



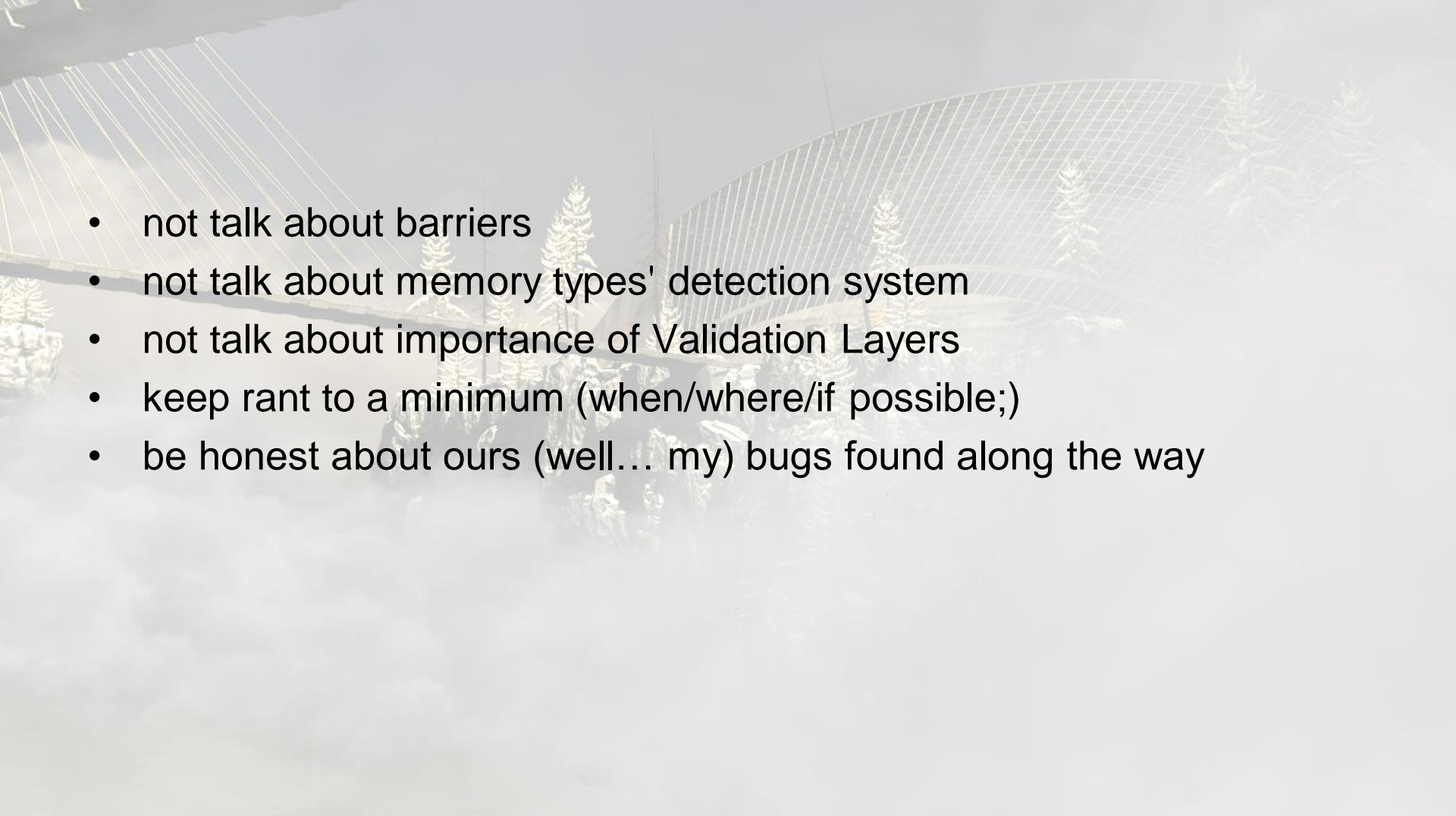
Getting Serious with Vulkan

Dean Sekulić
graphics programmer





In this presentation I will...

- 
- not talk about barriers
 - not talk about memory types' detection system
 - not talk about importance of Validation Layers
 - keep rant to a minimum (when/where/if possible;)
 - be honest about ours (well... my) bugs found along the way

I. Past

- Implemented Vulkan in The Talos Principle
- and Serious Sam Fusion (one engine to rule them all!)
- and Serious Sam VR: The Last Hope
 - (yes, VR==Virtual Reality!)

Vulkan implementation

- currently a wrapper around Direct3D 11
 - (ok, to be honest, OpenGL 2.1 and Direct3D 9)
- Support for multi-window applications
- TBDR architecture friendly
 - supports pre-clears and discards
- Supported resource creation and destruction in separate thread(s)
 - using transfer queue where available
- Performance is already really good
 - although a lot more work needs to be done

Performance

CPU side:

- ~50-100% faster than OpenGL
- ~20-30% faster than Direct3D 9
- on par with Direct3D 11
 - one of the reasons could be an overhead from our side (hashing and finding DS and PSOs!)
 - other reason is overhead in some vendors' drivers (Vulkan functions slower than D3D11 functions?!)

Performance

GPU side:

- mixed results, difficult to reach any conclusions
 - on same vendor/driver, Talos works slower than D3D11, but SamVR TLH is faster?!
- The main reason is (probably!) "handling" of uniforms in our shader system
 - designed for fast updates (CPU-side) and to support A LOT of different gfx APIs
- should be faster on TBDR GPU architectures
 - because of heavy usage of pre-clears or discards in Vulkan render passes (still not confirmed - maybe OpenGL/ES drivers already have some heuristics for this?!)
 - this can be easily disabled in our engine (via cvars) for performance testing purposes
 - tried a long time ago with OpenGL extensions, got us nowhere :(

Constant (uniform) and dynamic geometry buffers

- this is one of the things that could noticeably(!) impact Vulkan performance
- 4 methods for updating (selectable via console variable)
 - 1. update directly from system memory via `vkCmdUpdateBuffer()`
 - slow!
 - 2. have constant buffers in host/shared memory and update directly (the buffer is mapped all the time, of course)
 - can be slow for instancing data on some vendors (discrete GPUs!)
 - good for shared memory (default on Android!)
 - 3. batched updates from system memory
 - `vkCmdUpdateBuffer()` in batches when needed (mostly at the end of frame)
 - fast!
 - 4. batched copy buffer from host memory to device memory via `vkCmdCopyBuffer()`
 - fastest!
 - even a bit faster than 2. (in shared memory)
 - encounter some driver bugs in the beginning :(
 - but these were fixed a long time ago :)

BUG!

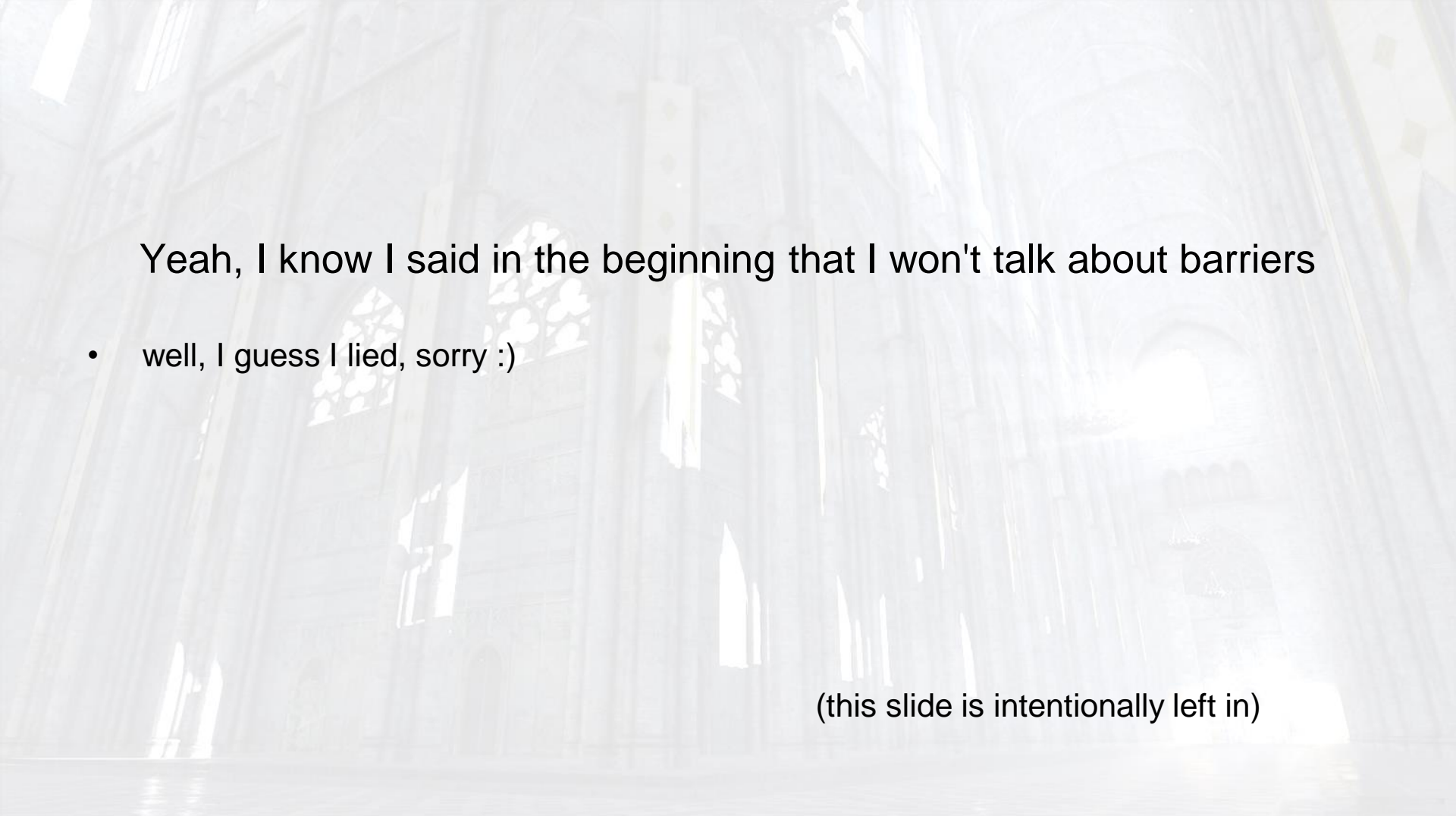
- flushing of buffer updates when spilled over!
- flush was not called, not even for valid constants so far in the buffer!
- but draw WAS!
- drawn with stalled/wrong constants' values! :O
- there are no Validation Layers in the world that can help you with this one!

RenderDoc!

- found so many overheads, like...
- redundant layout changes
 - came from suspending and resuming render-passes
- implemented batched barriers because of what I've seen in RDoc output (and I've seen A LOT, believe me!)
- Barriers' layout changes are automatically tracked and batched now
 - but this will not work with Vulkan's native multi-threaded renderer! (more on that later)

RendedDoc!

- found a lot of empty render-passes
- VLs might be able to catch all of this one day
- suspend render pass, no longer resumes RP
 - instead just flags RT as being changed
- in short – a must-have tool!
(no time for tons of other examples here, sorry)
- again (and again and again) THANX BALDUR!



Yeah, I know I said in the beginning that I won't talk about barriers

- well, I guess I lied, sorry :)

(this slide is intentionally left in)

Validation layers

- no need to waste words on how essential these are when implementing Vulkan
- but let me mention a serious issue...
- VL bug(s)!
- in the previous example with batched barriers, VL reported a wrong layout...
...at vkQueueSubmit(), too late! 😞
- turned out that VL incorrectly handles batched barriers with the same image but different mipmaps!

Validation layers

- didn't mention this to point out how VLs are bad
 - (because they are not only good, but I repeat, essential!)
- This just means that VLs are not bug-free
- Take into consideration an option that the VL report might be false, and your code is actually fine

<honest times>

my personal success rate of reporting VL bugs to LunarG is about 66.6667%
(yeah, about 1/3 turned out to be bugs on my side!)

</honest times>

Validation layers

- sometimes so overzealous to the point of
- well, being really, really funny!
- querying `vkGetPhysicalDeviceImageFormatProperties()` for something not supported and...

... pop - VL error! :)



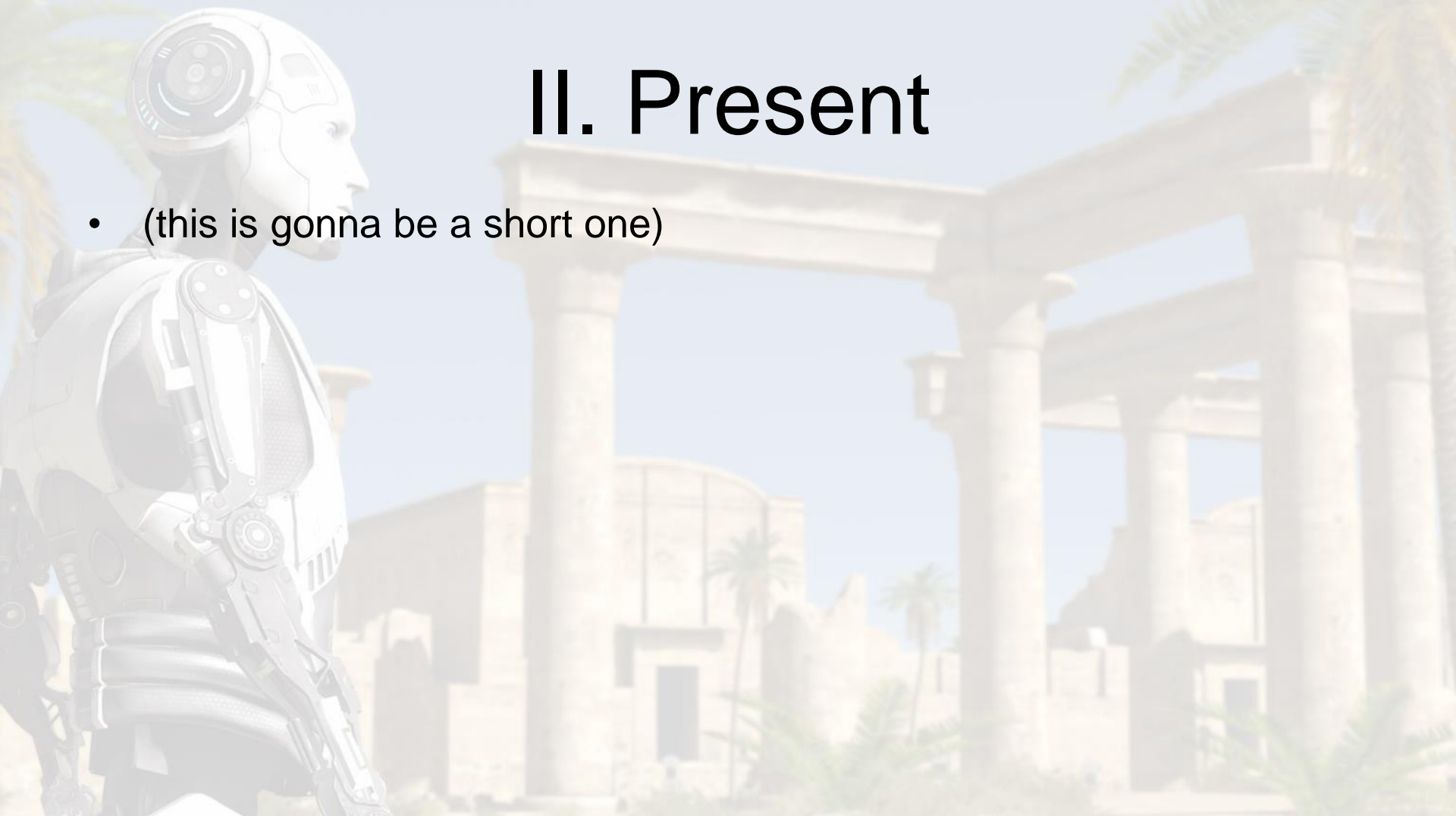
(INTERMISSION)

BUG!

- black grass in the Vulkan/Android version of "Talos" for months!
- our grass shader renders foliage with a bit of polygon (depth) offset
 - (works fine w/o offset)
- using depth offset function "vkCmdSetDepthBias()" = black grass
- setting depth bias inside PSOs = fine, no black grass
- someone somewhere decided to change calling the convention on Android from softfp (OpenGL) to hardfp (Vulkan)
- we're loading the Vulkan module dynamically
 - just copied function declarations from our OpenGL initializer and replaced functions' names :/
 - and the dynamic depth offset function "vkCmdSetDepthBias()" is the only function in Vulkan that we use which has floating-point input parameters
- (black was because scene offset didn't match depth-prepass' offset)

II. Present

- (this is gonna be a short one)



II. Present

- got the first gfx wrapper function in our engine for Vulkan :)
- mark when done with rendering texture as render-target and change layout to shader sampled texture
- one thing in our Vulkan "wrapper" is changing layouts "just in time"
(==late!)
 - when texture is bound as input to shaders, then change the layout
 - not so good
 - either too late for GPU
 - can happen even in the middle of the render-pass!
 - (must "split" RP, very bad)
- this changes layout early, outside of render-pass!

Early layout change

- used for depth prepass, our "fast-lights", post-processing, but...
- ...not always possible
- our refraction, zoom and glitch shaders
 - must be properly sorted (for translucency)
 - copy currently rendered RT into auxiliary RT
 - render back from aux to main RT with some effect applied
- even RP's input attachments will not help you with those!
 - (and this is why we can't have nice things)

Native Vulkan multi-threaded renderer



- still WIP!
- showed us a lot of stuff in our engine that needs to be changed in order to be (more) Vulkan-friendly
- but also showed some really strange decisions when it comes to Vulkan's secondary command buffers
 - (will talk about this later, don't worry, but first...)

A bit of self-criticism

- our late-change philosophy in the engine is now proved wrong
 - needs to know a lot of things up front to make Vulkan happy and efficient
 - not everything can be known up front (mentioned refraction example)
- layout changes in other threads are killing us!
 - BTW, it would be much easier without the need to specify source layout for Vulkan (when changing layouts)
 - it's really hard to track those across threads!

MT renderer, what if...

- we have our own, "generic" MT renderer
 - completely API agnostic
 - really fast round-robin command buffer, because
 - 99% of API functions are inlined
 - simple state changes (boolean this, boolean that)
 - main thread replays everything that was recorded
 - finds (via hashes) API objects (DS, PSOs...)
(might add keying on top of that)
 - binds what's needed
 - calls API functions
- So, we're a bit afraid of...

- Vulkan MT renderer is probably slower at recording than ours
 - at least because of the functions not being inlined, but also
 - because we don't track changes on worker threads, only main thread (we don't have to, because most of our gfx functions are inlined)
 - but secondary command buffer needs to know right away (at recording time) what goes in
 - so sometimes some objects could be set unnecessarily!
- ok, so... just use more threads for recording, you say?
 - How many more?
 - 8?
 - 16?
 - 32?
 - (now we even went over desktop CPUs, not to mention mobile!)
- 4 threads should be enough?
 - ARM CPUs: big.LITTLE, 4 perf, 4 low-power cores
 - 4 OK (better to just ignore lowP cores, or use one of those for background loading/streaming)
 - 4 threads on 4 core mobile (or desktop) CPUs
 - what about OS and/or (a ton!) of background process
 - now we're down to 3, I guess :/

- OK, so 3 cores is something we can count on
 - now, how much slower is the recording of command buffers, again?
 - replaying is much faster
 - it better be (we don't want any surprises there!)
 - unless some vendors want to correct me here...?
- and now...
- ... what if we update our generic MT renderer
- to record complete states in other threads ?!
 - and only thing left for main thread to do is calling Vulkan API functions
 - and there are not much of them left
(bind couple of vertex buffers, bind DS, bind PSO, draw!)
- can this be faster than native MT, in total?!
- (I don't know, I'm just asking...)
 - our command buffer player is still slower, but not by much (or is it?)
 - but recording is definitely much faster!

II. Present (cont'd)



- Node shader!
 - WIP!
- not directly related to Vulkan, but...
 - with each generated shader, we'll create corresponding DS and PSO also!

III. Future

- will go with proper uniform buffers (obviously!)
- after all the mess with packing (stdTHIS, stdTHAT), I'm guessing all our constants will just be float4
- still might need non-constant global initializers, because of our packing of floats into vectors (to reduce number of constant registers used by a given shader)
- hypothetical example:

```
struct _AllConstants {  
    float4 severallnOne;  
    float4 somethingElse;  
} AllConsts;  
static float c_fSpecPower = AllConsts.severallnOne.x;  
static float c_fCoatingFactor = AllConsts.somethingElse.y;  
....
```
- or we just go with #defines for this
 - but then some compilers might not like

```
#define c_vScreenCoords AllConsts.severallnOne.xy
```

and then use

```
float fX = c_vScreenCoords.x;
```

(expands to AllConsts.severallnOne.xy.x!)

Stuff that our Direct3D 11 support already has

- add support for multi GPU setups
- single-pass VR rendering
- mGPU VR rendering
 - will go with vendors' extensions where available
 - or one record and two submits in general case

Precreate everything!

- create DS, PSO, even command buffers on load!
 - but what about render-passes? (complicates things!)
- the only thing to set/update in draw time are uniform buffers
 - or not even those?!
 - toying with the idea to have one big uniform buffer for all constants of all rendering objects
 - and then just select an offset into that buffer via push constants!

More!

- change high(er)-level graphics/rendering engine to match Vulkan "binding" style
 - instead of `gfxBindTexture()`, `gfxEnableBlend()`...
 - just bind this DS, that PSO and some vertex buffers
- but we might keep the old style bindings as before (as a wrapper)
 - to make non-perf-critical rendering in engine simple
 - like editor objects, debug output (and other simple stuff...)
 - if Vulkan-friendly functions ends up (relatively) simple, we're gonna ditch our wrapper functions

And even more!

- decouple samplers from textures
 - there are <16 different samplers used in the whole game and/or editor
 - no point in setting them along textures every time
 - also has a lot of overhead when determining which sampler in which texture unit has been changed!

IV. Rant

The background image is a screenshot from a video game, likely Assassin's Creed Origins, depicting an ancient Egyptian temple complex. The scene is set during the day with a bright, hazy sky. In the foreground, there is a body of water with reeds. The middle ground features several stone structures with hieroglyphs on their walls. A prominent obelisk stands in the background, flanked by columns. The overall atmosphere is warm and historical.

Yes, rant!

- we have to rant, because
- we're Croteam, we always rant about anything and everything! :)
- we're gonna do tons of changes to make our engine (more and more) Vulkan-friendly!
 - that'll take a lot of effort, time and money
 - if we go "by the book", we surely wouldn't want to see that we
 - hit slow driver path
(there should be no slow paths in Vulkan - this is not OpenGL!)
 - driver is unoptimized
(ok, still work in progress, we understand)
 - we did (or will do) our homework, vendors - do yours!
 - "it wasn't meant to be implemented like that,,
(driver does a lot of hidden stuff behind our back)
 - slow command buffer replay, because reasons?!?!
(sorry, not buying this one!)
- if we implement Vulkan "directly" in our engine, we surely don't want drivers to do the wrapping that we just ditched!
- so, this is why...

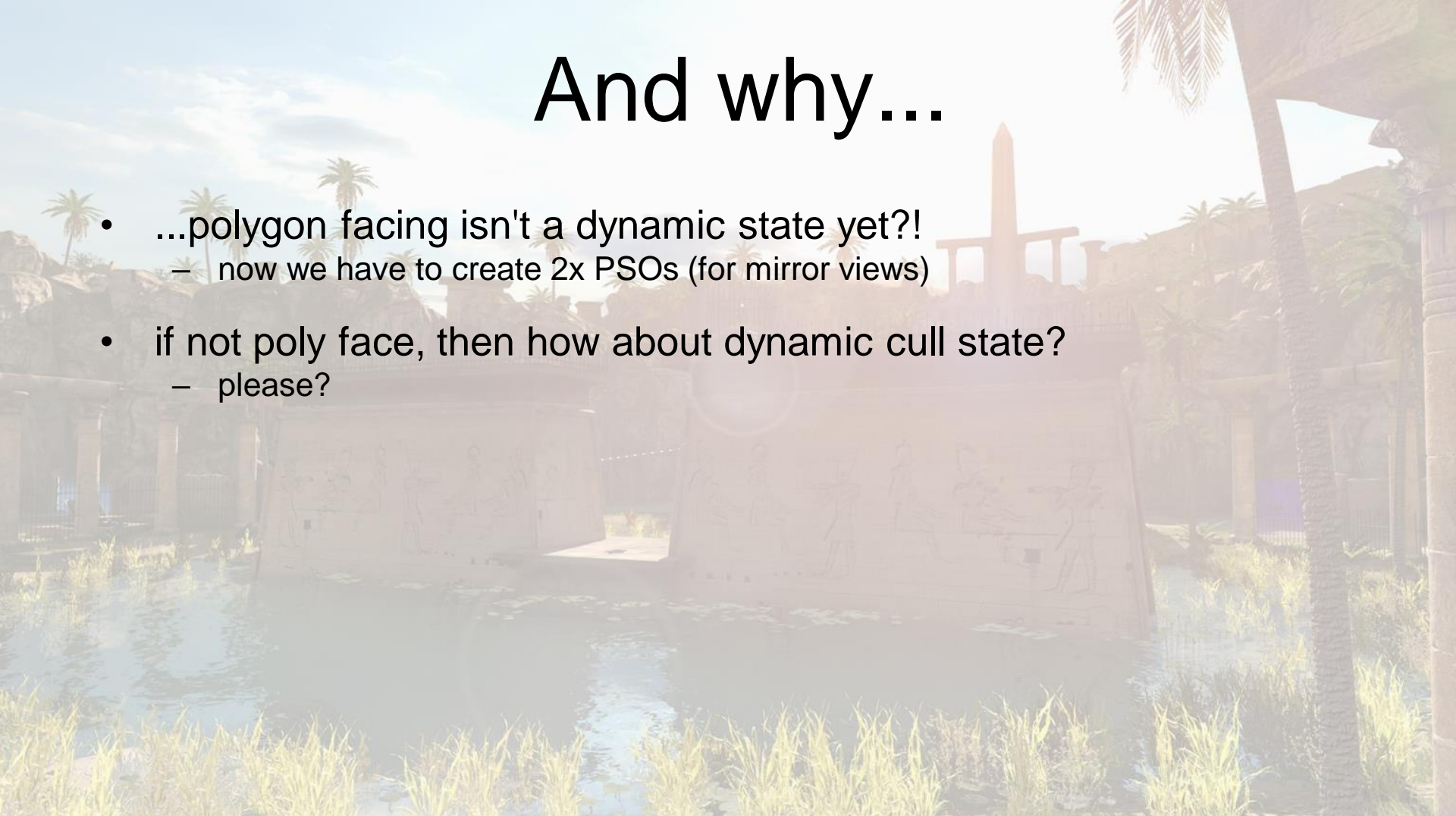
Optimize drivers!

(we've seen Vulkan drivers being slower than OpenGL, yes, believe me!)

- we want super-fast execution of secondary command buffers
 - otherwise, it defeats the purpose!
- as fast as possible recording of secondary command buffers
 - not just throwing more cores on the problem!
- fast(er) PSO creation
 - fully utilize PSO cache!
 - instead of not having it at all
 - or, even worse, faking it! (won't mention any names here)
 - don't compile shaders inside PSO if you have `vkCreateShaderModule()` for it
 - instead just patch shaders to fit given PSO
 - will that shader end up being slower than fully recompiled one?
(personally, I doubt it, but I could be wrong here)
 - if you really have to recompile a shader, at least try to put PSO cache into good use there

And why...

- ...polygon facing isn't a dynamic state yet?!
 - now we have to create 2x PSOs (for mirror views)
- if not poly face, then how about dynamic cull state?
 - please?



- expose hidden device memory allocations
 - not talking about allocation callbacks (these aren't of any use, anyhow)
 - device (shared) memory used by other processes and/or driver internals
 - we should at least have some feedback from Vulkan about those (like D3D12 has)
- hint about preferred memory allocations
 - should they be direct (left to Vulkan) or indirect (via our memory manager)
 - what underlying driver/HW can handle fast(er)?
 - for now, we just use adjustable threshold (default: >1MB alloc goes to Vulkan directly)
- hint about preferred image layouts
 - example: how to create mipmaps for render-target?
 - barriers with transfer_src and transfer_dst layout changes for every mip, or
 - barriers without layout changes (keep all mips in general layout while generating)

(yeah, I know, I'm talking about barriers... again. sorry!)

Occlusion queries

- still wrestling with those ☹️
- really hard to fit those into our visibility system
 - not all queries in a batch are in use
(there can be some "holes" inside the query pool)
 - reset but not begun/ended query implies wait forever on get result!
 - can't reset queries inside of the render pass

Native multi-threaded renderer

- requires secondary command buffers to specify the render-pass in which they'll be executed
 - at recording time (?!)
 - is this some kind of a hint for driver?
 - can this just be the image/frame-buffer's format compatibility, instead?
- you must specify (in the render-pass?!) that you either execute secondary command buffers or use any other Vulkan commands
 - if I may ask... why?!

Shader compilers (from SPIR-V)

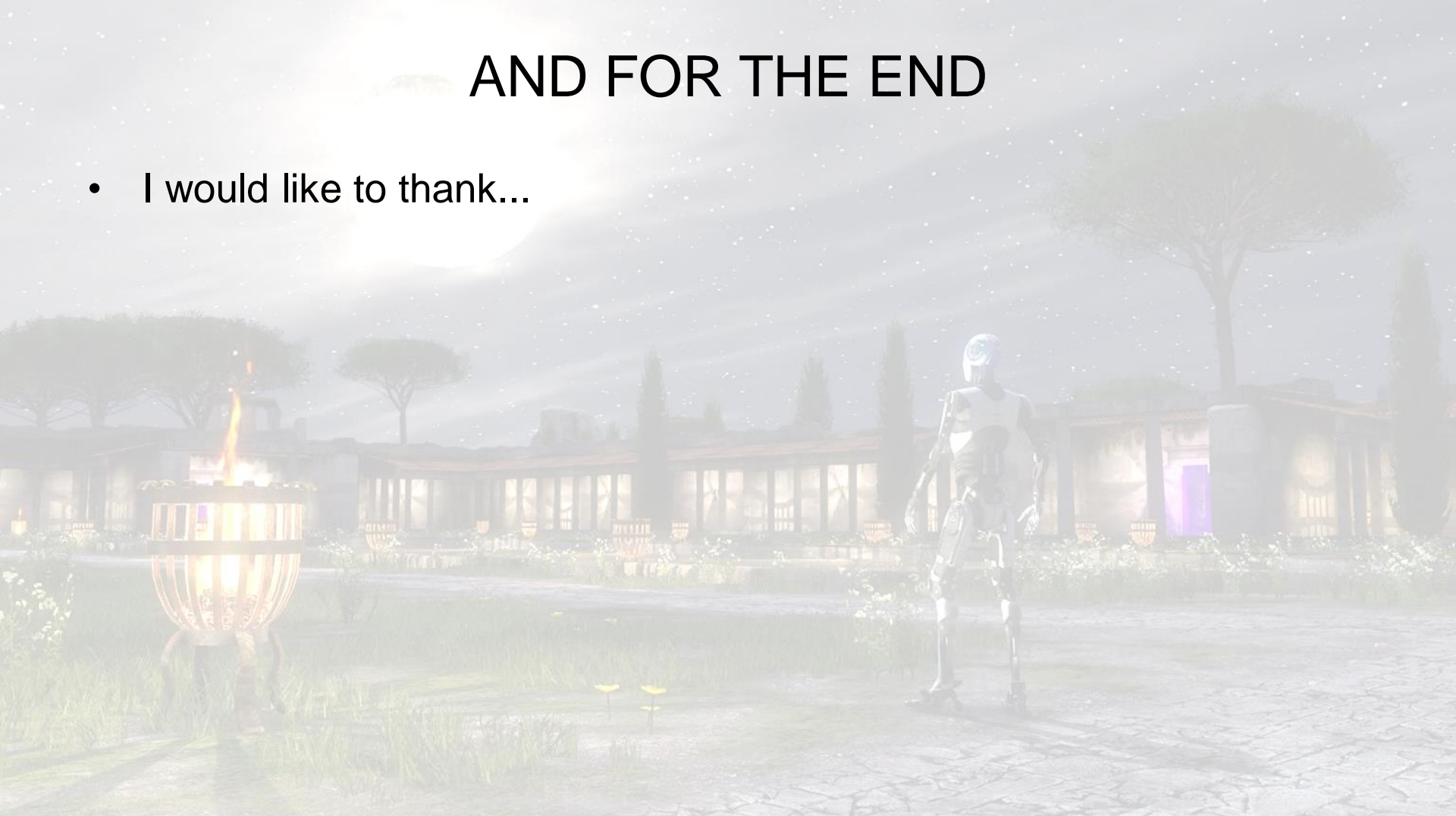
- still crash a lot on Android platform! :(
- or produce bad/wrong code
(it's been more than a year since Vulkan release, come on!)
- SPIR-V code size much larger than D3D's byte-code
 - even stripped and SMOLV-ed and zipped!
 - does this reflect to being slower than D3D?
 - we suspect MS has better optimizations, like
uniform float4 var;
static float subvar = var.x;
 - last line is compiled as (kind of) "#define" in HLSL,
but as an assignment in the main function in SPIR-V :(

And a special rant for Android platform

- what's with `SetSustainedPerformance()` there?
 - is it supported or not?
 - if not, why?
 - this is killing us on some phones!
 - thermal throttling can easily take away 50% of performance!
 - but we don't care, we have super-scalable engine
 - and we think Talos is the game that can actually be played on phones (not only tablets)
 - just need to know up front on what performance can we count on
- driver updates!
 - will not repeat myself here, but still...
 - GPU vendor -> google -> device vendor -> ISP -> user
 - (what are the chances that a user will get the driver update?)
 - Google project "Treble" should help, hopefully
 - but that's only for future OS versions! ☹️

AND FOR THE END

- I would like to thank...





(INTERMISSION)

BUG, again!

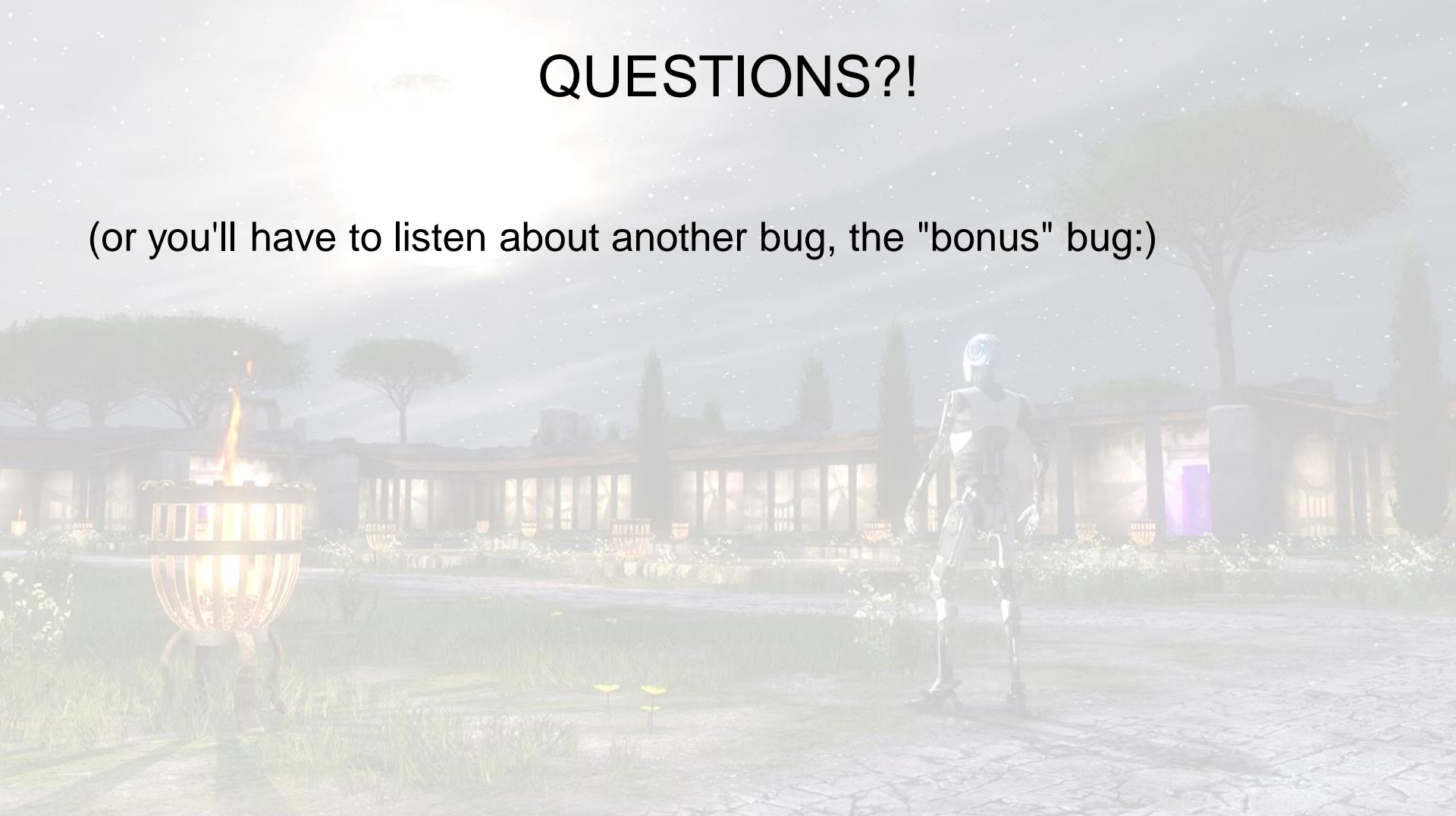
- destroying a render-pass (image, frame buffer) or a texture without delay
- using `vkWaitDeviceIdle()`
 - (some special cases, when perf is not an issue, HDR RT resolution change, level start, reloading of textures...)
- created a new image right away
 - and got the same Vulkan handle!
- but forgot to destroy (right away!) DS and PSOs referencing those objects
(these were scheduled for delayed destroy!)
- next DS/PSO bind for rendering would find those DS/PSO which will reference "wrong", old RP/TEX objects (same handle, different object!)
- sometimes GPU/driver crash, sometimes wrong rendering

Thank you

- special thanks goes to all the great folks at nVidia, AMD, Valve, LunarG, Intel, Samsung, ARM, QCOM... and Baldur for RenderDoc!
- and Alen (our CTO) who started all this by being really (pro)active on Vulkan Advisory Panel, while I was busy on other fronts
- also all friends and colleagues at Croteam who helped me with this port and gave me courage with their kind words
("You're never gonna finish this",
"Vulkan 'till retirement",
"Vulkan programmer's work is never done",
"Drop it while you're young... oh sorry, you're not young anymore"...)

QUESTIONS?!

(or you'll have to listen about another bug, the "bonus" bug:)



OK, bug then!

- in this case, it might be a nice option to ignore particular VL report of particular object (not just object type!)
- missed semaphores in vkQueueSubmit()
 - (I know, I know, I'm ashamed)
- but VL layers didn't report
 - could they?
 - we have another vkQueueSubmit() (before the "main" one) that doesn't need semaphores because it only updates content of constants buffer
- interesting one because this actually worked on all platforms and drivers apart from one
 - like this:
 - vkAcquireNextImage(signal semaphore)
 - vkQueueSubmit(no semaphores!)
 - vkQueuePresent(wait semaphore from acquire)