Getting Faster and Leaner on Mobile: Optimizing Roblox with Vulkan

Arseny Kapoulkine (Roblox)

Joe Rozek (ARM)

Runnable samples

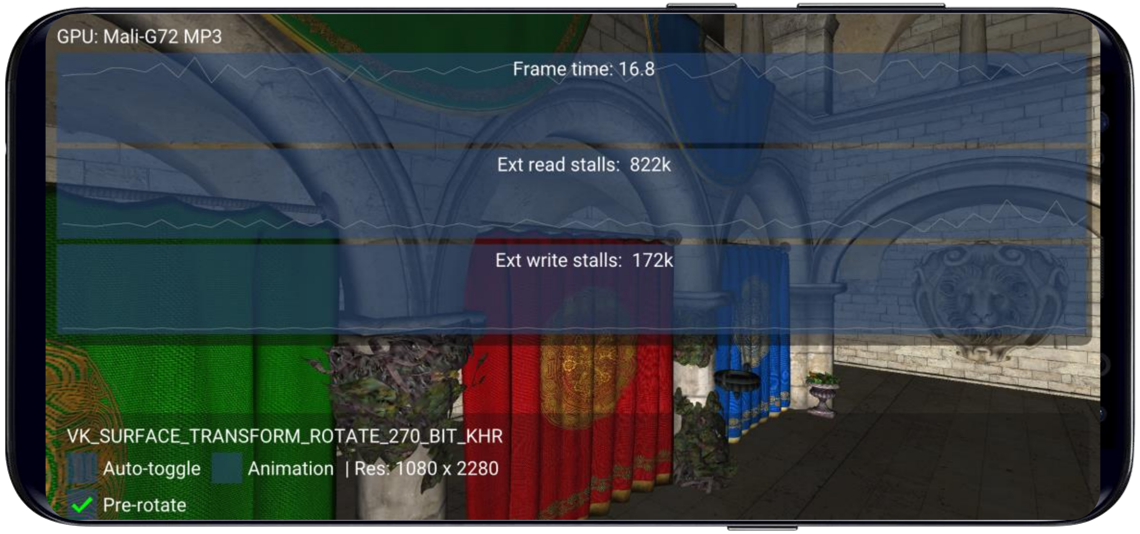
Tutorials

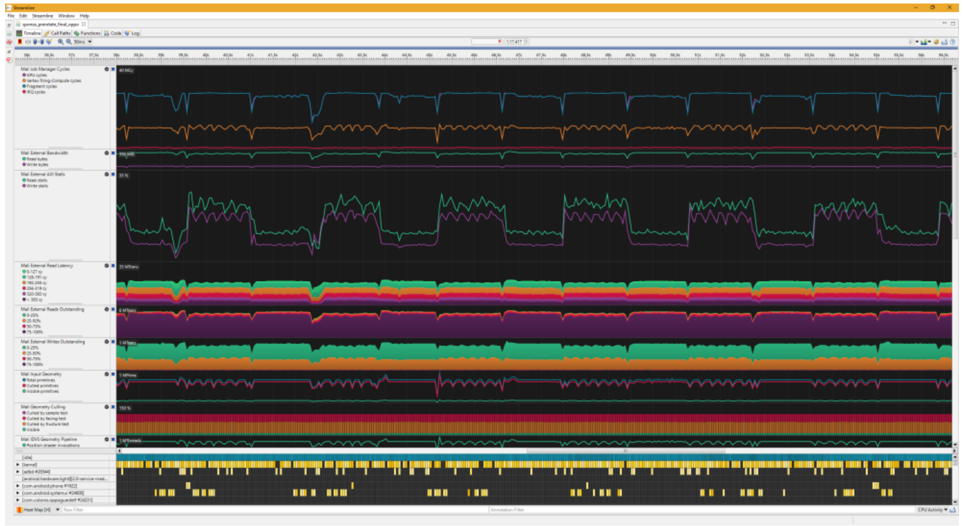
Performance analysis

Mobile-optimized, multi-platform framework

Vulkan best practice for mobile developers

- https://github.com/ARM-software/vulkan_best_practice_for_mobile_developers
- Multi-platform (Android, Windows, Linux)
- Hardware counters displayed on device (no need for root) with HWCPipe
- In-detail explanations, backed-up with data, of best-practice recommendations
- Guide to using performance profiling tools and analysing the results





Framework

Framework

- Platform independent (Android, Linux and Windows)
- Maintain close relationship with Vulkan objects
- Runtime GLSL shader variant generation + shader reflection (Khronos' SPIRV-Cross)
  - Simplify creation of Vulkan objects:
    1. VkRenderPass
    2. VkFramebuffer
    3. VkPipelineLayout
    4. VkDescriptorSetLayout
- Load 3D models (glTF 2.0 format)
  - Internal scene graph

Initialization

Render Pass
- Attachment Description
- Input Attachment
- Output Attachment

Descriptor Set Layout
- Texture Binding
- Uniform Binding

Descriptor Pool
- Texture Binding
- Uniform Binding

Framebuffer
- Render Pass
- Image View
- Image View

Pipeline Layout
- Descriptor Set Layout
- Push constants

Descriptor Set
- Descriptor Set Layout
- Descriptor Pool
- Image View
- Image View

Graphics Pipeline
- Render Pass
- Pipeline Layout
- Subpass
- Shader module
- Shader module
- Vertex Input
- Input Assembly
- Rasterization
- Multisample
- Depth Stencil
- Color Blend

Shader Module
- Texture Resource
- Image Resource
- Input Resource
- Output Resource

Object/Dependency

Application defined

Initialization

Render Pass
- Attachment Description
- Input Attachment
- Output Attachment

Descriptor Set Layout
- Texture Binding
- Uniform Binding

Descriptor Pool
- Texture Binding
- Uniform Binding

Framebuffer
- Render Pass
- Image View
- Image View

Pipeline Layout
- Descriptor Set Layout
- Push constants

Descriptor Set
- Descriptor Set Layout
- Descriptor Pool
- Image View
- Image View

Graphics Pipeline
- Render Pass
- Pipeline Layout
- Subpass
- Shader module
- Shader module
- Vertex Input
- Input Assembly
- Rasterization
- Multisample
- Depth Stencil
- Color Blend

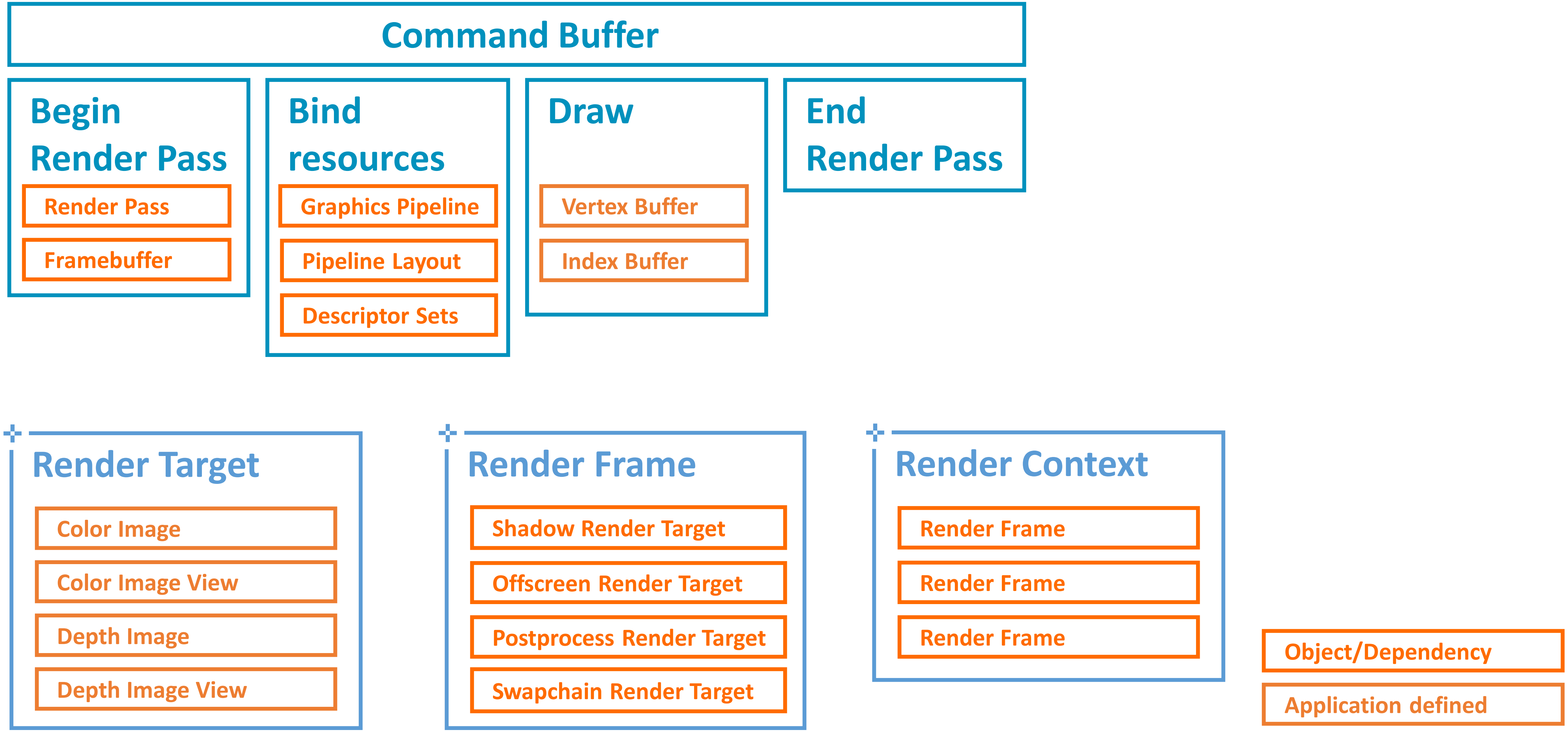
Shader Module
- Texture Resource
- Image Resource
- Input Resource
- Output Resource

GLSL Compiler

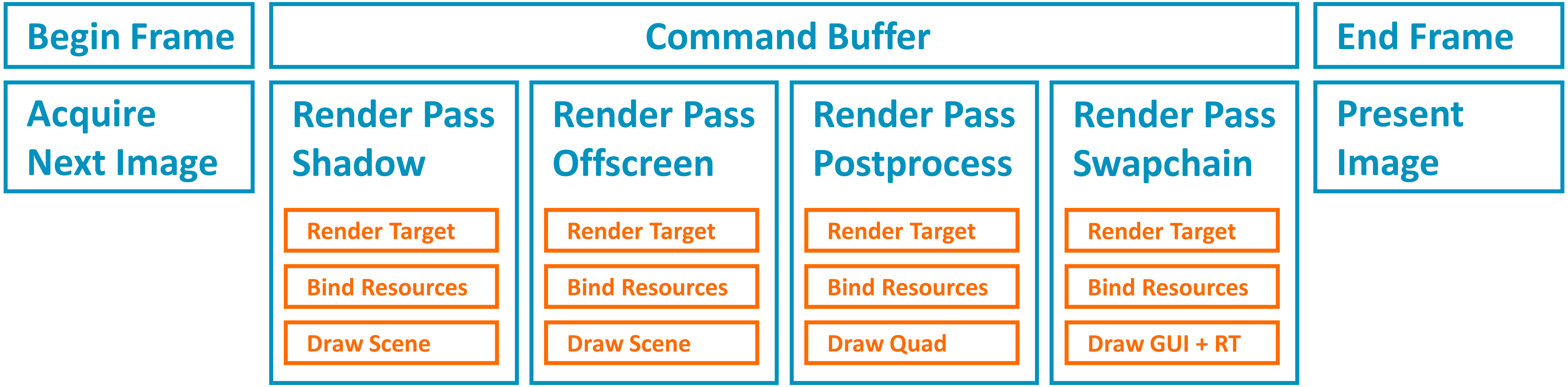
SPIRV Reflection

- Generated resource
- Object/Dependency
- Application defined

High-Level API



High-Level API



Tools

Streamline

Performance Analyzer

Speed up your app

- Find out where the system is spending the most time
- Tune code for cache efficiency

Tune your rendering

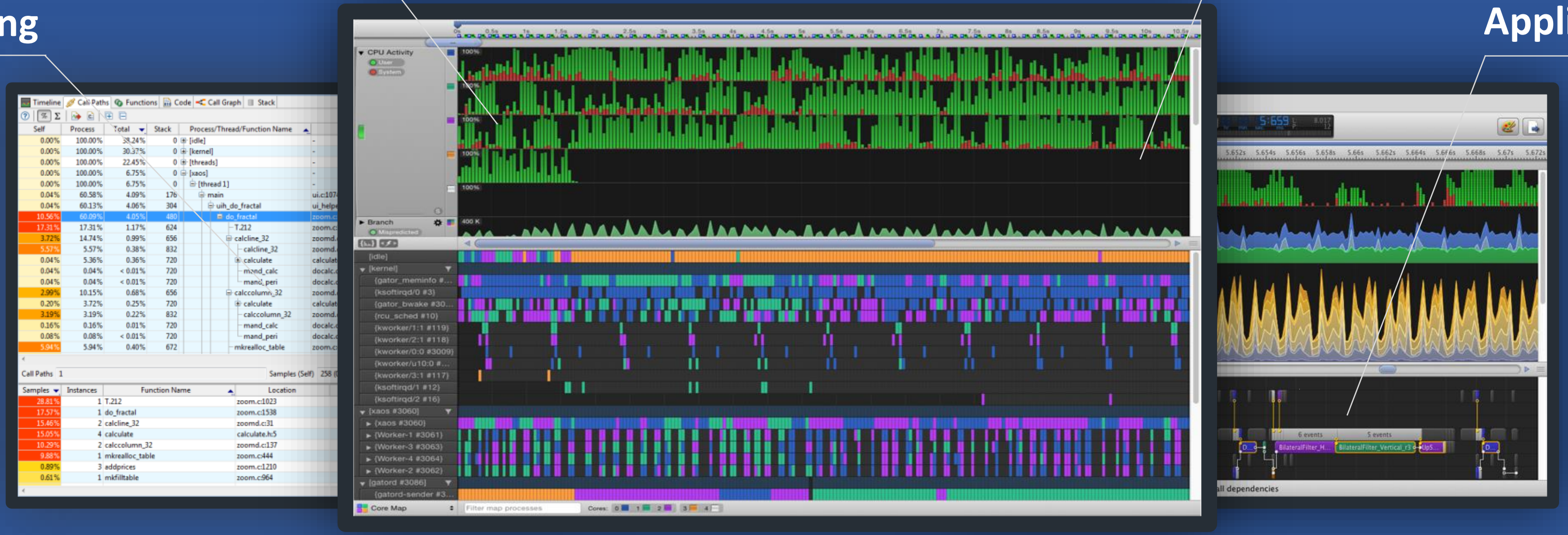
- Identify critical-path GPU shader core resources
- Detect content inefficiency

Native code profiling

- Break performance down by function
- View cost alongside disassembly listing

Application event trace

- Annotate software workloads
- Define logical event channel structure
- Trace cross-channel task dependencies



Arm CPU support

- Profile 32-bit and 64-bit apps for ARMv7-A and ARMv8-A cores
- Tune multi-threading for DynamIQ multi-core systems

Mali GPU support

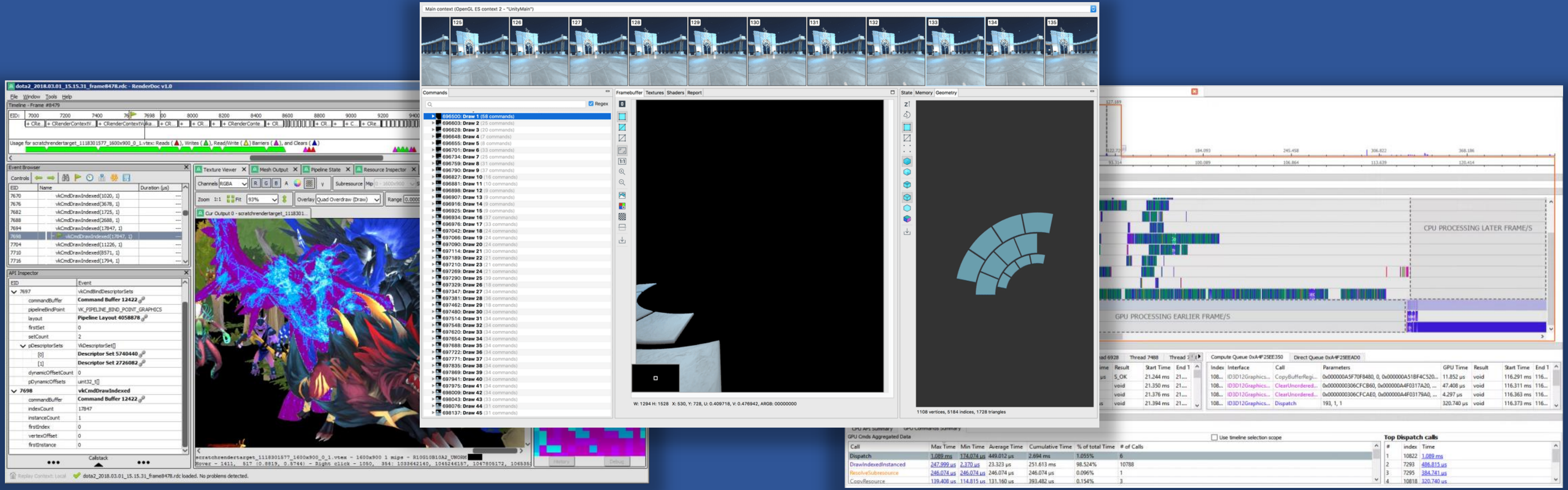
- Analyze and optimize Mali™ GPU graphics and compute workloads
- Accelerate your workflow using built-in analysis templates

Optimize for energy

- Move beyond simple frame time and FPS tracking
- Monitor overall usage of processor cycles and memory bandwidth

Debuggers

RenderDoc, GAPID, and CodeXL



RenderDoc

- Supports Windows 7, 8.x, 10, Linux, Android, and Stadia for capture and replay out of the box.
- Very Customizable, embeds the python runtime for programmatic access to frame captures.

GAPID

- Identify rendering issues, such as missing objects or object size and texture problems.
- Inspect the resources loaded by the graphics API.

CodeXL

- Support for Vulkan GLSL shaders, including ISA generation and performance statistics.
- Supports the Boltzmann driver, AMD Radeon R9 Fury, Fury X, Fury Nano GPUs, and 6th Generation AMD A-series APU processors.

Porting Roblox to Vulkan

What is Roblox?

- Online multiplayer game creation platform
- All content is user generated
- Windows, macOS, iOS, Android, Xbox One
- 100M+ MAU, 2.5M+ CCU

What is Roblox?

What is Roblox?

What is Roblox?

What is Roblox?

Why Vulkan?

- Lots of performance challenges on Android
- Need maximum performance without tweaking content
- Need modern* GAPI features for current/future rendering projects
- Long term desire to discontinue OpenGL

- We’ve investing in Vulkan for the long term
    - Performance, clear driver/hardware mental model
    - Unified shader pipeline through SPIRV
    - Potential to use on other platforms

Porting to Vulkan

- It took time!
  - Started November 2016
  - First working version March 2017
  - First working version in production November 2017
  - Fully live March 2018
- Continuous maintenance and performance tweaks ever since
- Seeing good steady adoption
  - March 2018: 17% of our Android userbase (Android 7.0+)
  - December 2018: 28% (Android 7.0+)
  - February 2019: 23% (Android 7.1+)
  - September 2019: 37% (Android 7.1+)

API pandemonium

- We did *not* rewrite the renderer to be “Vulkan-friendly”
  - D3D9, D3D11, GL 2/3, GLES 2/3, Metal, Vulkan
  - Slowly improving the common rendering interface
  - Balancing simplicity (engineers) vs performance (users)
- Clean and easy to use immediate-mode abstraction
  - Directly targets the given API without extra wrappers (e.g. MoltenVK)
  - Maximum performance within the interface constraints
- Features specific to a given API cost more
  - Can’t automatically benefit on other APIs / platforms
  - Work well if they can be implemented cleanly behind the abstraction

Incremental refactoring

- Evolving immediate-mode abstraction over time (since D3D9!)
- Many changes during Metal port, aged reasonably well with Vulkan

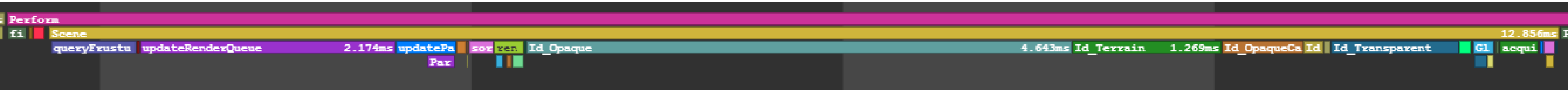
```
PassClear passClear;
passClear.mask = Framebuffer::Mask_Color0;

ctx->beginPass(fb, 0, Framebuffer::Mask_Color0, &passClear);
ctx->bindProgram(program.get());
ctx->bindBuffer(0, globalDataBuffer.get());
ctx->bindBufferData(1, &params, sizeof(params));
ctx->bindTexture(0, lightMap, SamplerState::Filter_Linear);
ctx->draw(geometry, Geometry::Primitive_Triangles, 0, count);
ctx->endPass();
```

ROBLOX

“It’s hard to beat the driver”

- A study in tradeoffs
- Seeing great performance despite not being 100% Vulkan-friendly
  - Faster CPU dispatch
  - Matching (D3D) or exceeding (GLES) GPU performance
- Seeing 2x-3x CPU performance gains across all vendors
  - End-to-end render frame, real-world contents
- Mobile test level @ 840 draw calls, single core
  - 2.4 GHz Cortex-A73, Mali-G72
  - GLES: 38 ms 😱
  - Vulkan: 13 ms



Best practices
through the lens
of perf/cost tradeoffs

Render passes

- Many complex topics in one
  - Load/store actions
  - Image layout transitions
  - Pipeline barriers
- Automatic vs manual tracking?

- ARM Tutorials: “Appropriate use of render pass attachments”, “Render Subpasses”

Render passes: immediate-mode frame structure

- We explicitly bracket all draw calls into passes
- Specify all information in beginPass() precisely
  - A full set of textures to render to (color/depth)
  - Which framebuffer textures need to be loaded from memory?
  - Which framebuffer textures need to be stored to memory?
  - Which framebuffer textures need to be cleared with what initial data?
  - Do we need to do MSAA resolve in endPass() and if so, where?
- Lazily create/cache VkRenderPass / VkFramebuffer with optimal setup

```
PassClear clear;
clear.mask = Framebuffer::Mask_Color0 | Framebuffer::Mask_Depth;
clear.depth = 1.0f;

PassResolve passResolve;
passResolve.mask = Framebuffer::Mask_Color0;
passResolve.target = shadowMap.get();
context->beginPass(shadowMapMSAA.get(), 0, 0, &clear, &passResolve);
```

ROBLOX

Render passes: load/store actions

- Avoid excessive memory bandwidth for tilers when loading/storing RT data
  - This is implicit in OpenGL(ES), guided by glClear / glDiscardFramebuffer
  - We specify this *explicitly* for *every* render pass
- If you need to clear the target, specify clear action/data
  - Do NOT use vkCmdClearColorImage/etc.!
- Examples:
  - During main 3D scene pass, we don’t need to load color/depth (use clear instead)
  - During main 3D scene pass, we don’t need to store depth
  - During post-processing passes, we don’t need to clear or load color attachment
    - It’s going to be overwritten with a full-screen triangle anyway – use LOAD_OP_DONT_CARE

Render passes: image layout transitions

- We use the concept of “default” resource state
  - For each texture we know what layout it’s “expected” to be in between passes
  - For textures with shader access this is SHADER_READ_ONLY
  - For textures without shader access this is COLOR_ATTACHMENT_OPTIMAL (or DEPTH)
  - For read/write textures this is GENERAL
  - Usually frowned upon in DX12, works well for us
- All image layout transitions are performed at the pass boundary
  - No “just in time” transitions!
- All image layout transitions are guided by load/store masks
  - An image that is not loaded is transitioned from UNDEFINED to COLOR_ATTACHMENT
    - Caveat: sometimes disables Transaction Elimination on ARM ☹
  - An image that is not stored is kept in COLOR_ATTACHMENT (or DEPTH_ATTACHMENT)
- Partially solves lack of “time travel” (we don’t have a render/frame graph)

Render passes: synchronization

- Use the same load/store metadata to infer synchronization
  - 90% optimal in our case
  - A texture that is stored is assumed to be accessed in the shader
- Important: dstStageMask=VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
  - The common case is that the render pass output is read in fragment shader
  - It's *crucial* that cases when the output is necessary in a different stage are explicit!
  - Vertex work for the subsequent pass can be scheduled before the previous pass ends
  - This is *really* important for tilers!

Render passes: MSAA resolve

- If there is one texture you should never use STORE_OP_STORE on…
  - … it's MSAA color/depth target that's technically two textures
- MSAA on mobile is wonderful when done right
  - Minimal extra shading cost
  - No extra bandwidth cost… when using pResolveAttachments
  - No extra memory cost… when using transient attachments
- Correct and fast MSAA render pass specification on mobile includes…
  - Color/depth MSAA (4 sample) target
    - loadOp = LOAD_OP_CLEAR, storeOp = STORE_OP_DONT_CARE
  - Resolved color (1 sample) texture specified with pResolveAttachments
    - loadOp = LOAD_OP_DONT_CARE, storeOp = STORE_OP_STORE
    - KHR_depth_stencil_resolve if you need depth as well
- Do *NOT* use vkCmdResolveImage

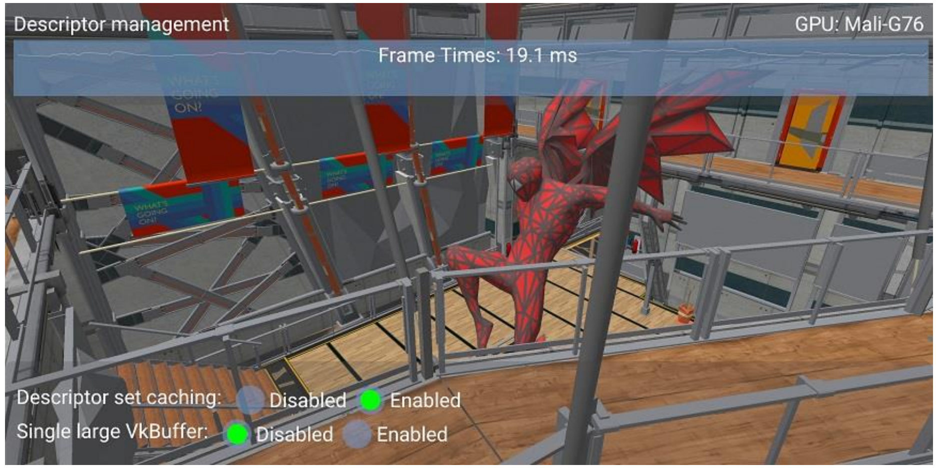
Render passes: transient attachments

- Attachments that aren't loaded/stored can be transient
  - Create image with VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
  - Allocate from memory type with VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT
- Transient lazily allocated attachments may consume no memory on tilers
  - This is especially valuable for MSAA targets
  - 720p 4x MSAA color + depth = 28 MB
  - With on-chip resolve, we never read or write this memory!
- A similar concept exists in Metal; we expose it through Texture usage
  - Usage_Render = UsageBit_Shader | UsageBit_Render
  - Usage_RenderOnly = UsageBit_Render
  - Usage_RenderMemoryless = UsageBit_Render | UsageBit_Memoryless
    - Transient render-only texture, has to get rendered in a single pass

Descriptor set management

- Resources are bound to shaders via descriptor sets
- Descriptor sets need to be…
    - Allocated
    - Updated
    - Bound
- Need an efficient management scheme for our simple interface

- ARM Tutorial: “Descriptor and buffer management”



Descriptor set management: interface

- We use slot-based binding model

```
void bindBuffer(unsigned int slot, Buffer* buffer);
void bindBufferRw(unsigned int slot, Buffer* buffer);
void bindBufferData(unsigned int slot, const void* data, unsigned int size);
void bindTexture(unsigned int slot, Texture* texture, SamplerState state);
void bindTextureRw(unsigned int slot, Texture* texture);
```

- This should look familiar and yet it’s not
  - Coupled textures and samplers (OpenGL ☹)
  - Only two namespaces, buffers and textures
    - No per-stage namespaces (constant buffer #3 is bound to the entire pipeline)
    - No difference between constant buffers and shader storage buffers
    - No difference between read-write (UAV) slots and read slots
  - An option to specify constant buffer data
- Works surprisingly well for Metal and Vulkan

Descriptor set management: implementation

- When creating the pipeline, we know ahead of time what slots each stage uses
  - This is discovered through shader reflection metadata
  - Validate compatibility between stages, e.g. uniform buffer #5 must be uniform in VS & FS
  - We build a “perfect” VkDescriptorSetLayout (denote stage usage with stageFlags)
- Note that we use at most 2 sets!
  - Buffers and textures
  - The guaranteed limit is 4 – fitting “into” the limit is a problem for many Direct3D1x ports
  - Some mobile hardware only supports a single combined set in hardware anyway
- All bindResource calls just update dirty masks and resource info
- Before each draw/dispatch we lazily allocate/update descriptor sets

Descriptor set management: allocation

- Descriptor sets are allocated out of pools
- We use a ring buffer of pools
  - If the current pool has space, allocate a descriptor in this pool (free-threaded)
  - Otherwise, get a pool out of the global “pool of pools” (requires a lock)
  - The pools get recycled using deferred reclamation (same as deferred destruction)
- Configuring the pools is not trivial!
  - For each pool, need to specify the number of sets and the number of resources
  - How do you pick the ratio?

Descriptor set management: allocation policy

- A pool per shader pipeline object
  - We know the number of textures/buffers each pipeline uses, can configure pools optimally
  - E.g. shadow map opaque pipeline: 1024 sets, 0 textures, 2*1024 buffers
  - E.g. scene opaque pipeline: 1024 sets, 8*1024 textures, 3*1024 buffers
  - A lot of space wasted on rarely used pipelines (postfx), more expensive to switch pipelines
- One type of pool, configured using worst-case descriptor count
  - E.g. one VkDescriptorPool has 1024 sets, 16*1024 textures, 8*1024 buffers
  - Simple – just one type of pool!
  - A lot of space wasted because the ratio of sets:textures:buffers varies
- Settled on one type of pool, configured for “average” usecase
  - sets:textures:buffers ratios determined by collecting data on typical levels
  - Simple, little space wasted in common case
  - Non-trivial space savings – tens of megabytes on moderate levels

ROBLOX

Descriptor set management: binding

- If any resources changed, allocate and update a new descriptor set
  - If textures changed but buffers didn't, only need one set, not two
- Note: when shader pipeline changes, sometimes don't need to rebind sets
  - See "Pipeline Layout Compatibility" section of Vulkan specification
  - For us this reduces the number of buffer descriptors we need by ~50% on complex scenes
- Do not use descriptor set copying for partial updates!
  - Faster to rewrite the entire descriptor set from scratch
- Optional: use descriptor templates from Vulkan 1.1 to reduce CPU cost
  - We do this – can be faster on some desktop drivers
- Optional: can cache descriptor sets between non-consecutive draw calls
  - We don't do this – doesn't happen that often, and adds complexity
- Important: use dynamic buffer offsets!

Descriptor set management: constant data update

- Most of our per-frame constant data is small and dynamic
- We sub-allocate it from a large buffer
  - bindBufferData() allocates from a 512 KB uniform buffer using bump pointer allocation
  - If we have more than 512 KB of uniform data per command buffer, allocate multiple buffers
- Instead of allocating a new buffer descriptor every time, use pDynamicOffsets
- Dramatically reduces number of buffer descriptors and improves performance

```
VKAPI_ATTR void VKAPI_CALL vkCmdBindDescriptorSets(
        VkCommandBuffer             commandBuffer,
        VkPipelineBindPoint         pipelineBindPoint,
        VkPipelineLayout            layout,
        uint32_t                    firstSet,
        uint32_t                    descriptorSetCount,
        const VkDescriptorSet*      pDescriptorSets,
        uint32_t                    dynamicOffsetCount,
        const uint32_t*             pDynamicOffsets);
```

ROBLOX

Descriptor set management: constant data update tradeoff

- This implementation leads to dispatch cost tradeoffs…

```
void bindBuffer(unsigned int slot, Buffer* buffer);
void bindBufferData(unsigned int slot, const void* data, unsigned int size);
```

- Do you pre-upload the uniform buffer data or use bindBufferData?
  - bindBufferData has to memcpy into the large buffer – bad!
  - bindBufferData doesn't need a new buffer descriptor – good!
- In practice, the choice is usually obvious
  - bindBufferData for one-off constant values – frequent!
  - bindBuffer for constant data that's used across many/most draw calls – rare!

General performance tuning

- The driver is much slimmer than a typical GL driver
  - This surfaces things that were trivial/unnoticeable before!
- Don’t call vk* functions unless you need to
  - Especially important for creating objects – we cache everything we can
  - Still faster to do state filtering (vkCmdBind*) in your code
- Aggressively eliminate cache misses
  - Reduce allocations and indirections in your abstraction
- Aggressively eliminate contention
  - Use “pool of pools” for any resource caches
  - Use lock-free read / locked write cache for pipeline states
- Call most functions through pointers obtained via vkGetDeviceProcAddr
  - volk (github.com/zeux/volk) loader does this for us; a few % wins on some drivers

Conclusion

- Getting good performance out of Vulkan is easy*!
    - This doesn’t necessarily require a renderer redesign
    - We target 5 graphics APIs and 4 major OpenGL version from the same code
- A lot of the performance advice is cross-platform/vendor
- When in doubt:
    - Read vendor performance guides
    - Use vendor-provided samples
    - Profile!

Thank you!