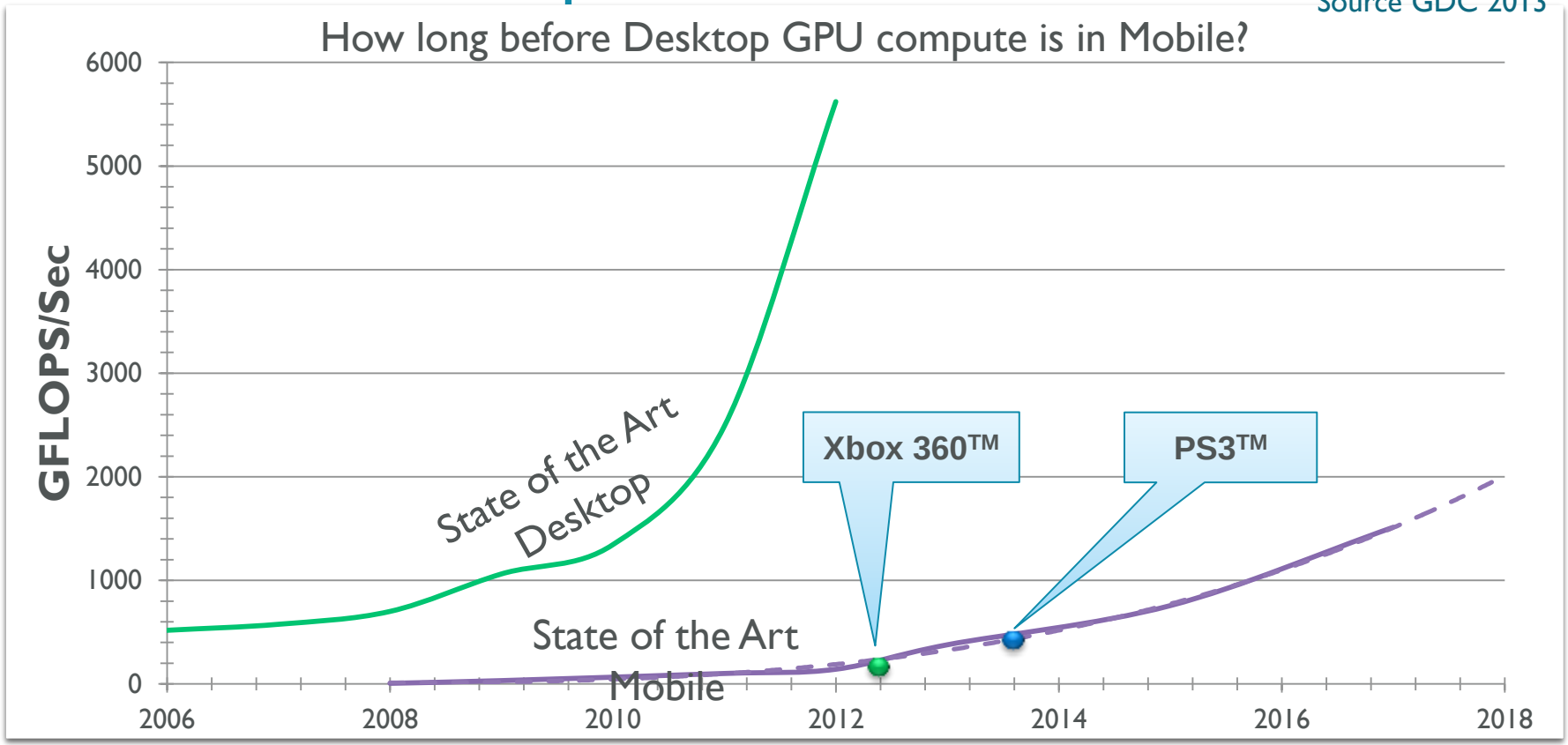
Design the app based on knowledge of the strengths and specifics of your system

Use well known, efficiently implementable effects

End up with something attractive and well performing

Mobile GPU Compute Growth Year on Year

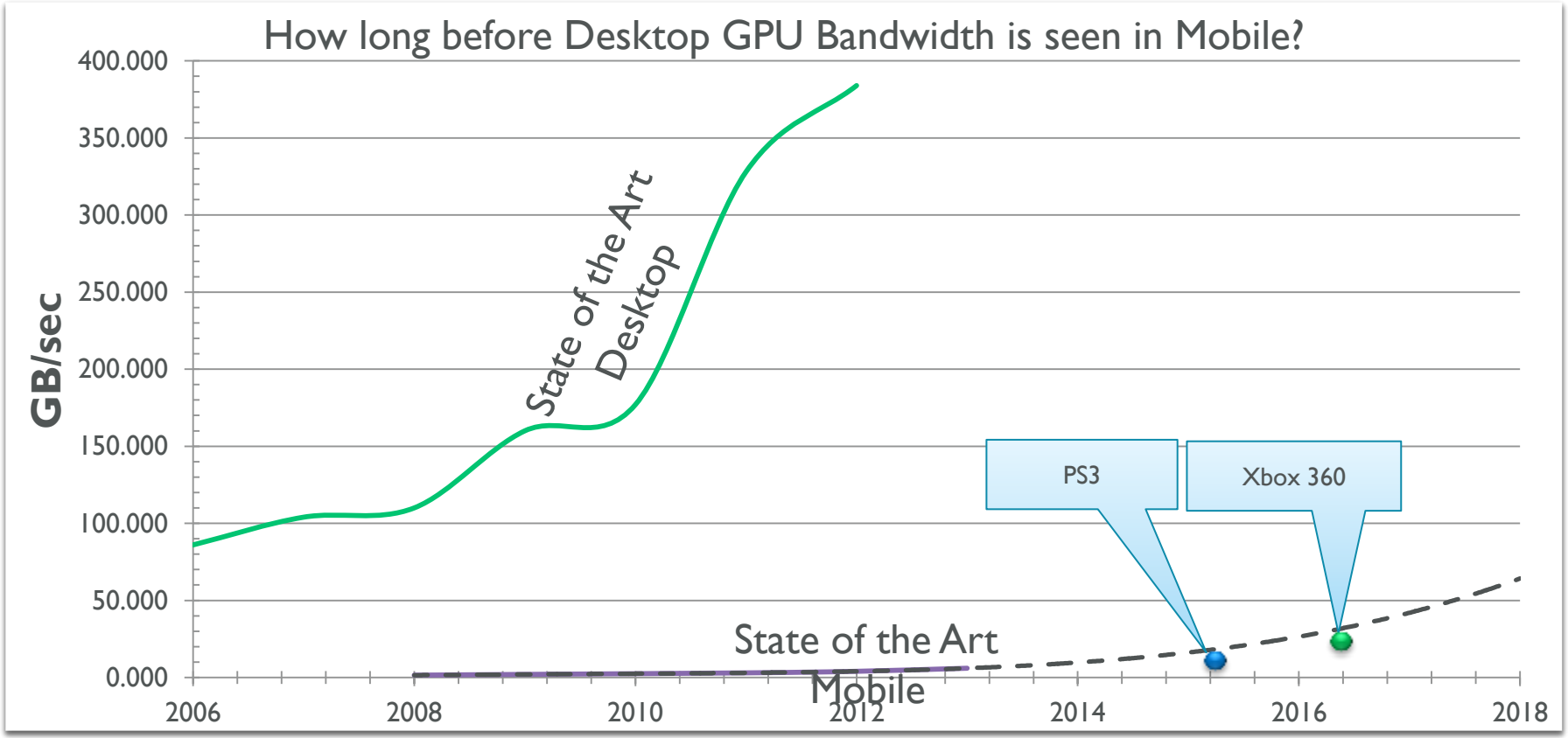
Source GDC 2013



ARM

Mobile GPU Bandwidth Growth Year on Year

Source GDC 2013



ARM

Why is Bandwidth not Progressing as Fast?

- Simple… Power!

- Desktop = 170 Watts to >300 Watts…and that's just the GPU!
- Console = 80-100 Watts (CPU/GPU/WiFi/Network)
- Mobile Platform = 3 - 7 Watts (CPU/GPU/Modem/WiFi)!

ARM Mali Utgard and Midgard Architectures

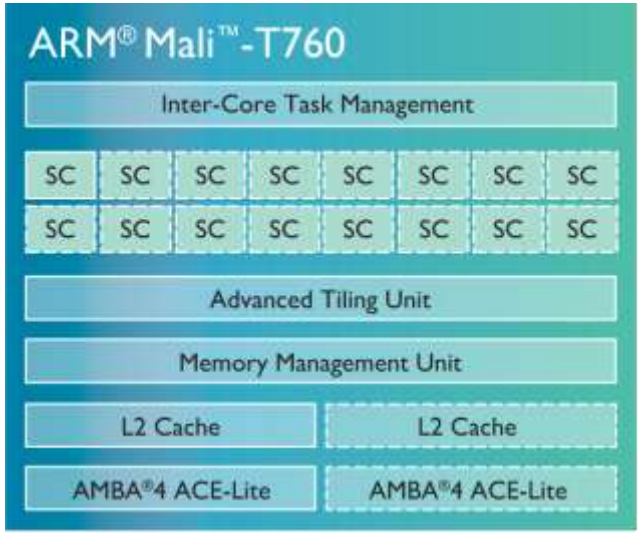
Utgard
Mali-400MP, Mali-450MP, Mali-470

OpenGL® ES 2.0/1.1 support
Scalable to 8 cores*, MSAA

Unified shader cores
GP GPU

Less internal and SoC bandwidth utilization

Midgard
Mali-T720, Mali-T760, Mali-T820, Mali-T830, Mali-T860, Mali-T880



OpenGL® ES 3.1/3.0/2.0/1.1 support and MS Windows® compliant for Direct3D® 11.1.
Full Profile OpenCL®1.2 support
Scalable to 16 cores, MSAA, ASTC, Transaction Elimination, AFBC.

ARM

Samsung Galaxy S6 ARM Mali-T760 MP8 Supported Extensions

GL_ANDROID_extension_pack_es31a
GL_ARM_mali_program_binary
GL_ARM_mali_shader_binary
GL_ARM_rgba8
GL_ARM_shader_framebuffer_fetch
GL_ARM_shader_framebuffer_fetch_depth_stencil
GL_EXT_blend_minmax
GL_EXT_color_buffer_float
GL_EXT_color_buffer_half_float
GL_EXT_copy_image
GL_EXT_debug_marker
GL_EXT_discard_framebuffer
GL_EXT_disjoint_timer_query
GL_EXT_draw_buffers_indexed
GL_EXT_geometry_shader
GL_EXT_gpu_shader5
GL_EXT_multisampled_render_to_texture
GL_EXT_occlusion_query_boolean
GL_EXT_primitive_bounding_box
GL_EXT_read_format_bgra
GL_EXT_robustness
GL_EXT_shader_io_blocks
GL_EXT_shader_pixel_local_storage
GL_EXT_shadow_samplers
GL_EXT_sRGB
GL_EXT_sRGB_write_control
GL_EXT_tessellation_shader
GL_EXT_texture_border_clamp
GL_EXT_texture_buffer
GL_EXT_texture_cube_map_array
GL_EXT_texture_format_BGRA8888
GL_EXT_texture_rg
GL_EXT_texture_sRGB_decode
GL_EXT_texture_storage
GL_EXT_texture_type_2_10_10_10_REV
GL_KHR_blend_equation_advanced
GL_KHR_blend_equation_advanced_coherent
GL_KHR_debug
GL_KHR_texture_compression_astc_hdr
GL_KHR_texture_compression_astc_ldr
GL_OES_compressed_ETC1_RGB8_texture
GL_OES_compressed_paletted_texture
GL_OES_copy_image
GL_OES_depth_texture
GL_OES_depth_texture_cube_map
GL_OES_depth24
GL_OES_draw_buffers_indexed
GL_OES_EGL_image
GL_OES_EGL_image_external
GL_OES_EGL_sync
GL_OES_element_index_uint
GL_OES_fbo_render_mipmap
GL_OES_geometry_shader
GL_OES_get_program_binary
GL_OES_gpu_shader5
GL_OES_mapbuffer
GL_OES_packed_depth_stencil
GL_OES_primitive_bounding_box
GL_OES_required_internalformat
GL_OES_rgb8_rgba8
GL_OES_sample_shading
GL_OES_sample_variables
GL_OES_shader_image_atomic
GL_OES_shader_io_blocks
GL_OES_shader_multisample_interpolation
GL_OES_standard_derivatives
GL_OES_surfaceless_context
GL_OES_tessellation_shader
GL_OES_texture_3D
GL_OES_texture_border_clamp
GL_OES_texture_buffer
GL_OES_texture_compression_astc
GL_OES_texture_cube_map_array
GL_OES_texture_npot
GL_OES_texture_stencil8
GL_OES_texture_storage_multisample_2d_array
GL_OES_vertex_array_object
GL_OES_vertex_half_float
GL_OVR_multiview
GL_OVR_multiview_multisampled_render_to_texture
GL_OVR_multiview2

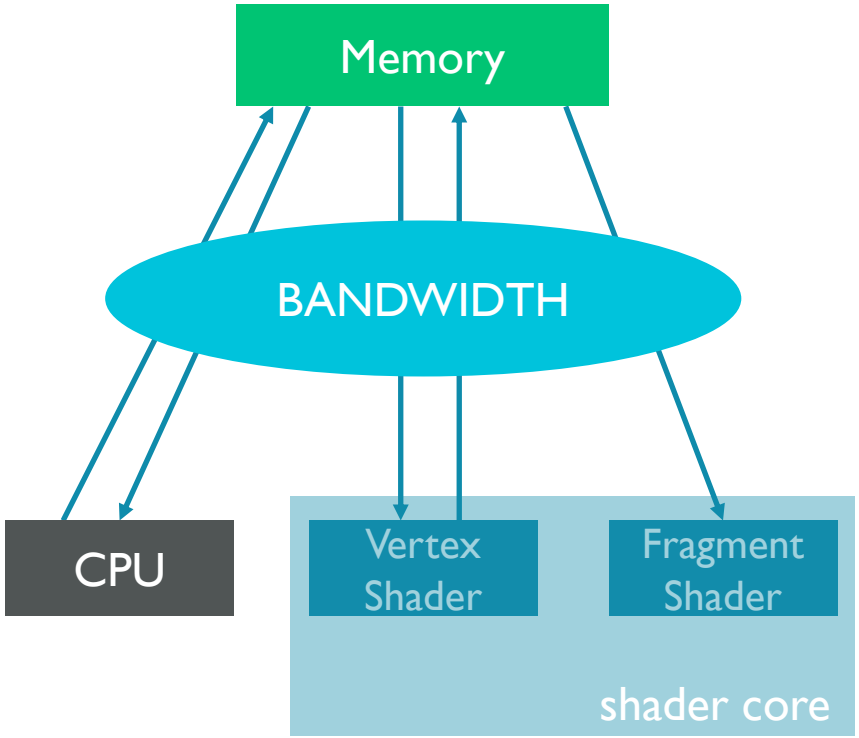
ARM

Advantages of Tile Based Architecture

- Tile-based rendering minimizes the amount of power hungry external memory accesses needed during rendering
- 4xMSAA – very efficient in ARM Mali GPUs
- Blending is fast and power efficient as it is performed on chip without data transfer to main memory
- Write bandwidth saving by only updating tiles that have changed from the previous frame: Skips writing the tile to the FB if the content is the same, saving SoC power

Factors Influencing the Load on the System Elements

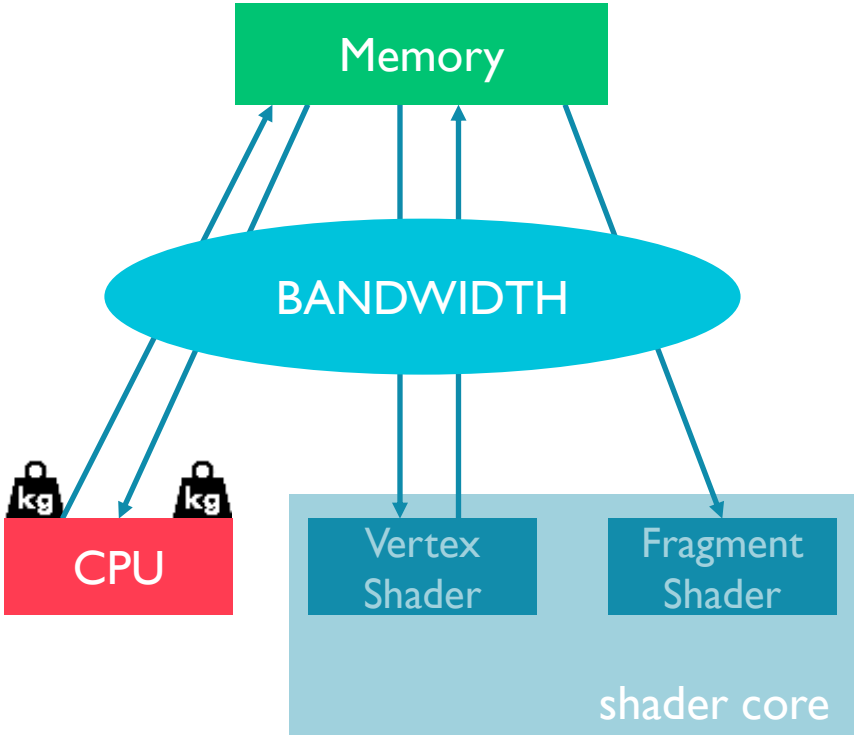
The Key Elements in the System



Factors Influencing CPU Load

- Time spent in application logic
- Draw call overhead
- Culling
- Uniform data copying
- Vertex/index data copying
- Per-frame resource updating

- They all stack!!!



ARM

CPU: Draw Call Overhead

- The load associated with sending API commands to the driver
- Draw call are calls to
  - glDrawArrays(), glDrawElements()
  - glDrawArraysInstanced, glDrawBuffers, glDrawElementsInstanced, glDrawRangeElements
  - glDrawArraysIndirect, glDrawElementsIndirect

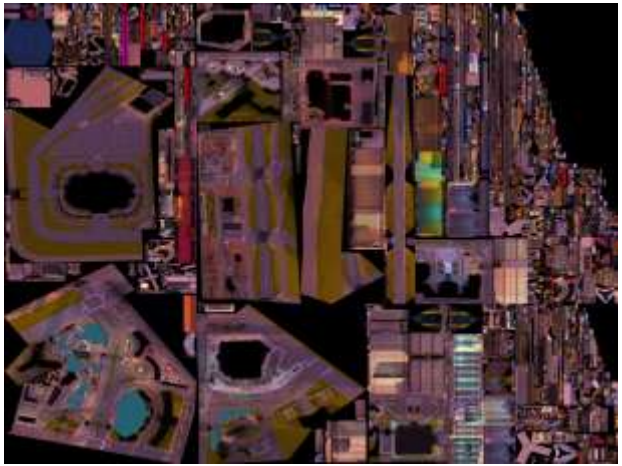
- This is where most of the work in the driver happens
  - ~ 0.03 - 0.1 ms/draw call
- Meaning:
  - @ 60 FPS ~ few hundred draw calls per frame (depends on the CPU)

Main Draw Call Optimization Measures

- Batching
- Culling

Batching: Fewer Draw Calls, Less Overhead

- The goal of batching is to regroup as many meshes in fewer buffers to get better performance.
- Build a texture atlas (collage) containing the textures of all the parts of the objects
  - Usually the artists prepares the atlas
- Update texture coordinates accordingly
- Build common vertex and index buffers that contain the vertices of all grouped meshes.



Heavy atlasing from Gangstar Vegas - Gameloft, GDC 2014

Mixing and Serving up a Batch

Vertex shader:

```
uniform mat4 transforms[3];
attribute vec4 pos;
attribute float id;
void main(){
  mat4 trans=transforms[(int)id];
  glPosition=trans*pos;
}
```

Mesh 1 [(1,1),(0,1),(0,0),(1,0)]
Mesh 2 [(1,2),(0,2),(0,0)]
Mesh 3 [(2,2),(2,1),(1,1)]
Index1: [0,1,2,0,2,3]
Index2: [0,1,2]
Index3: [0,1,2]

Attrib1:[(1,1),(0,1),(0,0),(1,0),(1,2),(0,2),(0,0),(2,2),(2,1),(1,1)]
Index: [0,1,2,0,2,3,4,5,6,7,8,9]

Attrib2:[0,0,0,0,1,1,1,2,2,2]

GL Code:

```
float transforms[16*instanceCount];
.
. /* Load matrices into float array */
.
glUniformMatrix4fv(transID,4,false,transforms);
```

OpenGL ES 2.0 “Instancing”

- Use the batching concept for the same mesh

```
drawBuilder.addGeometry(geo1);
drawBuilder.addGeometry(geo1);
drawBuilder.addGeometry(geo1);
drawBuilder.addGeometry(geo1);
drawBuilder.Build();
```

- The overhead in the vertex shader will be always less than issuing a draw call for each instance of the object

Culling

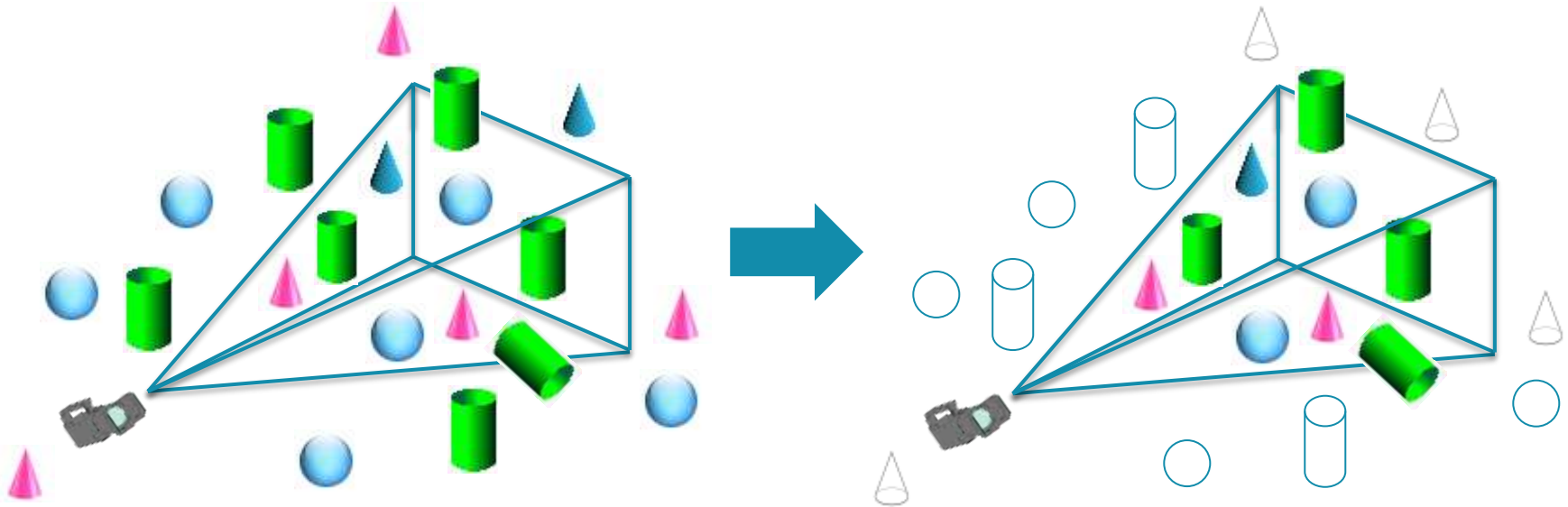
- Culling: is the art of finding the smallest set of objects that really "need" to be drawn by avoiding drawing objects that won't contribute (much) to the visual end result

- Several types of culling, use those that best fit your app.

- Good culling:
  - Reduces the number of draw calls
  - Reduces the amount of geometry to process
  - Reduces overdraw and fill rate

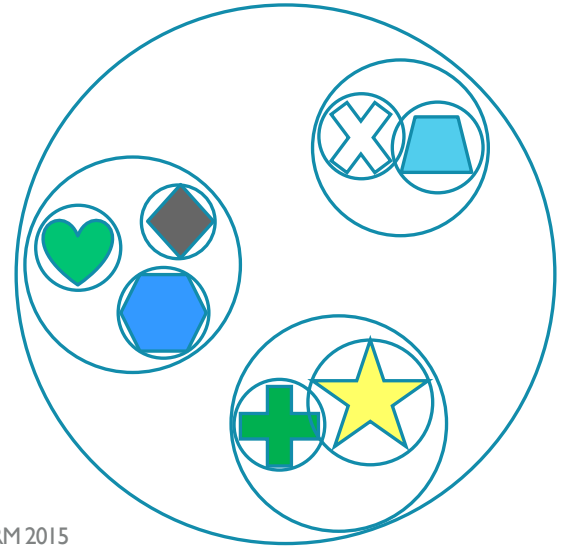
Frustum Culling

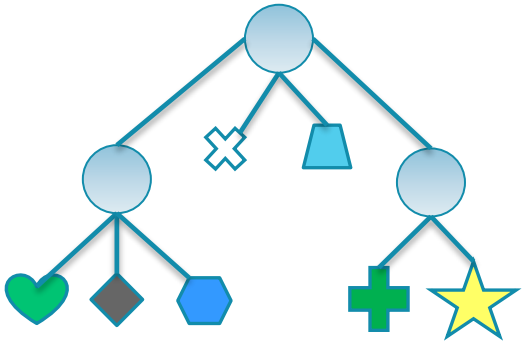
- Avoid drawing objects that are not in the view frustum.



Hierarchy Culling

- Break down your world into a tree-like structure
  - Wrap all objects in a bounding volume (sphere or box)
  - Create a hierarchy of bounding volume nodes
- Benefits: Speed up CPU culling, especially for large scenes



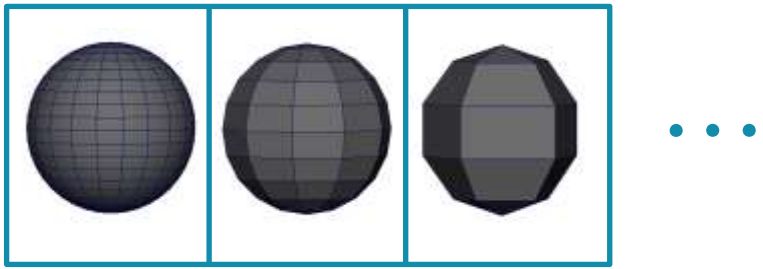


Distance Culling

- Cull nodes based on their screen size
- if(size_bounding_volume) / distance_to_camera) < threshold) return;

Batching + Culling + Level-of-Detail

- Prebuilt batches containing objects that might appear in the scene and only allocate instances on-the-fly as objects passed culling.
- Allocate instances front-to-back with decreasing level-of-detail
- Batch



• • •

- Only the used section of the batch is drawn
- Design your level carefully. Same object types in the same areas to exploit this system

Front-to-Back Sorting

- Mali supports early-Z rejections of fragment when depth testing is enabled
- Front-to-back sorting allows you to make use of it and discard fragments before it reaches the fragment shader
- Sorting methods reduce overdraw and material changes at the cost of CPU

Iron Man 3 - Gameloft: Sorting Objects Before Rendering

- When no sorting is applied:
  - Mid-range device: average 18 FPS, constant micro-freezes
  - Over 35 program changes per frame

Iron Man 3 - Gameloft: Material Sorting Results

- Reduced program changes to an average of 16 (35 -> 16)
  - Micro-freezes are reduced.
- Average 22 FPS, smoother gameplay (18 -> 22)
- But still a lot of overdraw…

Iron Man 3 - Gameloft: Front-to-Back Sorting Results

- Sorting first by material, objects with the same material then sorted front to back
- Constant 24 FPS (22 -> 24)
- The skybox is rendered as the last opaque object

A Good OpenGL ES Rendering Approach

- Prebuild draw call objects (renderables) initialized load-time
  - Holds all geometry textures and shaders for all objects types
  - For static batching determine the "maximum" number of instances per object on the screen
  - For dynamic batching no work is needed during load-time
- A graph of cull nodes holds the current scene
  - Nodes contain bounding volume, transform, material properties and pointer to renderer that draw the object
- For every frame you need to:
  - Clear all renderables (free all instances)
  - Traverse the cull graph and generate a list of drawing candidates
  - Sort drawing candidates front-to-back
  - Traverse the sorted list, allocate instances in the renderables as you go until you run out of instances
  - Draw non-empty renderables

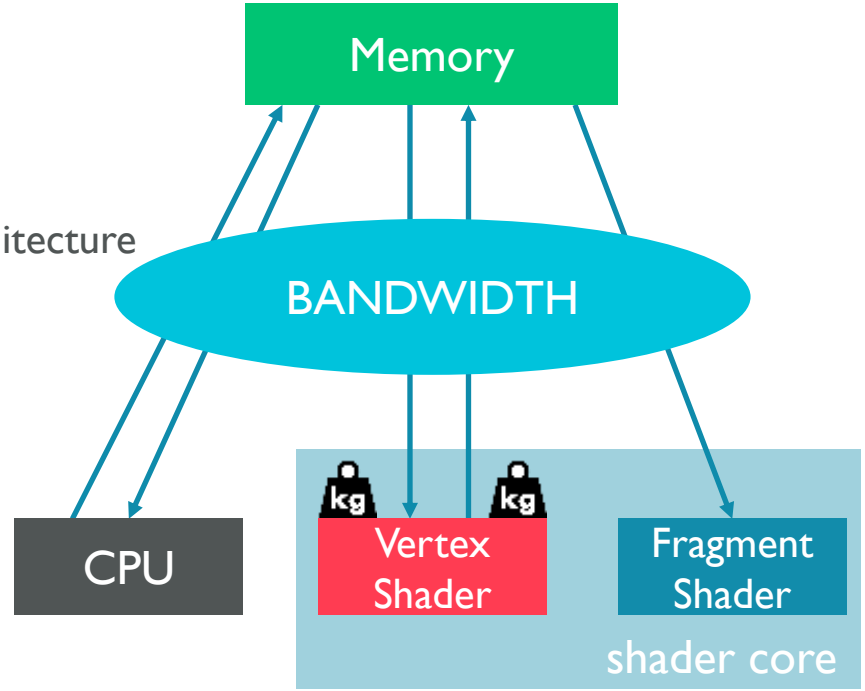
CPU Optimization Summary

- It is all about reducing the number of draw calls.
  - Keep draw calls ~ few hundred per frame

- Rendering engine must do:
  - Culling
  - Batching
  - Front-to-back sorting
  - Dynamic level-of-detail
  - All at once!

Factors Influencing Vertex Processing Load

- Long vertex shader
  - Overly sophisticated effects
  - Badly implemented shaders
  - Effects that don't map well to the architecture
- A lot of vertices
  - Lack of culling
  - Lack of LOD
  - Too high-poly dataset
- Expensive vertex formats
- Cache inefficiency



LOD Reduction

- Only objects that take up a lot of screen real state need to be high-poly
- Draw poly-reduced version for far away objects
- Ask your artist to create LOD-levels for all high-poly models
  - LOD 0 100% vertices
  - LOD 1 50% vertices
  - LOD 2 25% vertices
  - LOD N = (LOD N-1)%/2 vertices

LOD Levels

- Good artist workflow
  - Create high-poly mesh with fine details
  - Recreate low-poly version and project high-poly details into textures
  - Reduce low-poly version into LOD levels with texture seam fixed

- ZBrush, Maya, 3SDStudio Max – tools to simplify meshes
- ZBrush, Maya, 3SDStudio Max, Photoshop, CrazyBump – normal maps

Normal Maps Instead of High-Poly

- Normal maps can be used to represent fine surface details instead of using lots of triangles

- But normal maps are also an expensive per-pixel effect, so this is a trade off

Expensive Vertex Formats

- Is it convenient to use float32 for all vertex attributes?
  - Midgard supports OES_vertex_half _float extension and we can use half-floats natively
  - Position and texture coordinates tend to need highp but for other values we can use mediump
  - For ARM Mali-400/450 we need to handle compact formats manually

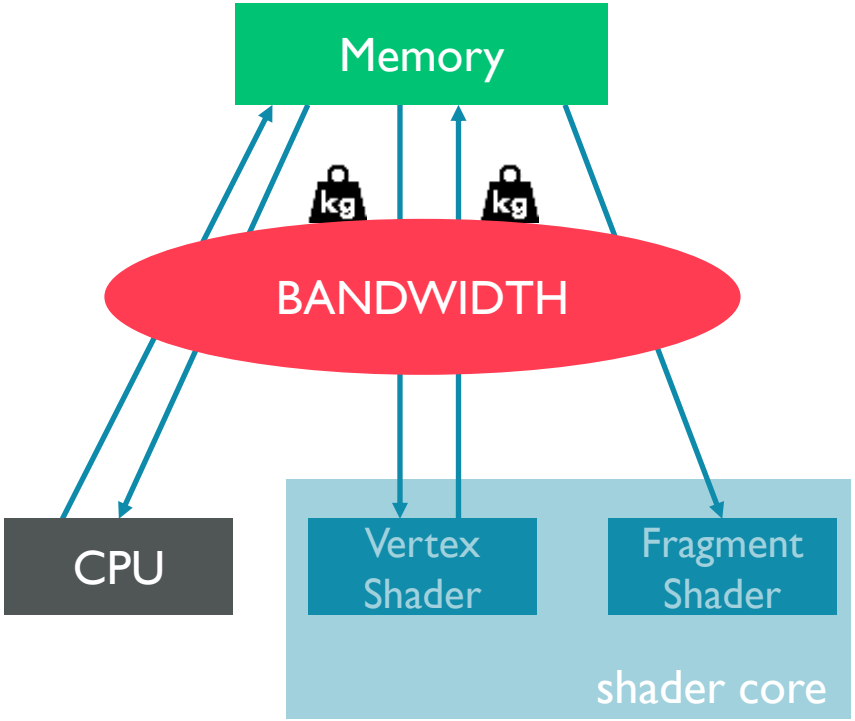
Attribute	Native float32	Compact format
Position	3 floats = 12 bytes	3 floats = 12 bytes
TexCoord	2 floats = 8 bytes	2 floats = 8 bytes
Normal	3 floats = 12 bytes	3 half-float = 6 bytes
Binormal	3 floats = 12 bytes	3 half-float = 6 bytes
Tangent	3 floats = 12 bytes	0 bytes reconstruct in shader
SUM	56 bytes	32 bytes

Vertex Processing Optimization Summary

- It is all about reducing the number of vertices and the shader complexity
  - Use LOD
  - Use normal maps appropriately
  - Manage attribute format wisely
  - Consider moving operations to the CPU/Fragment shader

Factors Influencing Bandwidth Load

- Expensive pixel formats
- Expensive geometry formats
- Excessive geometry data
- Lots of data
  - Render to texture
  - High Resolution
  - HDR
- … and all stacks



- Hard to detect
  - Will behave as pixel or vertex bound
  - Can be found using DS-5 Streamline tool

The Main Bandwidth Eaters

- Reading / writing PIXELS
- Expensive texture and FBO formats easily eat up all your bandwidth
  - In 1920 x 1080, 32bpp is a lot: 8 MB
- FBOs (be careful with OpenGL ES 2.0!)
  - Rendering to non-RGB(A) formats not allowed. Lot of BW wasted if you don't need 3 or more components
  - No < 16bpp renderable formats
  - No suitable format for shadow maps, must use RGBA 32
- Use Texture Compression

Handling Framebuffers Correctly

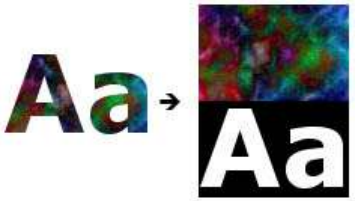
- Avoid unneeded flushes containing a sub-set of the final rendering
  - Bind each off-screen FBO once per frame and render it to completion in one go. Rebinding an FBO requires to reload old render state from memory and write over the top of it.
- Call glClear() for every attachment at the start of each FBO's rendering sequence when the previous contents of the attachments are not needed
  - The render state can be completely dropped when it applies to whole surfaces, so a clear of the whole render target should be performed where possible.
- The application should tell the driver which of the color / depth / stencil attachments can be discarded at the end of rendering the current render pass
  - Failure to invalidate the unneeded buffers may result in them being written back to memory, wasting memory bandwidth and increasing energy consumption of the rendering process.
  - Transient buffers in frame N should be indicated by calling glInvalidateFramebuffer() *before* unbinding the FBO in frame N.
- https://community.arm.com/groups/arm-mali-graphics/blog/2014/04/28/mali-graphics-performance-2-how-to-correctly-handle-framebuffers

Texture Compression Formats Supported in ARM Mali GPUs

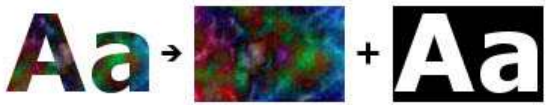
- ETC1 – ARM Mali-400 GPU
  - 4bpp
  - RGB No alpha channel
- ETC2 – ARM Mali-T604 GPU
  - 4bpp
  - Backward compatible
  - RGB also handles alpha & punch through
- ASTC – ARM Mali-T624 GPU and beyond
  - 0.8bpp to 8bpp
  - Supports RGB, RGB alpha, luminance, luminance alpha, normal maps
  - Also supports HDR and 3D textures

ETC1 Texture Compression and Alpha Channel Handling

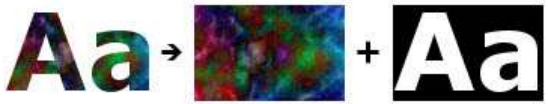
- Create a texture Atlas with TCT
  - The alpha channel is converted to a grayscale image
  - In the fragment shader an additional texture fetch for the alpha channel with the proper coordinates



- Pack Alpha Separately
  - The alpha channel as a second packed texture, combine both in the shader code
  - More flexible but requires a second texture sampler in the shader



- Separate Raw Alpha
  - The alpha channel as a raw 8 bit single-channel image, combined with the texture data in the shader
  - Allows uncompressed alpha but requires a second texture sampler in the shader



ASTC RGBA Available Block Sizes

Compression ratios for a RGBA 8 bit per channel texture of 1024x1024 pixel resolution (4 MB uncompressed size).

Block Size	Texture size	Compr. ratio
4x4	1 MB	4.00
5x4	819 KB	5.00
5x5	655 KB	6.25
6x5	546 KB	7.50
6x6	455 KB	9.00
8x5	409 KB	10.01
8x6	341 KB	12.01

Block Size	Texture size	Compr. ratio
10x5	327 KB	12.53
10x6	273 KB	15.00
8x8	256 KB	16.00
10x8	204 KB	20.08
10x10	164 KB	24.97
12x10	136 KB	30.12
12x12	114 KB	35.93

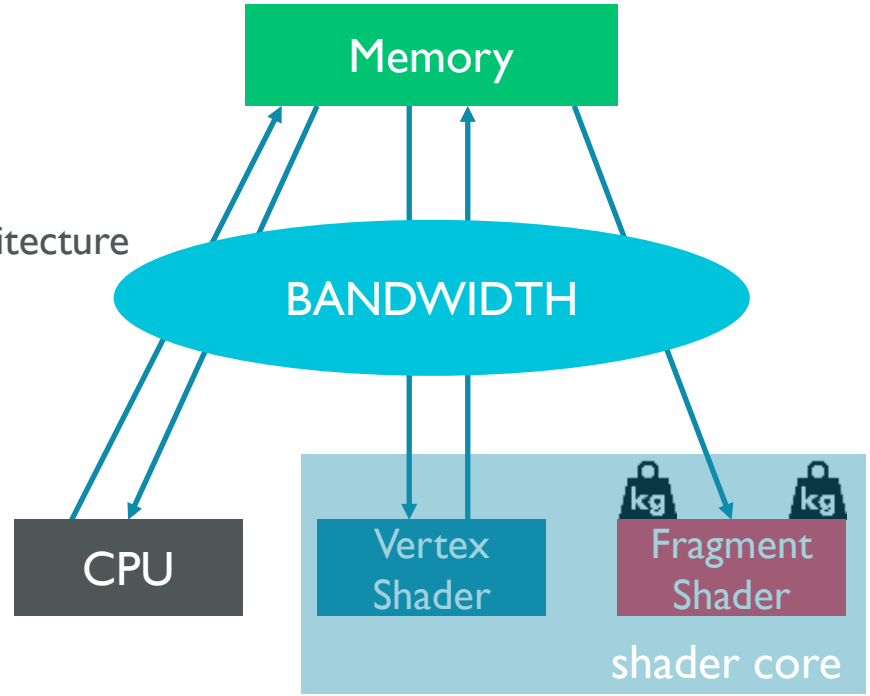
Simple Bandwidth Trade-Offs

- Pulling down the LOD bias in the shader
  - texture2D(tex, tc, 0.5);
  - Trades texture quality for bandwidth
- Turn off tri-linear filtering
  - GL_MAG_FILTER = GL_LINEAR
  - GL_MIN_FILTER = GL_LINEAR_MIPMAP_NEAREST
  - Trades texture filter quality for bandwidth
- Always use mipmaps!

Factors Influencing Fragment Shader Load

- Long fragment shaders
  - Overly sophisticated effects
  - Badly implemented shaders
  - Effects that don't map well to the architecture
- Overdraw
  - Application controlled
  - Z-sorting
  - Particle effects
- Too high resolution

- … They also stack too



Writing Fast Pixel Shaders

- Expensive shaders are OK as long as they cover a limited portion of the screen
  - Cost = Shader weight * number of pixels covered
- Use available tools to analyse your shader
  - The **Mali Offline Compiler** is a brilliant tool for optimizing shaders (MaliDeveloper.arm.com)
- Get used to sacrificing correctness for performance
  - Simplifications are valid if the result is consistent, credible and looks good ☺

Summary

- Graphics programming is all about making trade-offs and compromises vs performance

- Main elements of the system to consider when optimizing mobile games
  - CPU - reduce draw calls, batching, culling, front-to-back sorting, Dynamic LOD
  - VS - Use LOD, normal maps, manage format attributes, consider moving op. to CPU/FS
  - FS - Limit resolution, the screen space for sophisticated effects and particle effects

    Simplify shader. Avoid overdraw by sorting front-to-back

    Consider your budget: available average shader cycles/pixel @ a given resolution at a given FPS
  - BW - Limit reading/writing pixels, render texture pass, full screen post-processing

    Use texture compression, mipmaps

    In GLES 2 very limited render to texture formats

For more information visit the Mali Developer Centre:

http://malideveloper.arm.com

- Revisit this talk in PDF and audio format post event
- Download tools and resources
- Find out more about batching at: http://community.arm.com/groups/arm-mali-graphics/blog/2015/04/13/dynamic-soft-shadows-based-on-local-cubemap
- Find out more about best practices at: https://community.arm.com/groups/arm-mali-graphics/blog/2014/04/28/mali-graphics-performance-2-how-to-correctly-handle-framebuffers

Thank you

ARM

Questions