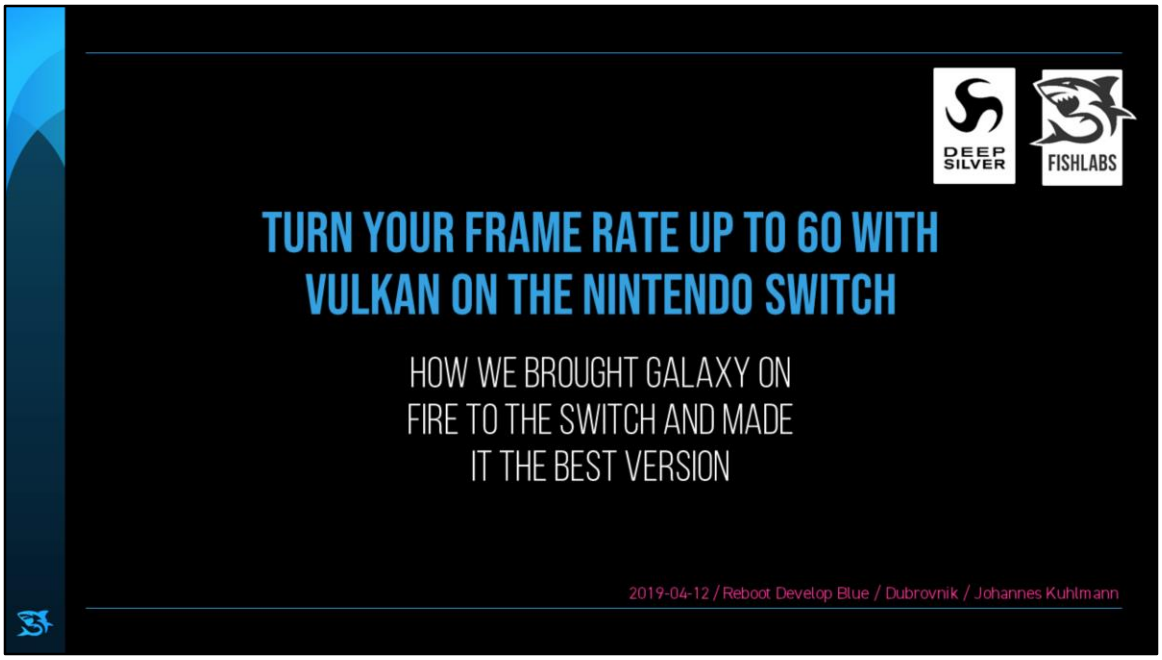
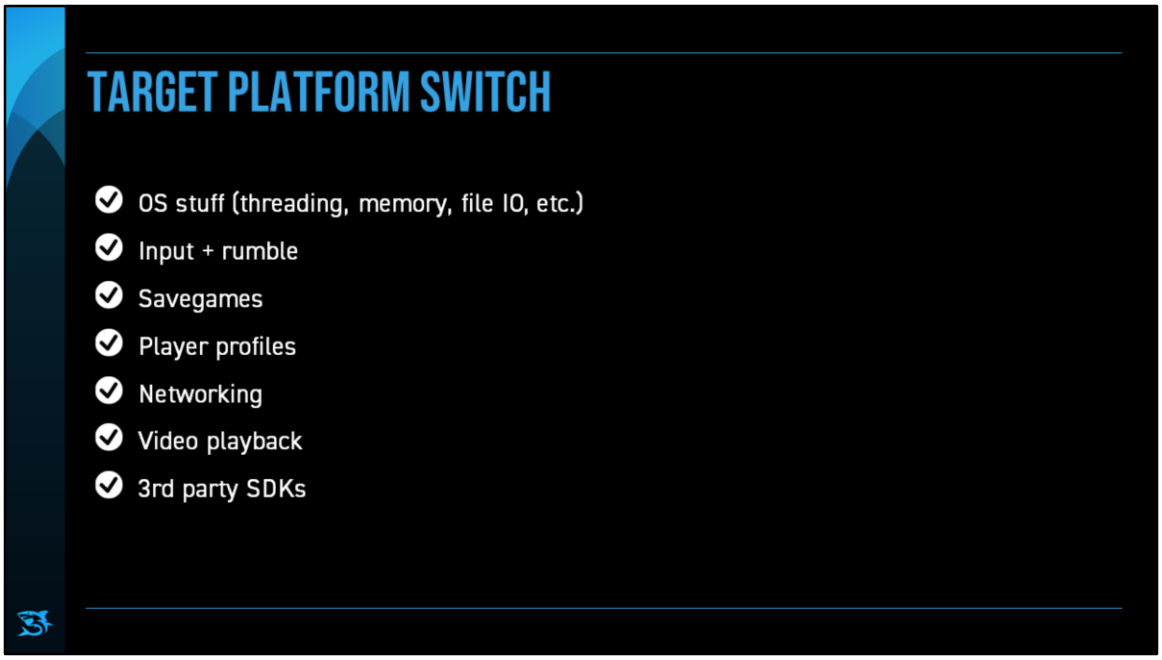

DEEP SILVER
FISHLABS
TURN YOUR FRAME RATE UP TO 60 WITH VULKAN ON THE NINTENDO SWITCH
HOW WE BROUGHT GALAXY ON FIRE TO THE SWITCH AND MADE IT THE BEST VERSION
2019-04-12 / Reboot Develop Blue / Dubrovnik / Johannes Kuhlmann

This is the Switch
Can play on small screen and also connect to TV
Selling very well
--
https://www.nintendo.co.jp/ir/en/finance/hard_soft/

Our first game for Switch: Manticore – Galaxy on Fire.

First released on iOS (Metal), and Android (Vulkan, also OpenGL ES with reduced visual quality).

Knew from experience, that renderer/graphics API biggest time sink when bringing a game to a new platform.
So, very happy, Switch supports OpenGL + native API + Vulkan.
As we already had the Vulkan renderer, the decision was easy to go with that for the Switch version.



When porting a game to a new platform, you have these usual suspects.

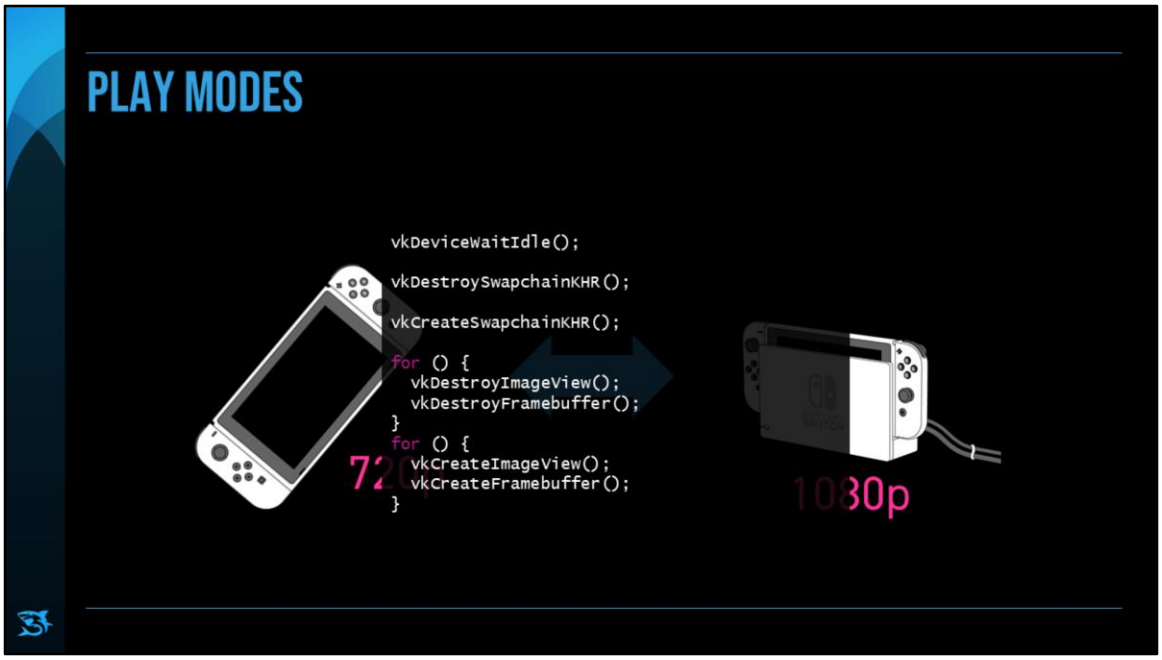
All straight-forward + quickly done (a bit exaggerating…).
Had the game running on the Switch after roughly 1 month (1 person).

But didn't just want to do a simple port; wanted to do the best version of GOF3.

f2p to premium, game pad inputs in menus, lots of other gameplay & mission tweaks.

correct surface extension and DONE

--
https://www.khronos.org/conformance/adopters/conformant-products#submission_310

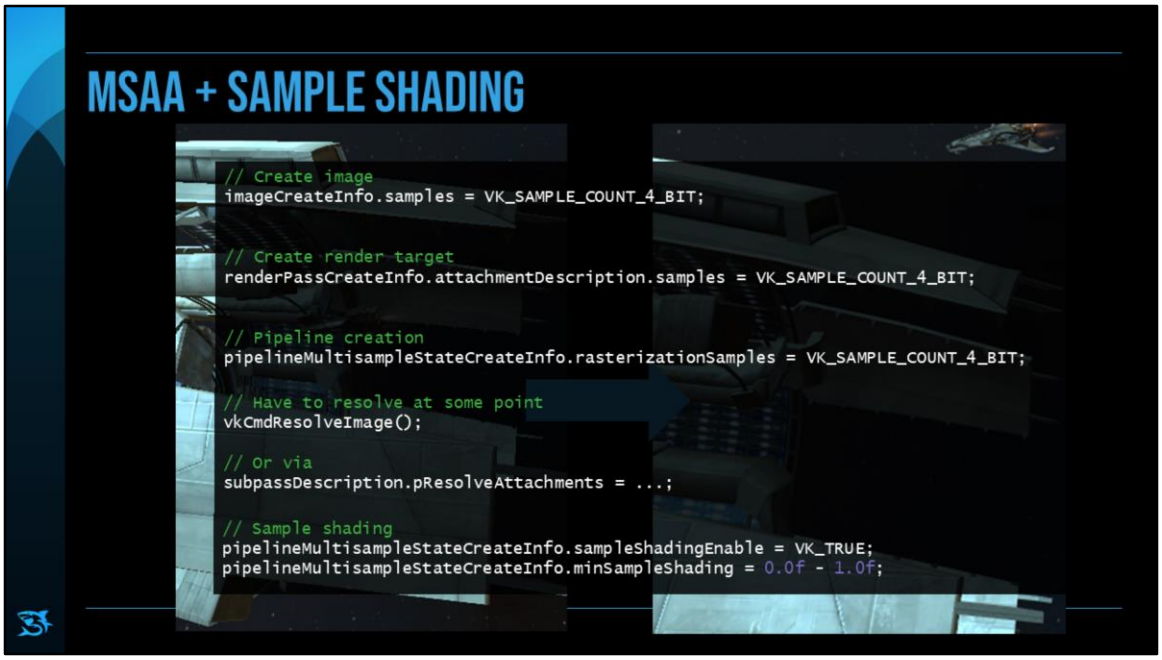


Let's support the Switch's No. 1 USP

Undocked: play anywhere -> 720p
Docked: on big screen -> 1080p
GPU runs faster in docked mode.

No graphical corruption allowed
Engine + game must be prepared to change resolution at any time
(Was not the case for us in the beginning (mobile normally has a fixed resolution))

Swapchain + framebuffers (+ render targets) need to be resized.



We implemented some easy-to-do visual improvements.

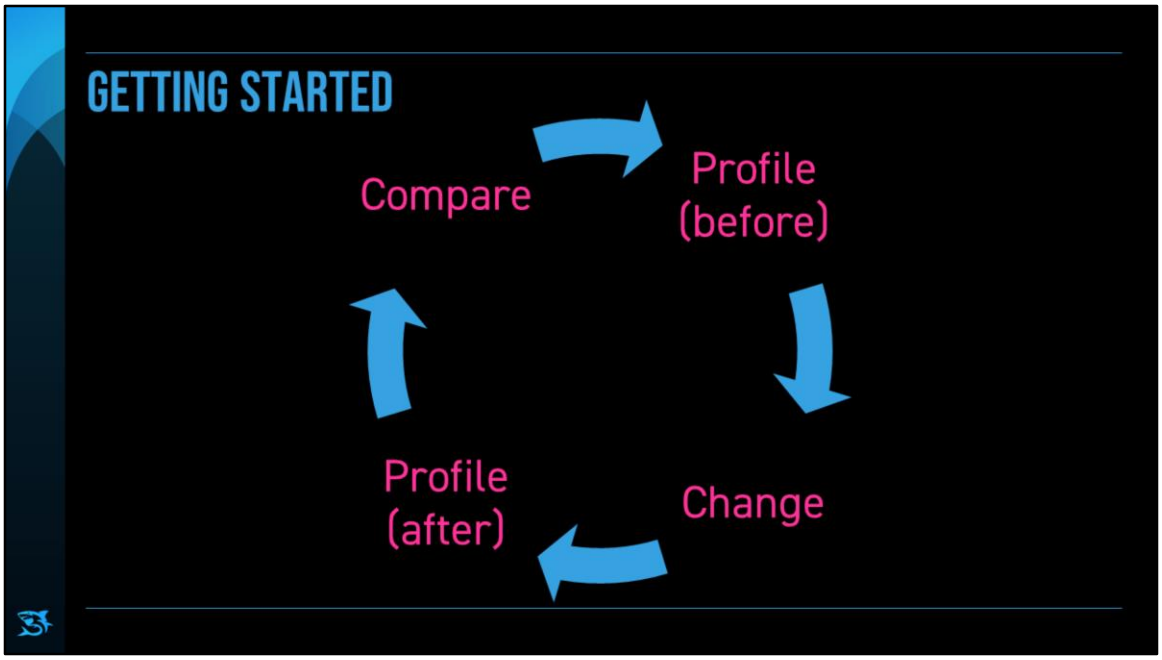
First, check if it’s supported.

First, check if it’s supported

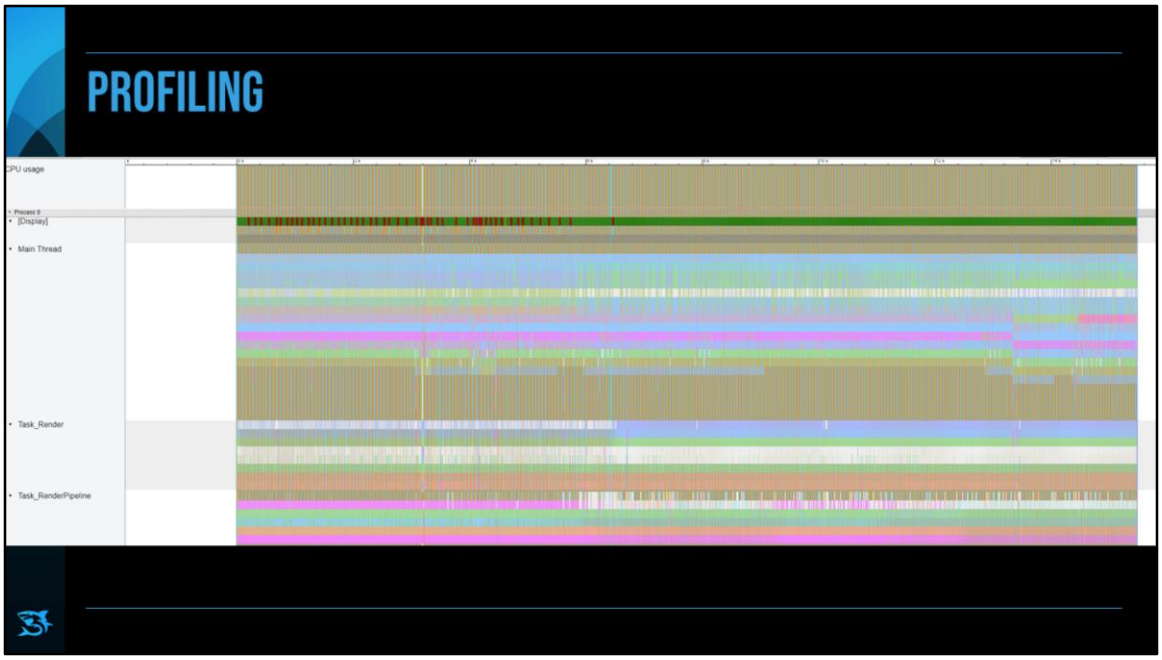
Game runs on Switch and looks great.
THE END.
Or not?

Focused on visual quality over higher performance. Always targeted 30 FPS.
But, many scenes now ran at 60 FPS.

So, we decided to ship constant/stable 60 FPS.
But, then started noticing frame drops in battle scenes & later levels.



Never ever optimize anything without measuring first.



Our custom profiling solution: instrumented, JSON
View in Chrome Tracing View
Can easily see slow frames.

There are also Open Source solutions. -> MicroProfile, Remotery

For CPU, you'd have a profiler from you platform which are mostly sampling profilers.

Used a combination of ours + platform's.

Let’s look at some optimizations that are not necessarily related to Vulkan.

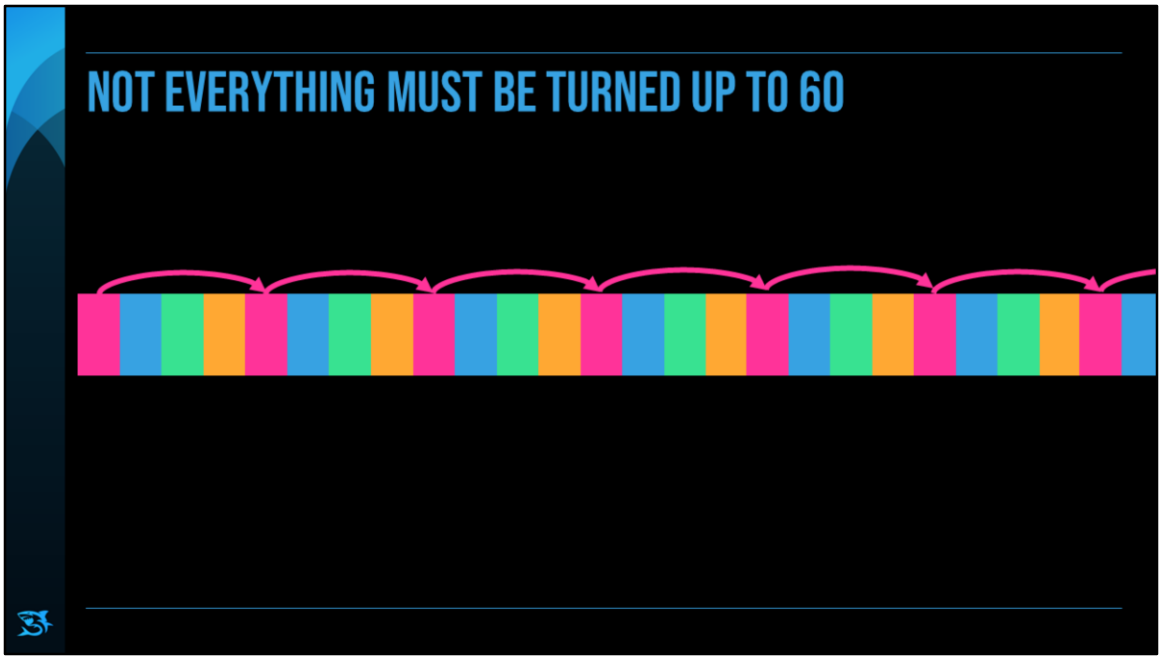
Vulkan instrumental to achieve high framerate
But need a solid + performant basis

This is a pretty big one.
No new thing to us. But drill down to the instruction level, loads the most expensive instructions?!

Had an ECS which was a very good head-start.
Data from two components difficult to combine.
Optimization: Copy object world transform into render component

3rd party Particle SDK: three passes over linked list. Got rid of two of those at least.

Small map / hash map: Same interface, just a vector linear search



Not everything has to run at 60. May work better to not run every 16.6 ms.

Handle a fixed fraction of objects each frame. That keeps the load equal per frame.

Some good systems to start looking into:
AI
- perception
- Following other objects
- Just the aiming

update frequency depend on the distance to the player
singleplayer only
But in practice it can work very well
Example: enemy ship movement

Profile!
Logic to determine update set can take longer than just doing it

(Had mesh LODs already)

Reduce particle emission rate based on distance. Mainly for ship/missile trails.

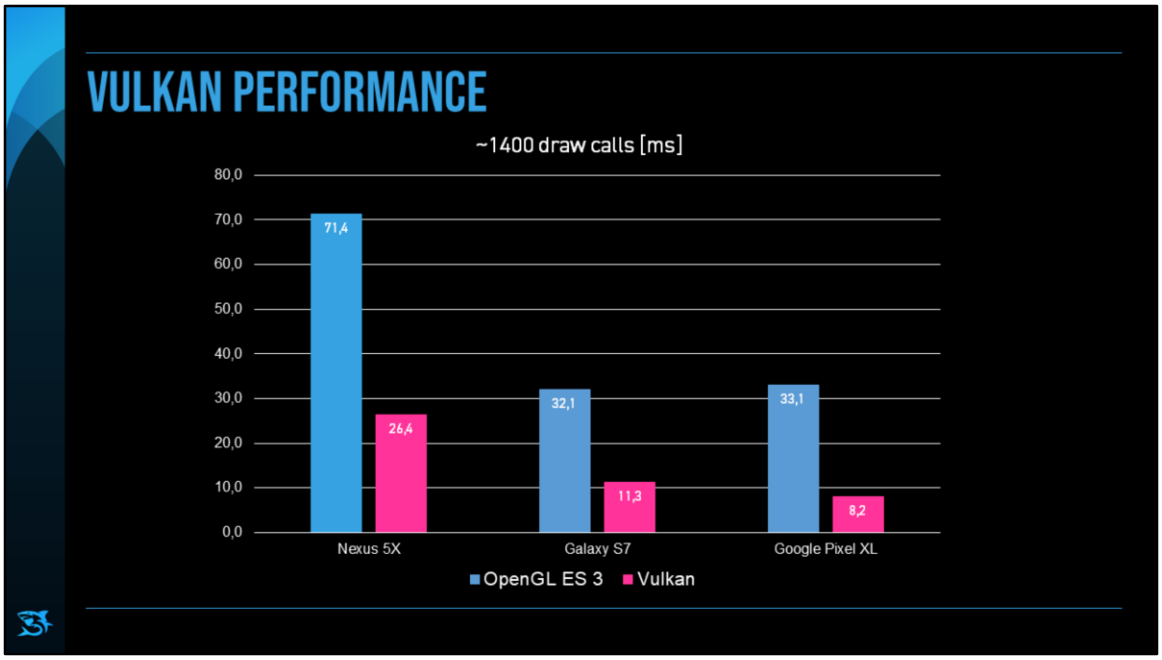
Bigger steps, but still looks nice from afar!

Reduce quality when missing performance targets.

Can be CPU or GPU optimization. In our case here, mainly a GPU optimization.

Careful in loading screens. Need upper and lower bounds.

Let’s look at some optimization related to Vulkan.



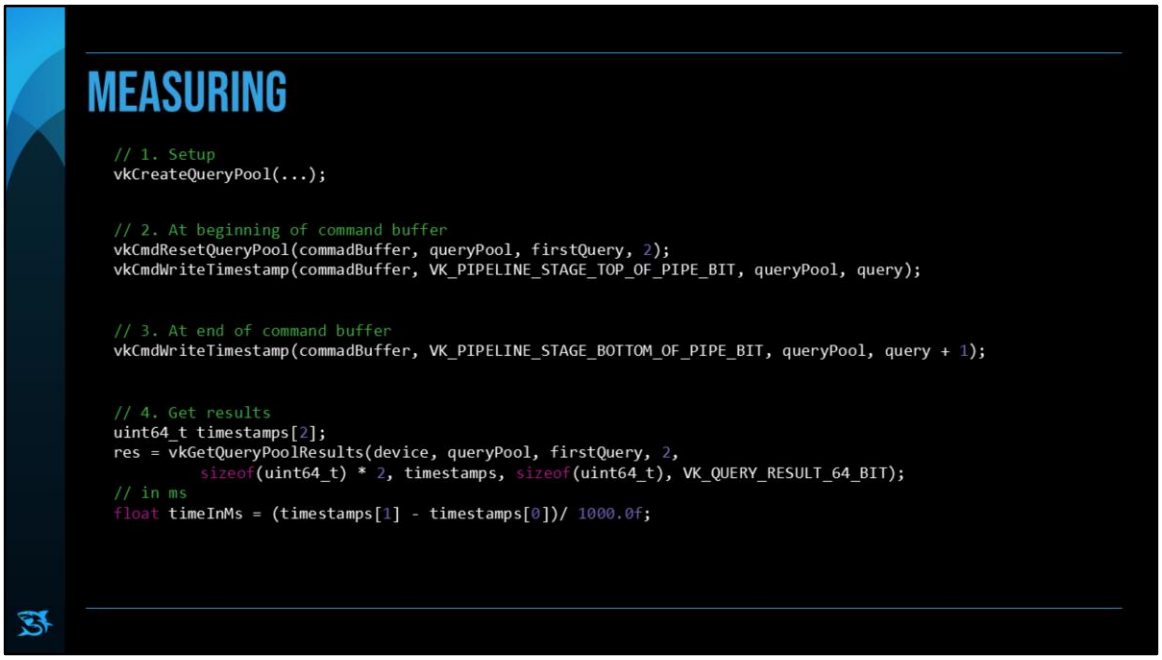
Look at what Vulkan brings to the table

Didn't have OGL on Switch, but this is from Android (2017).

3x to 4x speed-up
So, if you're aiming for 60 fps it's definitely an important part

VULKAN PERFORMANCE BASICS CHECKLIST

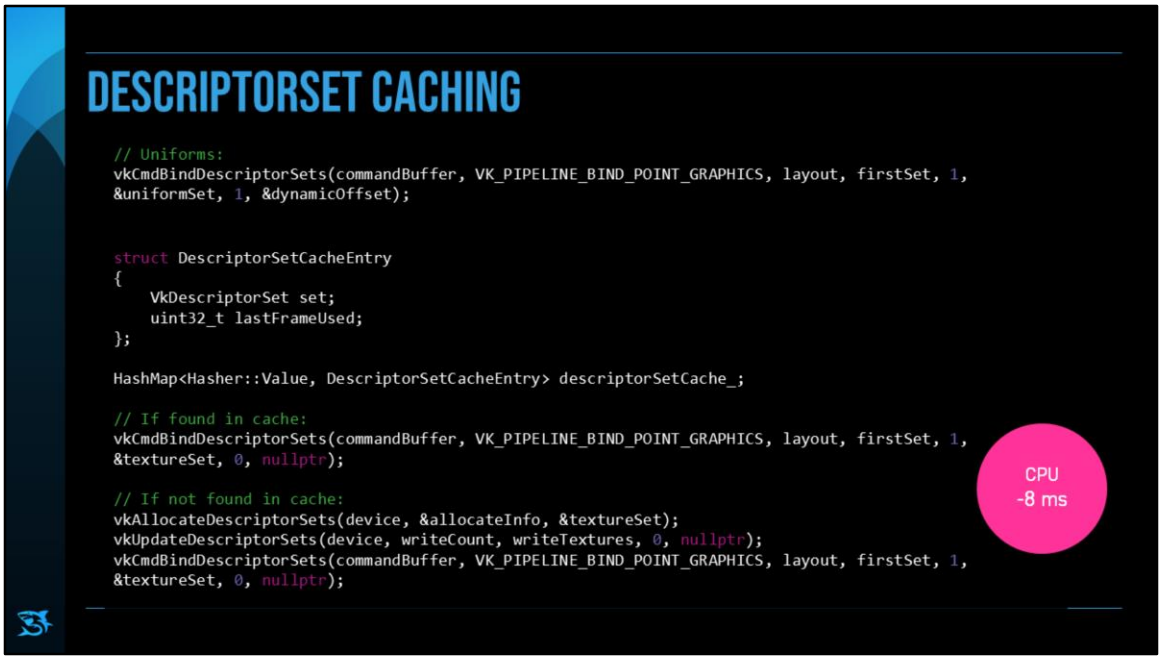
- No `vkDeviceWaitIdle()` or `vkQueueWaitIdle()`
- Textures in
  - `VK_IMAGE_TILING_OPTIMAL`
  - `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`
- Group DescriptorSets by frequency



Measure time spent on GPU

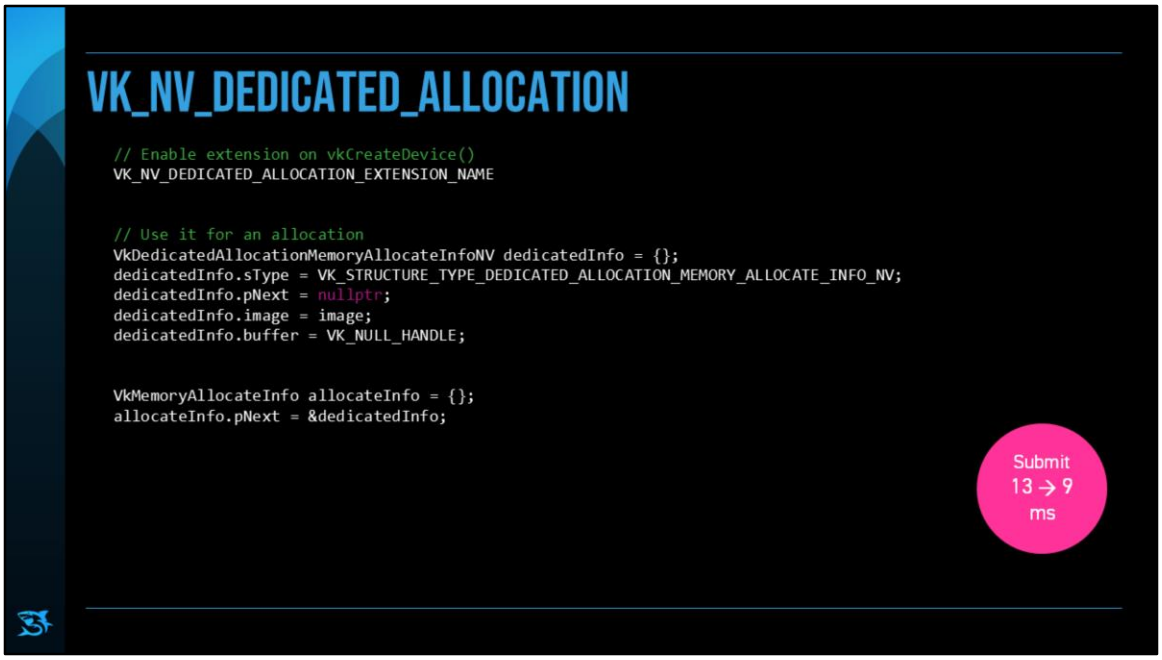
Can also use GPU debugger.
Can use Renderdoc on Switch now.

--
https://renderdoc.org/



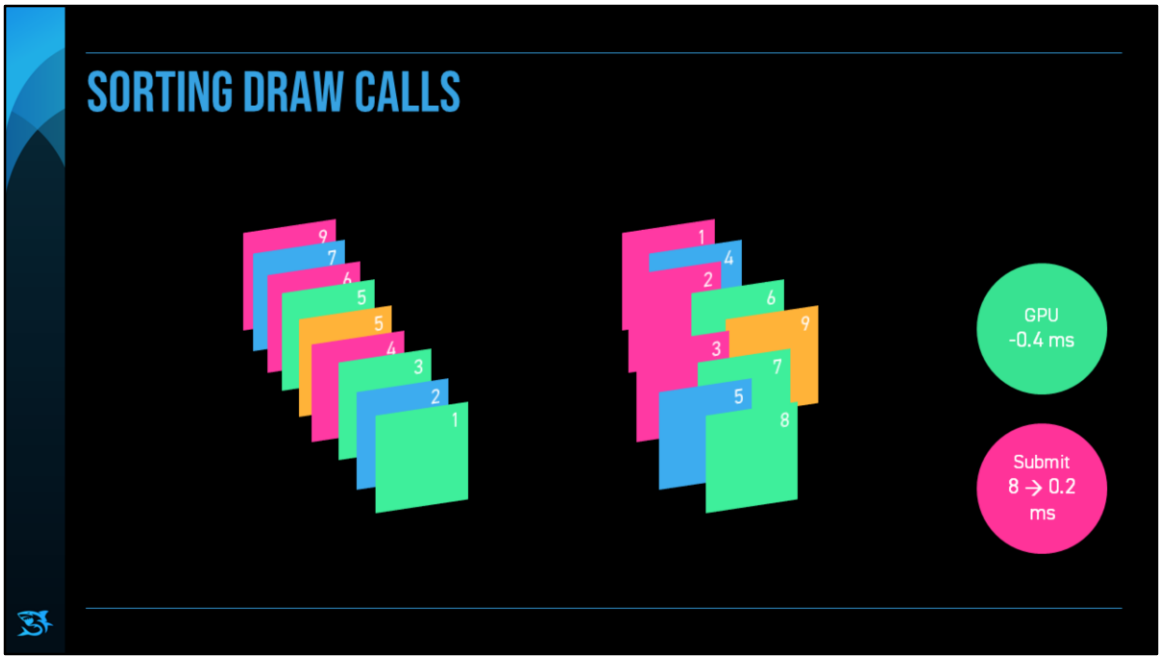
Uniforms can be handled rather easily. Use one big buffer + dynamic offset. That way we can just reuse the DescriptorSets.

Textures more complicated. Always updating DescriptorSets is very expensive. So we cache them.
Hash image + imageview + storage + sampler.



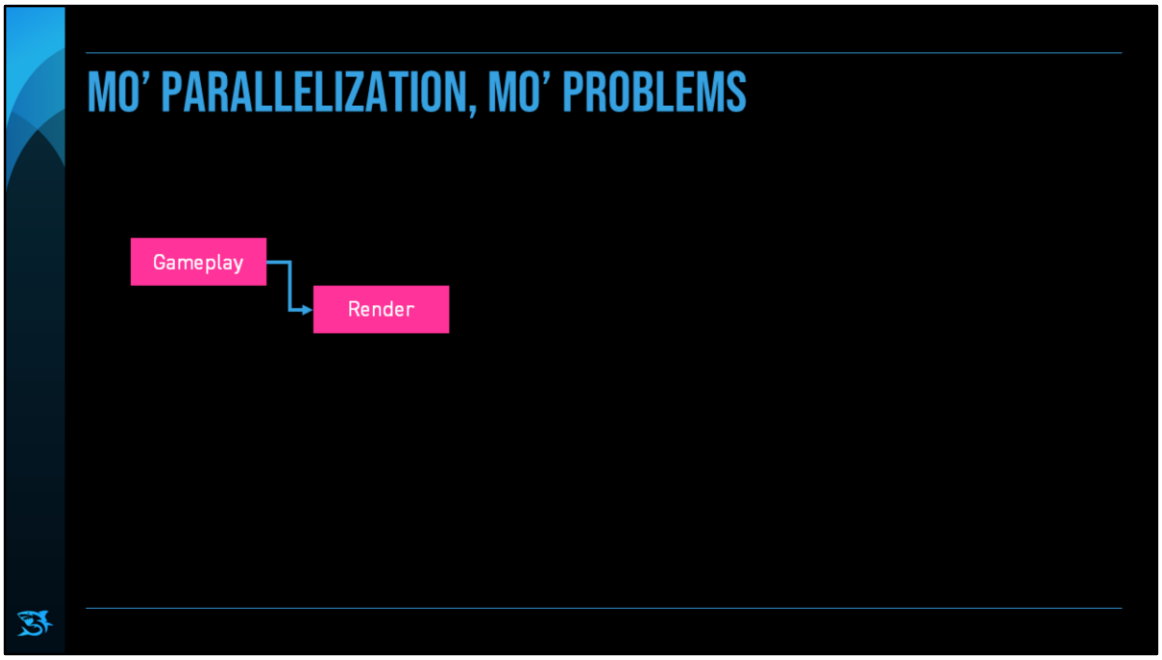
"Normal device memory allocations must support memory aliasing and sparsebinding, which could interfere with optimizations like framebuffercompression or efficient page table usage."

--
https://github.com/KhronosGroup/Vulkan-Docs/blob/1.0/doc/specs/vulkan/appendices/VK_NV_dedicated_allocation.txt



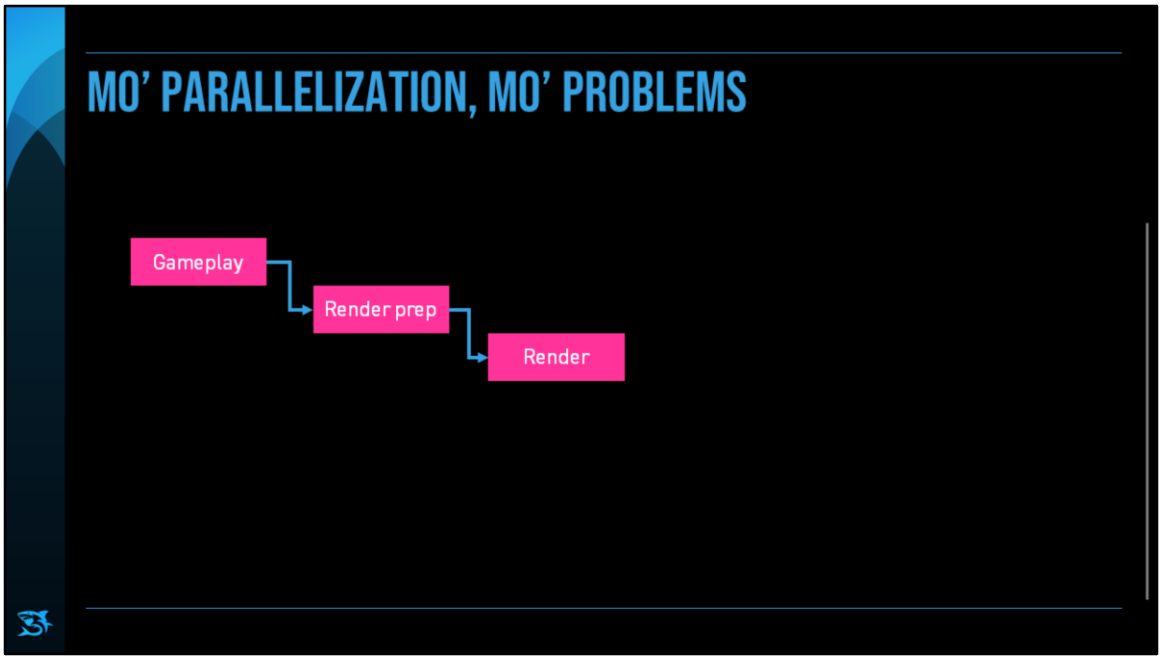
Sorting draw call can be important for perf -> confer Other APIs.
Transparent draw calls have to be ordered by depth anyway.

For opaque draw calls, the visual results doesn't depend on draw order.
Sorting by pipeline state gave us the best performance improvement.

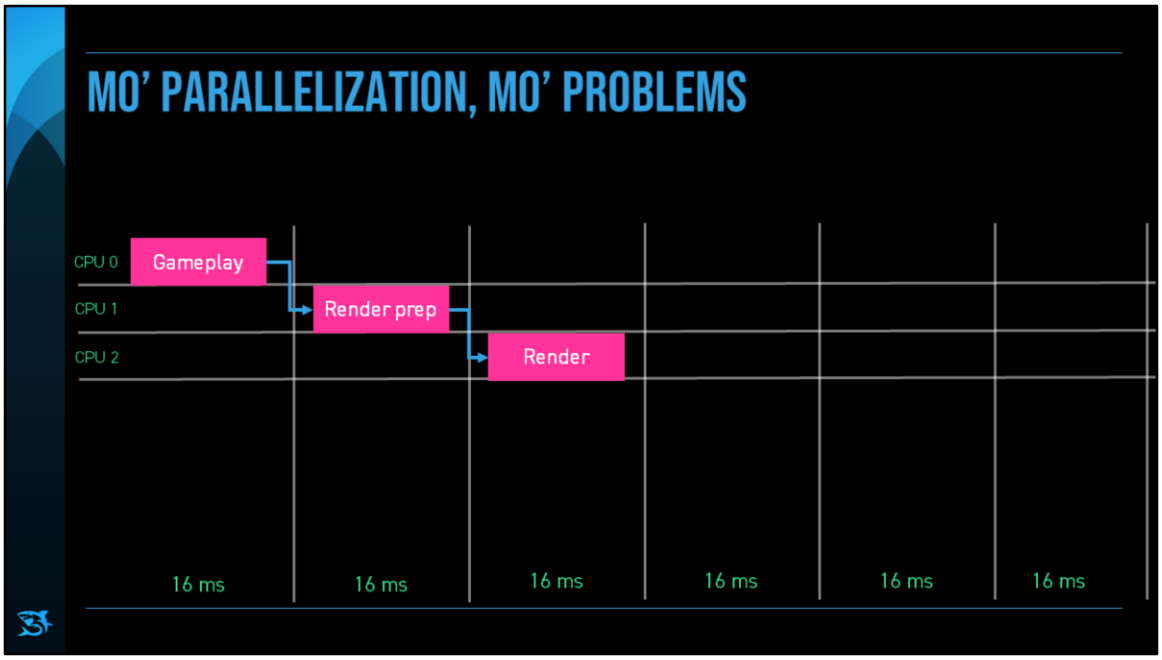


Biggest help in achieving 60.

Already decoupled: gameplay + render. During profiling, we found another big, parallelizable chunk.

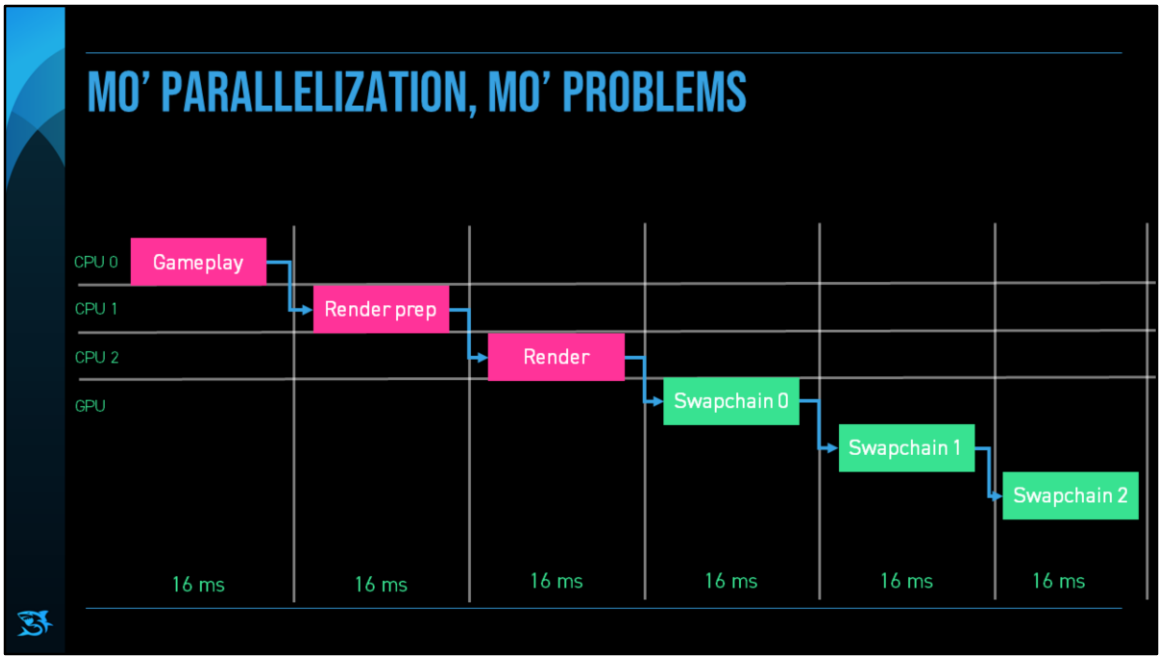


So we now have:
Gameplay
Render Prep: uniform calculations, sorting, culling, selection of LODs, etc.
Render: actual Vulkan draw calls and pipeline setting, etc.



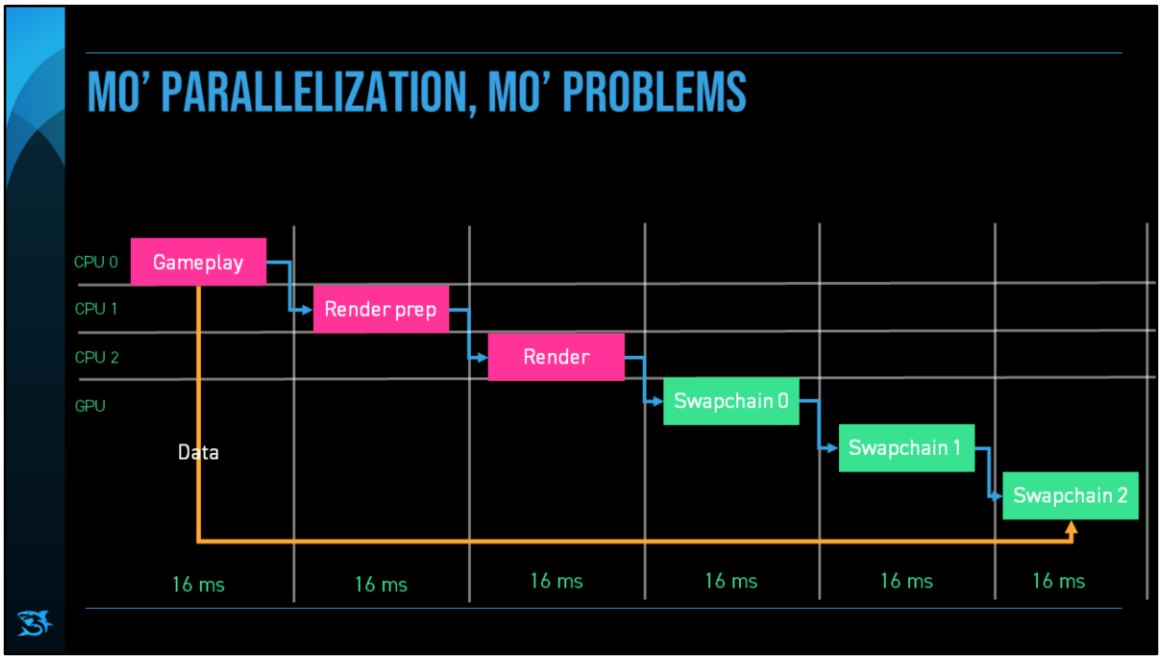
Let's parallelize that
Now at 50 ms time for logic on CPU.

Low-prio Bg threads: loading data, pipelines, audio



Another processor: GPU
Parallel to CPU and is itself a massively parallel device; works on multiple frames at once.
Always try to submit commands to the GPU early and if possible multiple times per frame. So that it can start working earlier.

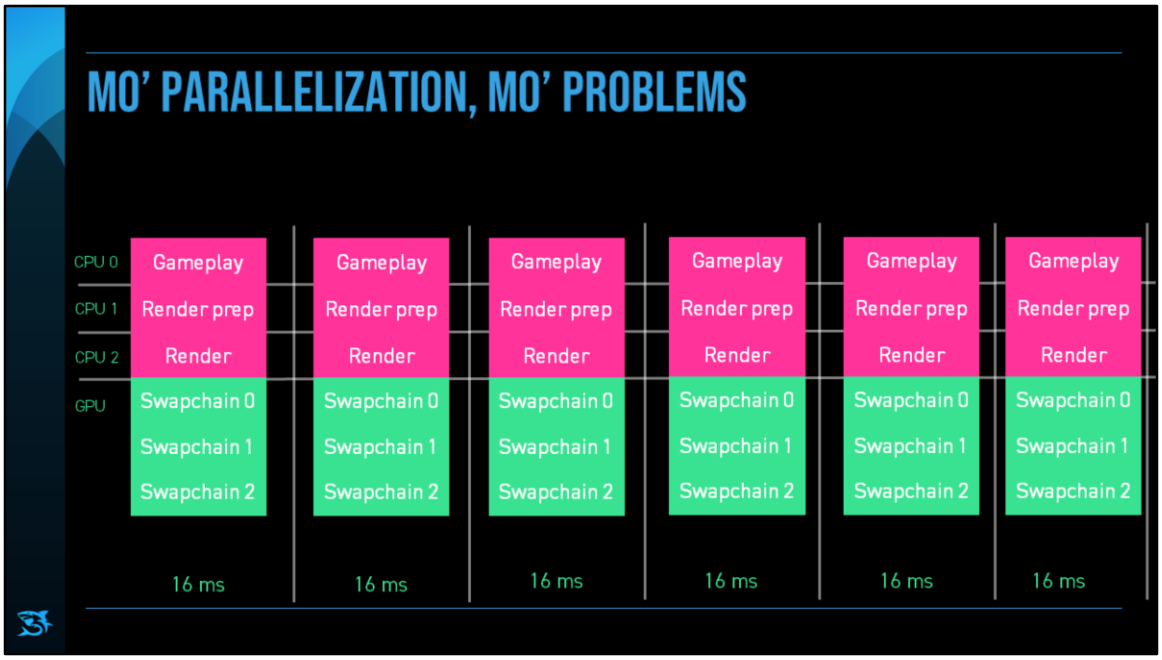
Still have only 16.6 ms time on the GPU per frame.

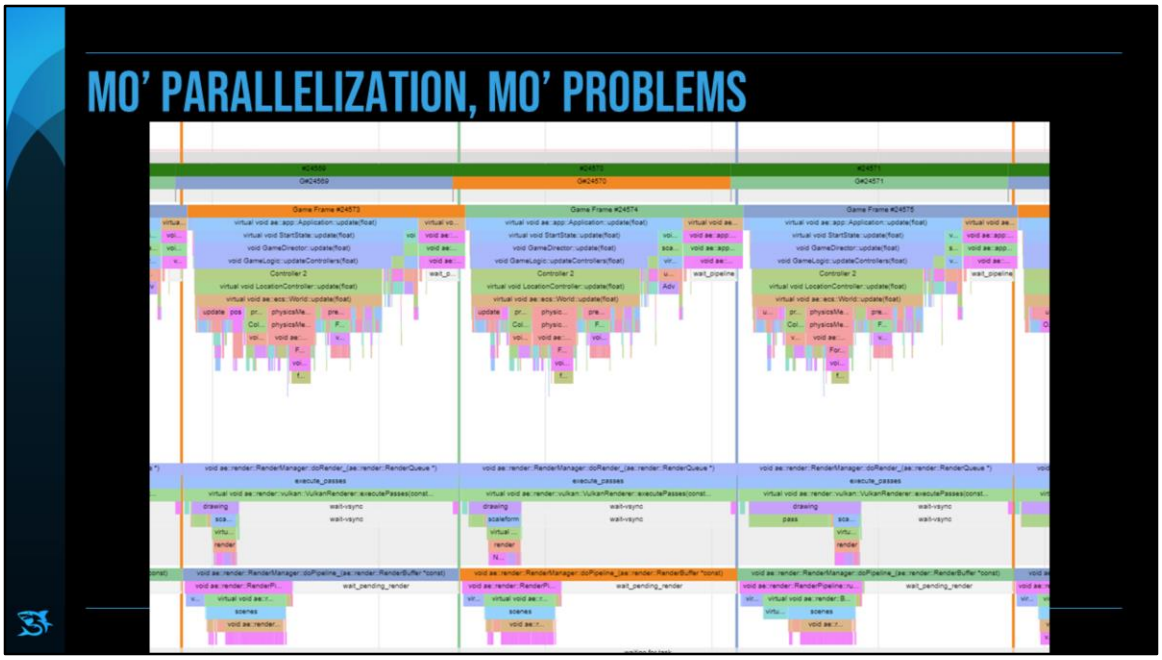


Make sure all data is available as long as it's required.
Example: ship transform.
-> Copy or preserve data

This has worked very well for us… but it could be better
Limited in cores you can support and also long chain of deps

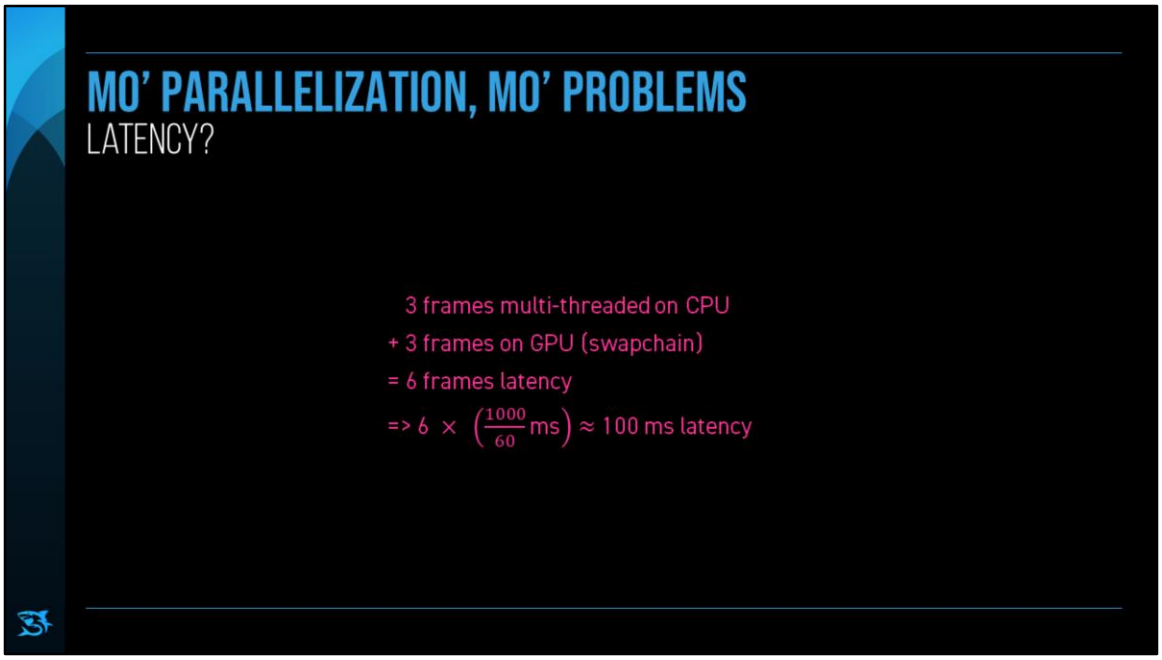
Instead go wide per task. But how to implement that is left as exercise to you attendees.


MO' PARALLELIZATION, MO' PROBLEMS
CPU 0
CPU 1
CPU 2
GPU
Gameplay
Render prep
Render
Swapchain 0
Swapchain 1
Swapchain 2
16 ms
16 ms
16 ms
16 ms
16 ms
16 ms

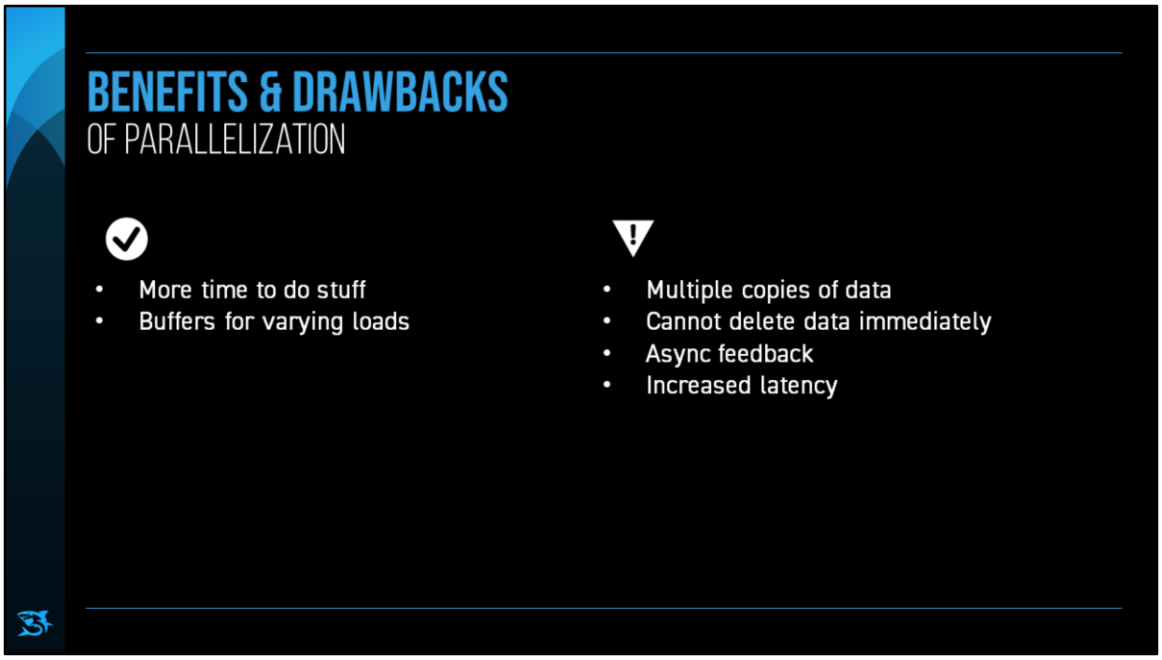


How it looks in our profiling tool

Can see the three distinct workloads.
Grey areas are where we're waiting. So in this case there's still some room.



Button press -> result on screen



Copies for: particles, transforms, other uniform, changing of assigned materials/shaders.
Keep data around until GPU is done: textures, shaders, meshes, etc.
Asynchronous feedback for: (occlusion queries), screenshots, time measurements

Looks like we have more drawbacks, but more time is definitely worth it

AND THEN THE WORST CASE HAPPENED

SUMMARY

1. Switch is an easy target platform.
2. Vulkan fulfills the cross-platform promise.
3. Synchronize properly.

THANK YOU!

Questions?

Contact: j.kuhlmann@dsfishlabs.com
@j66k

Website: https://dsfishlabs.com/