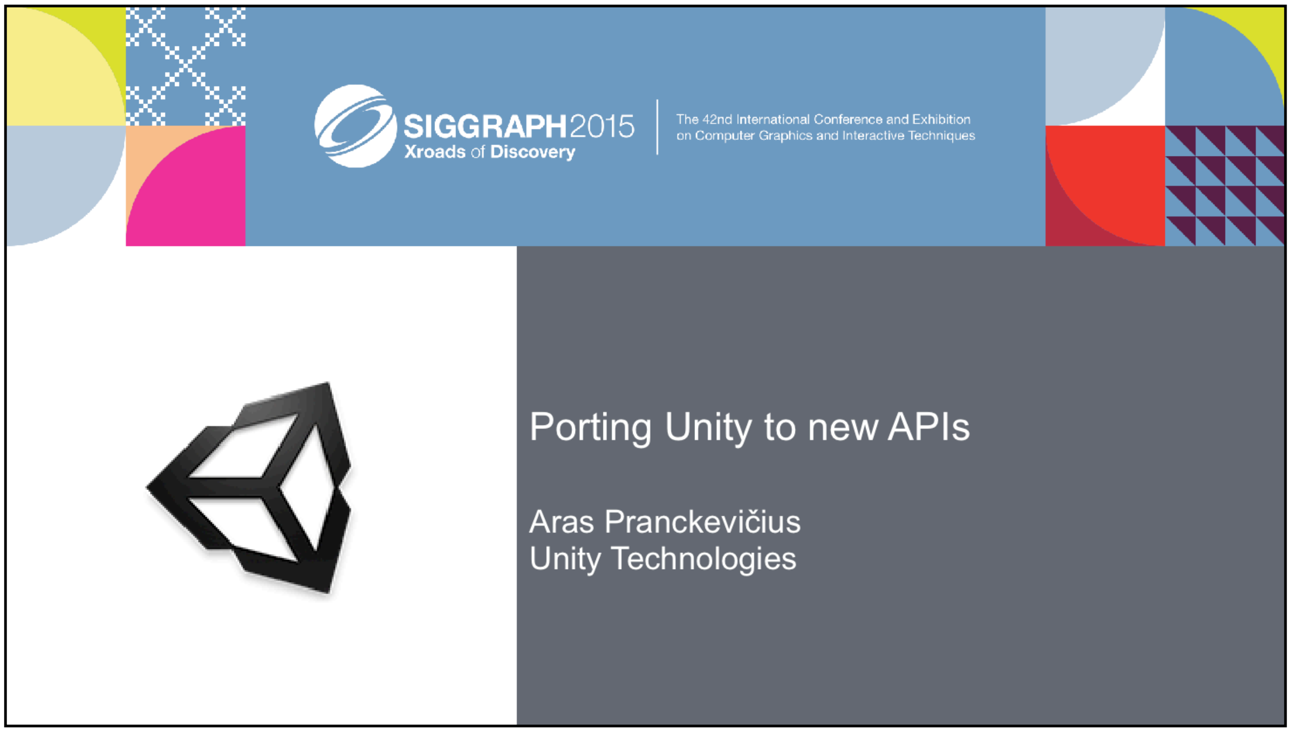
SIGGRAPH2015
Xroads of Discovery

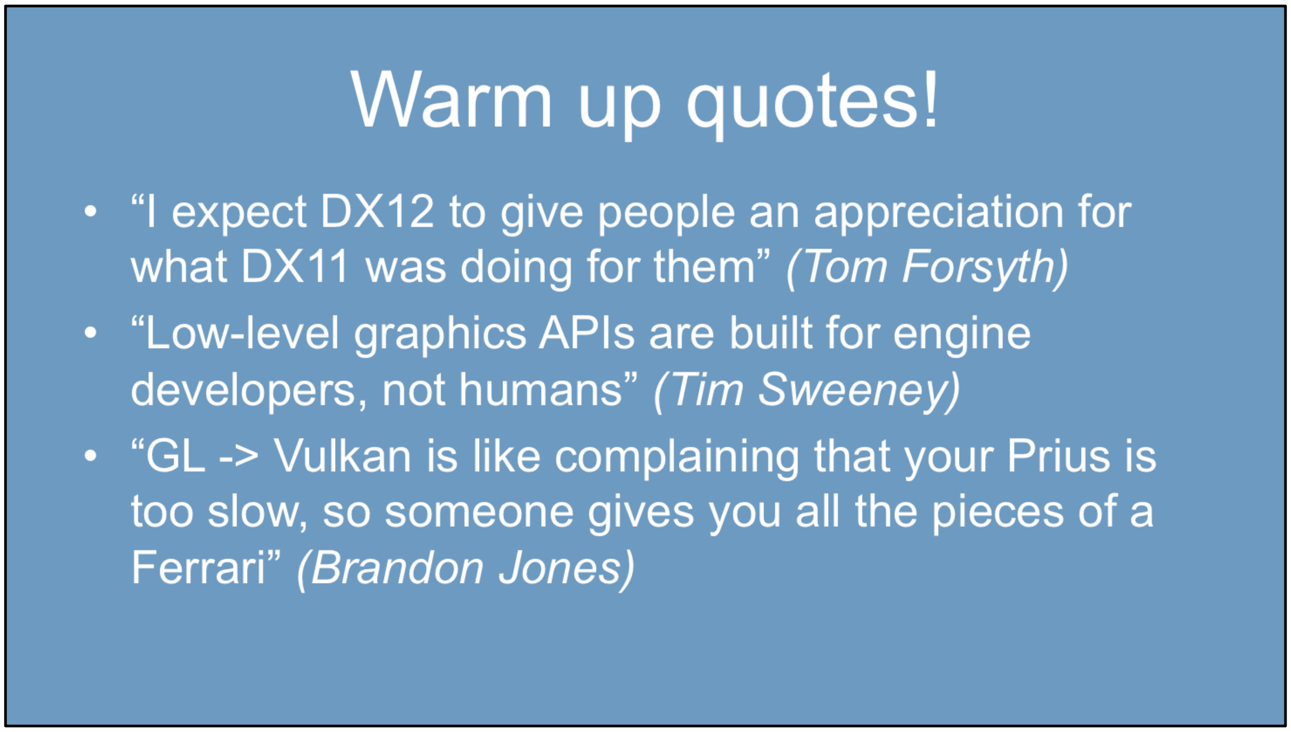


SIGGRAPH2015
Xroads of Discovery

The 42nd International Conference and Exhibition on Computer Graphics and Interactive Techniques

Porting Unity to new APIs

Aras Pranckevičius
Unity Technologies



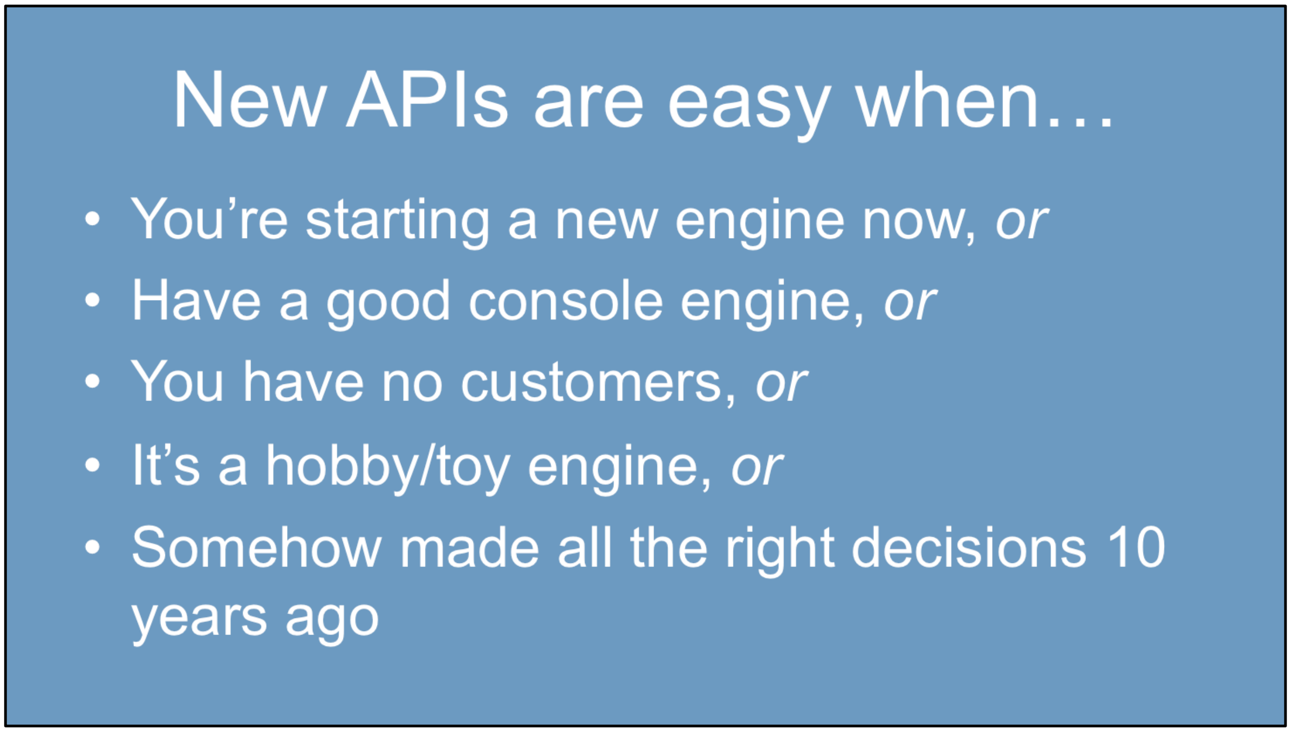
Warm up quotes!

- “I expect DX12 to give people an appreciation for what DX11 was doing for them” *(Tom Forsyth)*
- “Low-level graphics APIs are built for engine developers, not humans” *(Tim Sweeney)*
- “GL -> Vulkan is like complaining that your Prius is too slow, so someone gives you all the pieces of a Ferrari” *(Brandon Jones)*

https://twitter.com/TimSweeneyEpic/status/628713913094422528

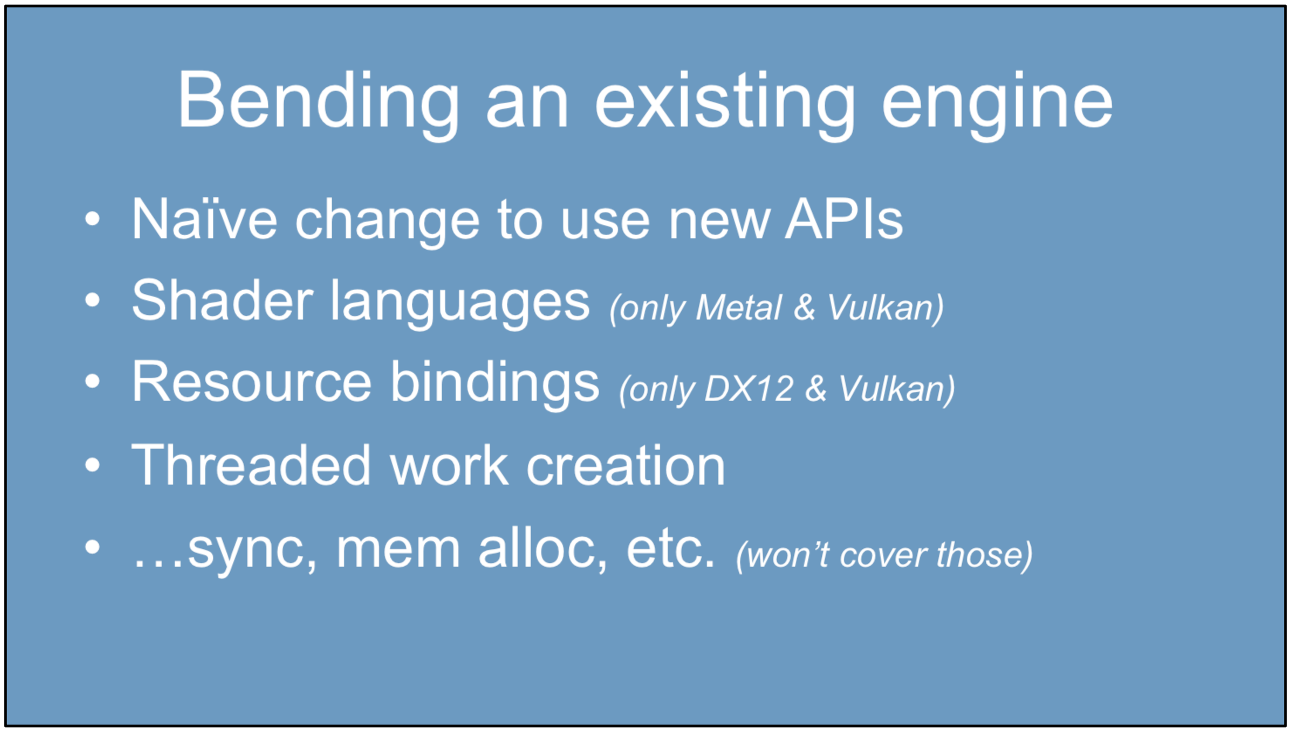
https://twitter.com/tom_forsyth/status/628751206492569601

https://twitter.com/Tojiro/status/628664374408839169



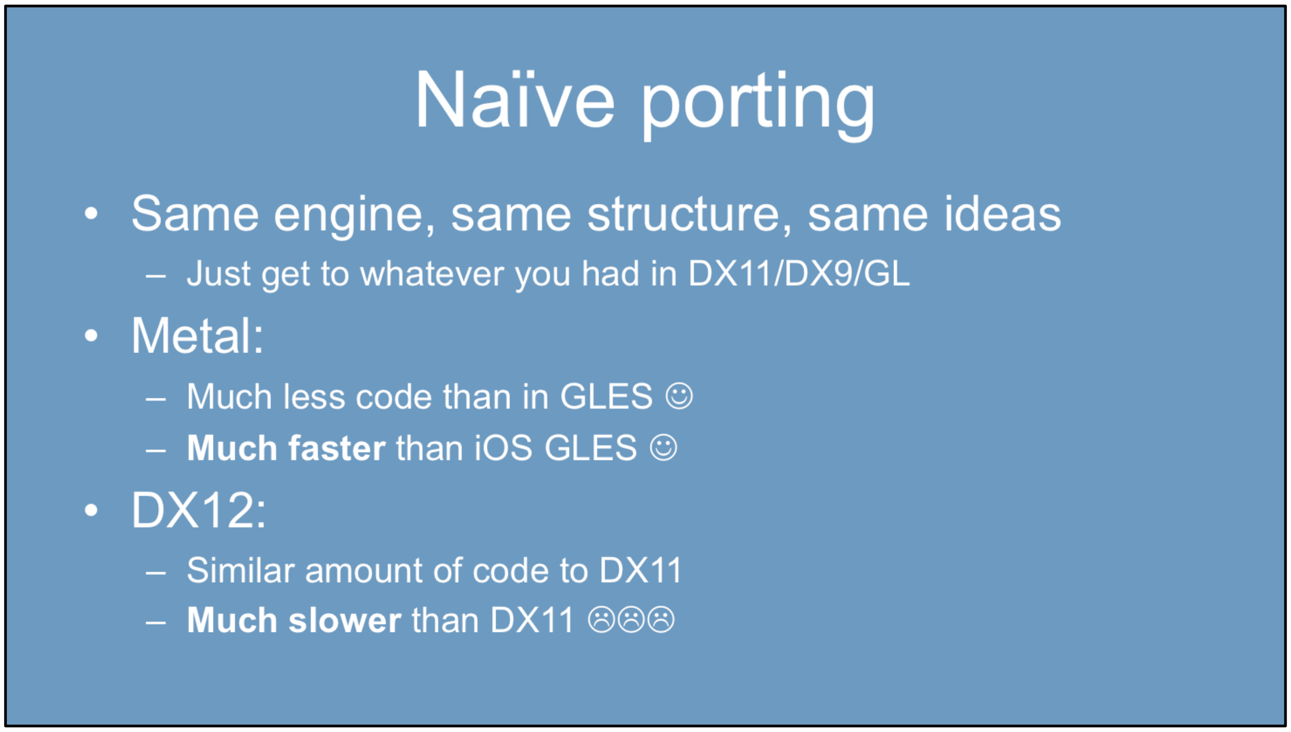
New APIs are easy when…

- You're starting a new engine now, *or*
- Have a good console engine, *or*
- You have no customers, *or*
- It's a hobby/toy engine, *or*
- Somehow made all the right decisions 10 years ago



Bending an existing engine

- Naïve change to use new APIs
- Shader languages *(only Metal & Vulkan)*
- Resource bindings *(only DX12 & Vulkan)*
- Threaded work creation
- …sync, mem alloc, etc. *(won’t cover those)*



Naïve porting

- Same engine, same structure, same ideas
  - Just get to whatever you had in DX11/DX9/GL
- Metal:
  - Much less code than in GLES ☺
  - **Much faster** than iOS GLES ☺
- DX12:
  - Similar amount of code to DX11
  - **Much slower** than DX11 ☹☹☹

GfxDevice (our "graphics platform abstraction") implementation sizes in Unity right now (~5.3), so not reflecting the "initial port" state.

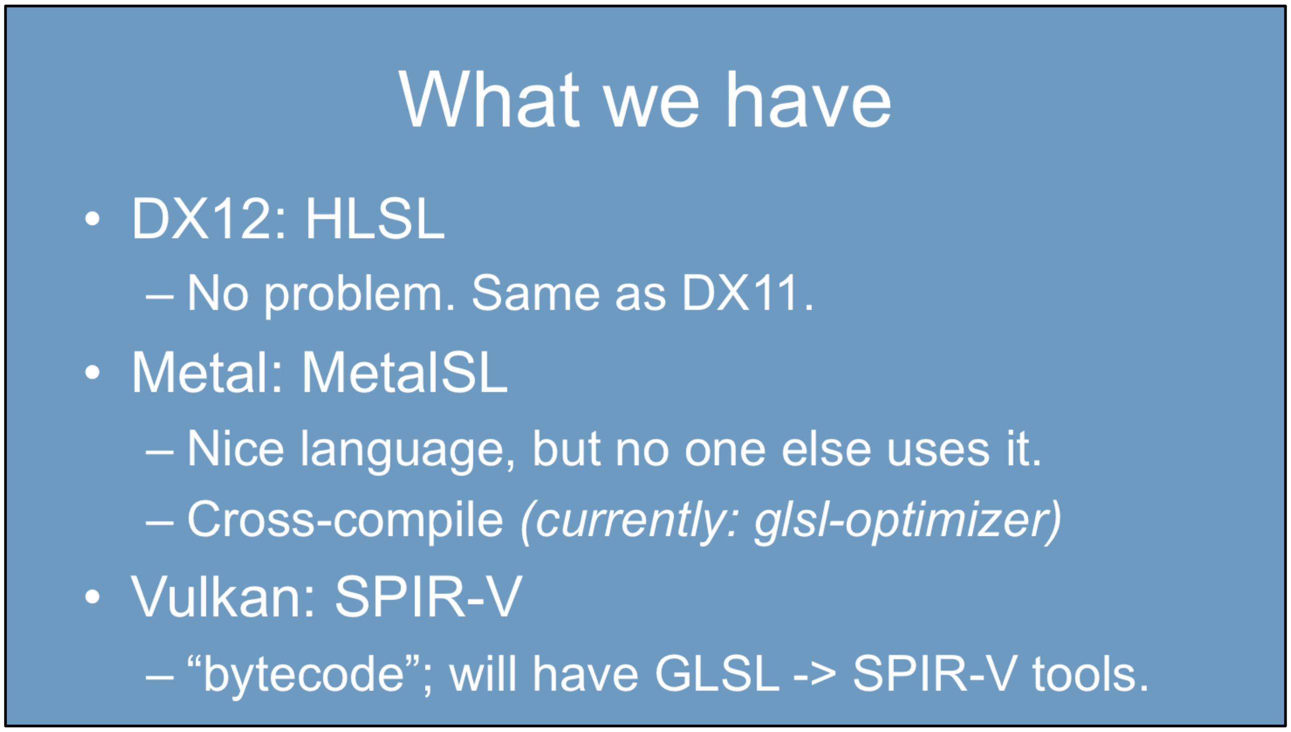
OpenGL + ES **727**kb *(does all GL/ES versions, and this includes all the functions/ extensions loading code)*

D3D11 **474**kb *(quite some related to WSA/WP8, and DX11 bytecode massaging)*

D3D12 **364**kb
D3D9 **231**kb
Legacy GL **231**kb *(dekstop GL 2.0-2.1, will go away soon)*

Metal **132**kb

SHADING LANGUAGES



What we have

- DX12: HLSL
  - No problem. Same as DX11.
- Metal: MetalSL
  - Nice language, but no one else uses it.
  - Cross-compile *(currently: glsl-optimizer)*
- Vulkan: SPIR-V
  - “bytecode”; will have GLSL -> SPIR-V tools.

Our current messy pipeline:

• HLSL is the source language

  • We also do some code analysis / generation (Unity surface shaders); that is a combination of Cg 2.2 and MojoShader.

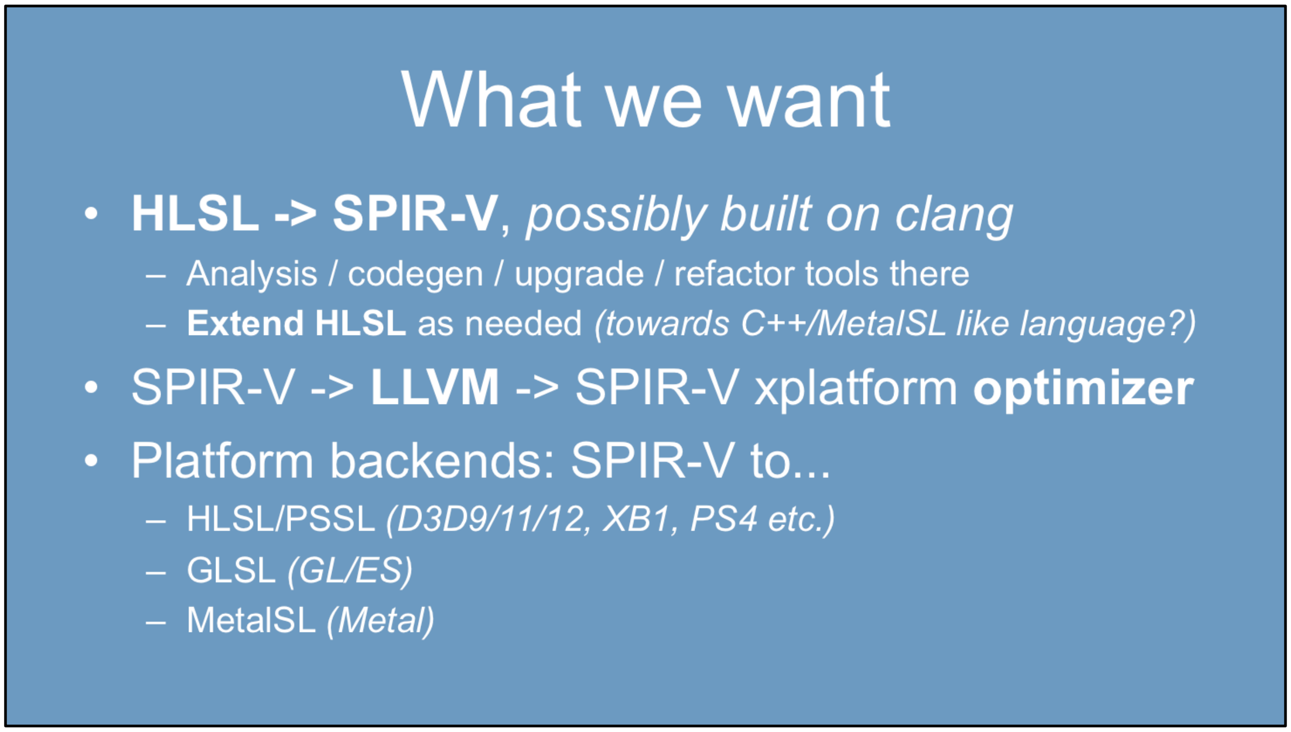
• D3D9/11/12: use d3dcompiler_xx.dll

• GL2, GLES2: HLSL -> (hlsl2glslfork) -> GLSL -> (glsl-optimizer) -> GLSL

• GL3/4, GLES3: HLSL -> (hlslcc) -> GLSL

• Metal: HLSL -> (hlsl2glslfork) -> GLSL -> (glsl-optimizer) -> MetalSL

• Vulkan: ???



What we want

- **HLSL -> SPIR-V**, *possibly built on clang*
  - Analysis / codegen / upgrade / refactor tools there
  - **Extend HLSL** as needed *(towards C++/MetalSL like language?)*
- SPIR-V -> **LLVM** -> SPIR-V xplatform **optimizer**
- Platform backends: SPIR-V to...
  - HLSL/PSSL *(D3D9/11/12, XB1, PS4 etc.)*
  - GLSL *(GL/ES)*
  - MetalSL *(Metal)*

HLSL is not ideal shading / compute language, but there's massive, *massive* amount of shader code written in it already.

We'd want that to continue working :)

Eventually we'd want to move towards something like MetalSL, where it's basically a C/C++ subset *(no virtuals, no exceptions etc.)*, but extended for GPU specific things with various annotations.

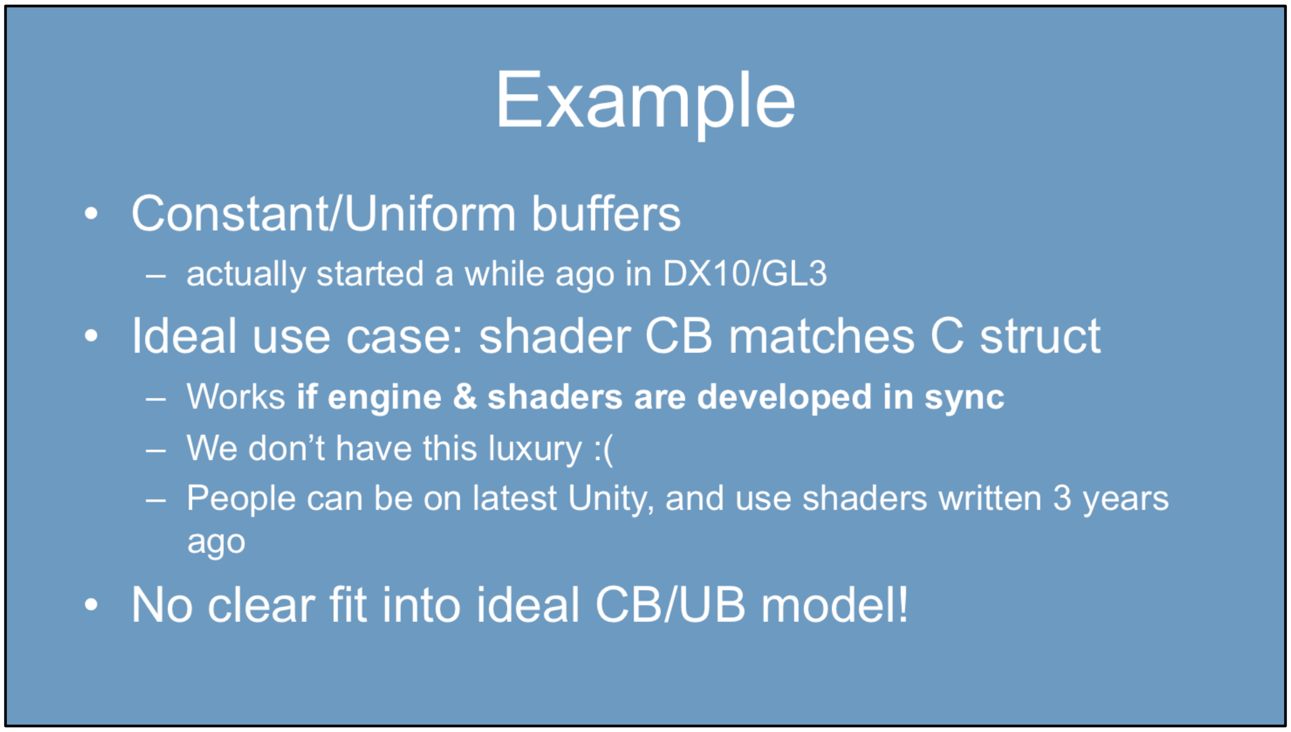
RESOURCE BINDINGS

Why so slow?

- Initial DX12/Vulkan port will not run fast
- Lots of “late” decisions
  - e.g. at Draw time, “so what has changed? which tables need update?” etc.
  - That is the “driver cost”! But drivers were heavily optimized already. Your new code is not.

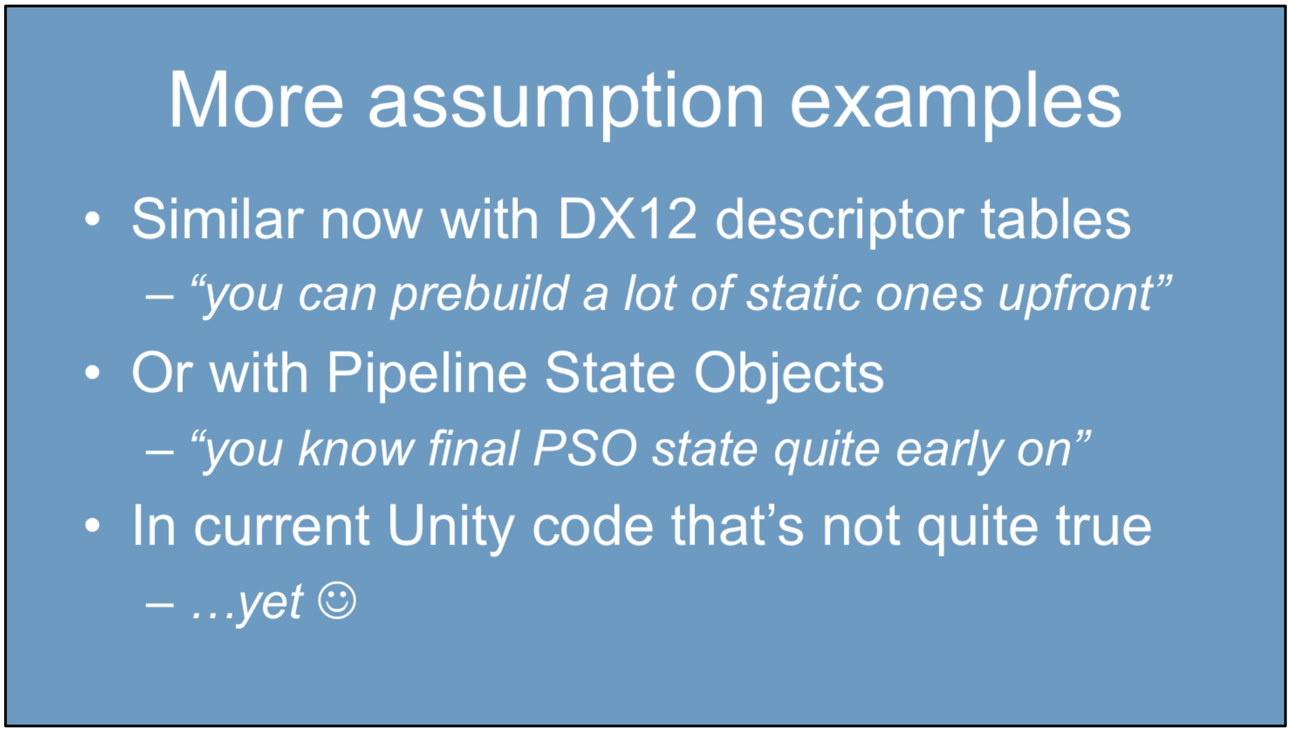
APIs assume that…

- You have *a lot* of knowledge
  - Which things change and when
  - Which things don’t change
  - Layouts of shader resources / constants
- Ideally you have that knowledge!
  - Your existing code might not *(yet)*



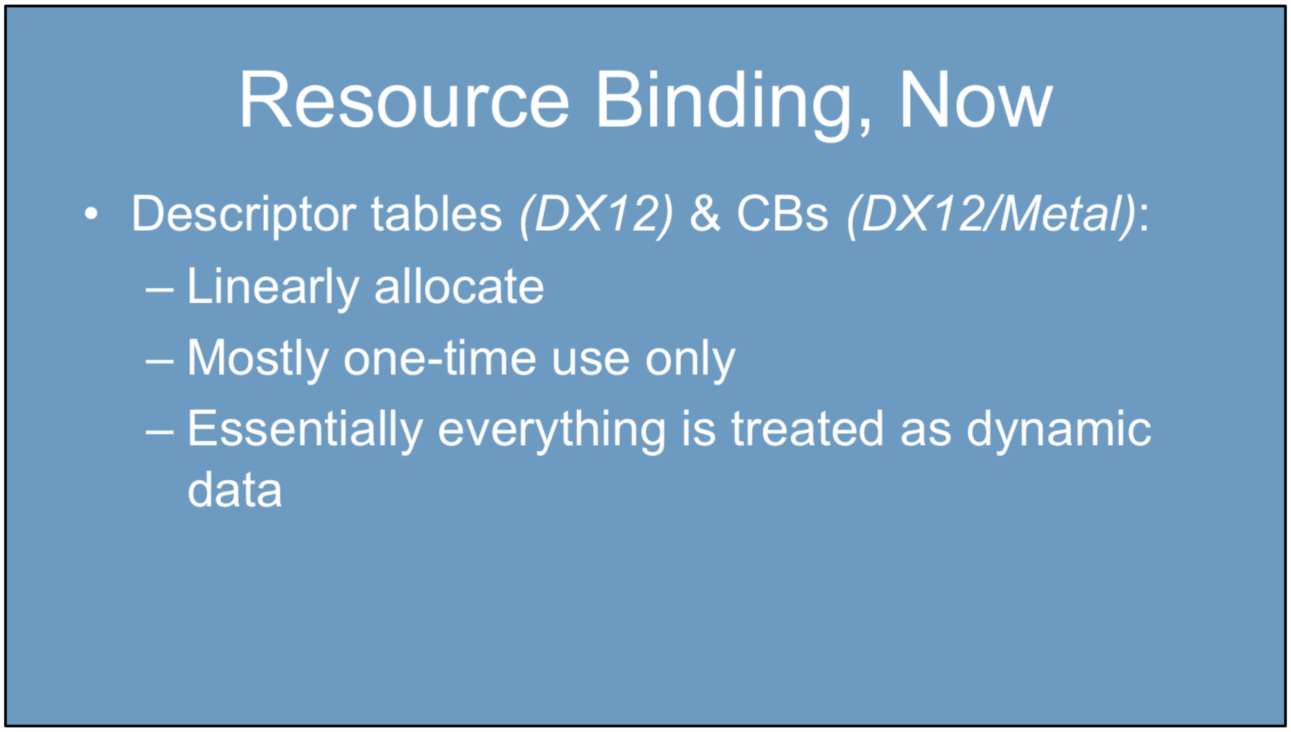
Example

- Constant/Uniform buffers
  - actually started a while ago in DX10/GL3
- Ideal use case: shader CB matches C struct
  - Works **if engine & shaders are developed in sync**
  - We don't have this luxury :(
  - People can be on latest Unity, and use shaders written 3 years ago
- No clear fit into ideal CB/UB model!



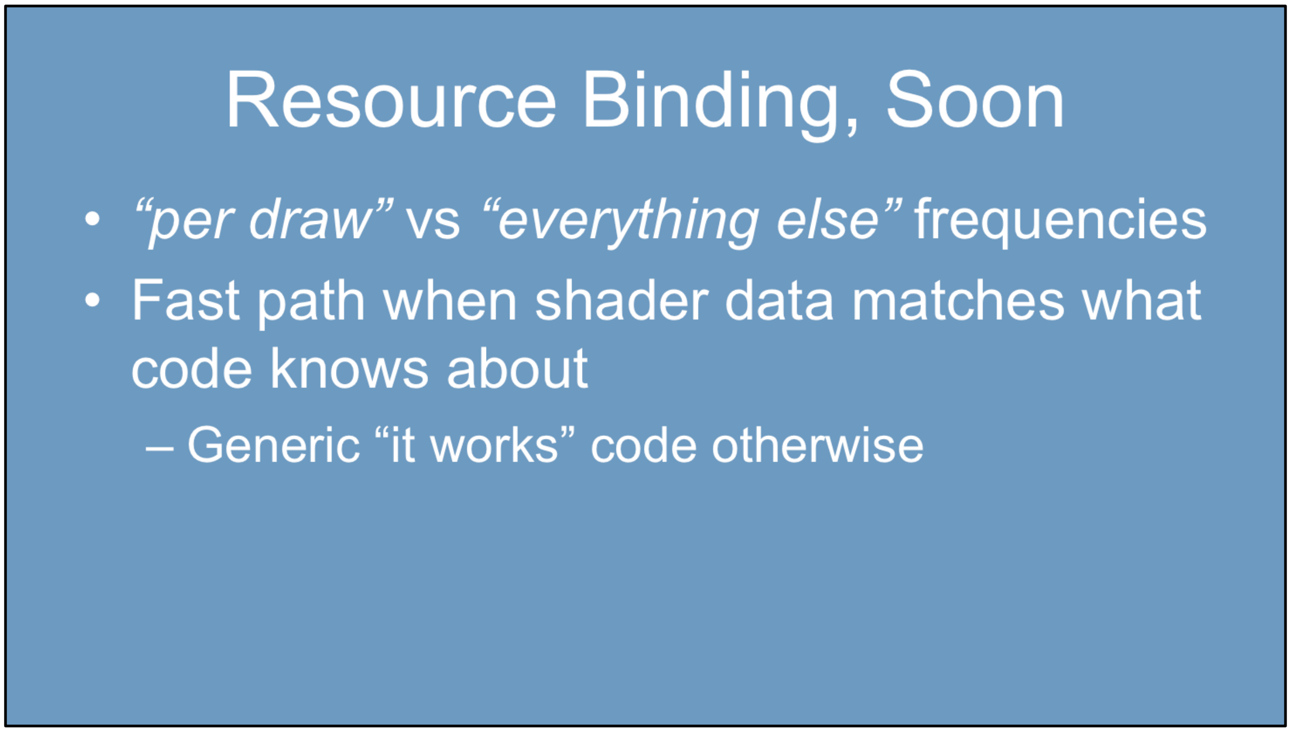
More assumption examples

- Similar now with DX12 descriptor tables
  - *"you can prebuild a lot of static ones upfront"*
- Or with Pipeline State Objects
  - *"you know final PSO state quite early on"*
- In current Unity code that's not quite true
  - *…yet* ☺



Resource Binding, Now

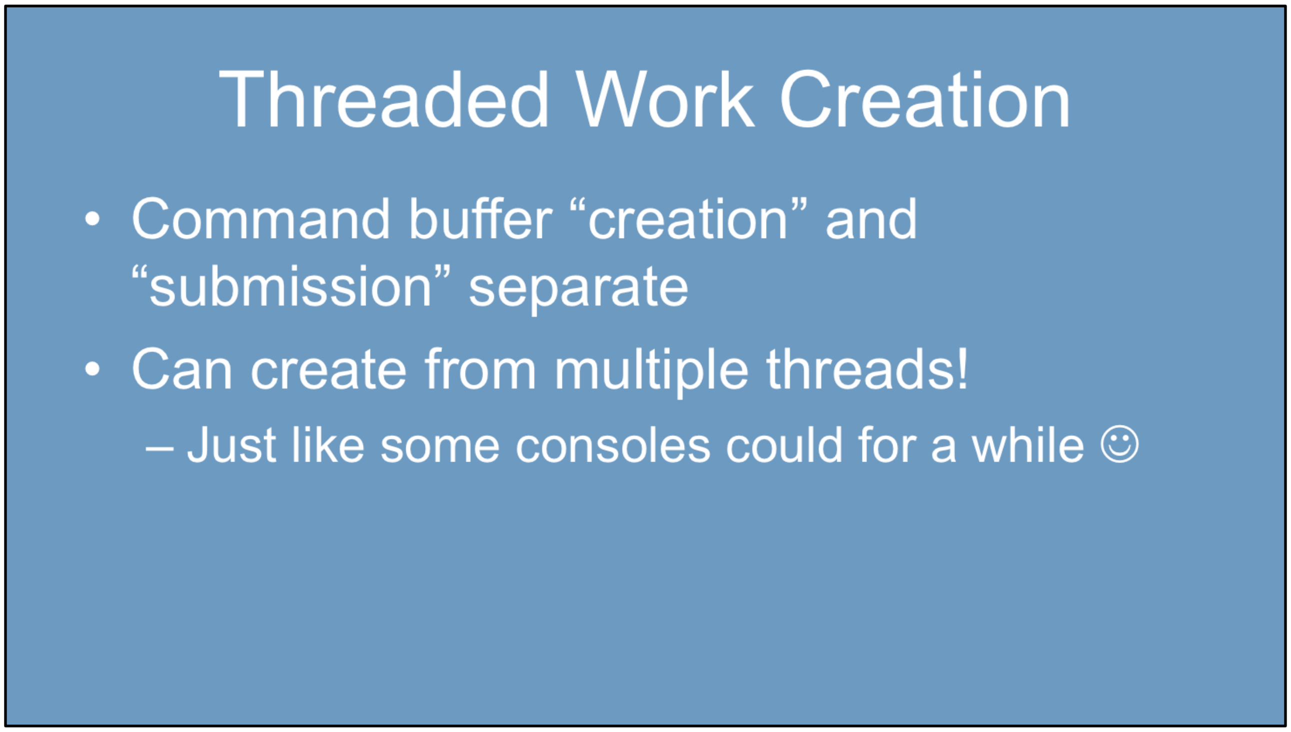
- Descriptor tables *(DX12)* & CBs *(DX12/Metal)*:
  - Linearly allocate
  - Mostly one-time use only
  - Essentially everything is treated as dynamic data



Resource Binding, Soon

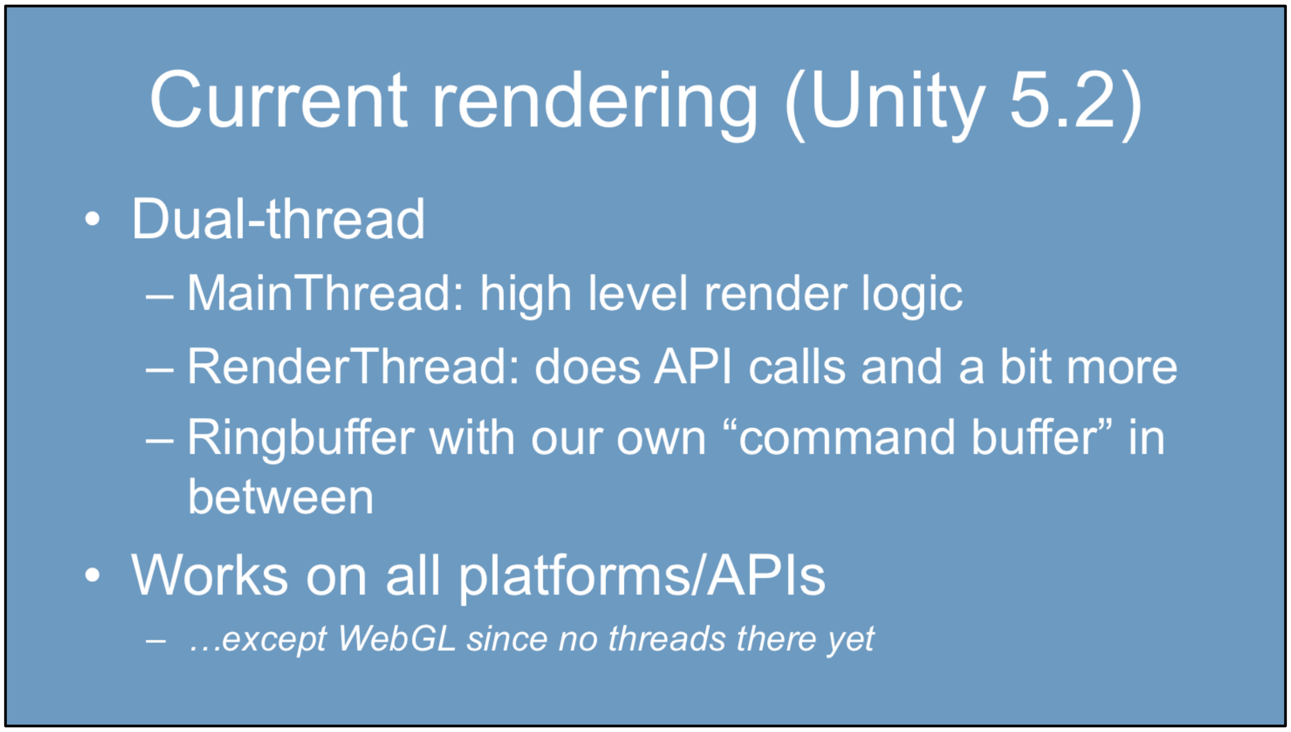
- *“per draw”* vs *“everything else”* frequencies
- Fast path when shader data matches what code knows about
  - Generic “it works” code otherwise

THREADED WORK CREATION



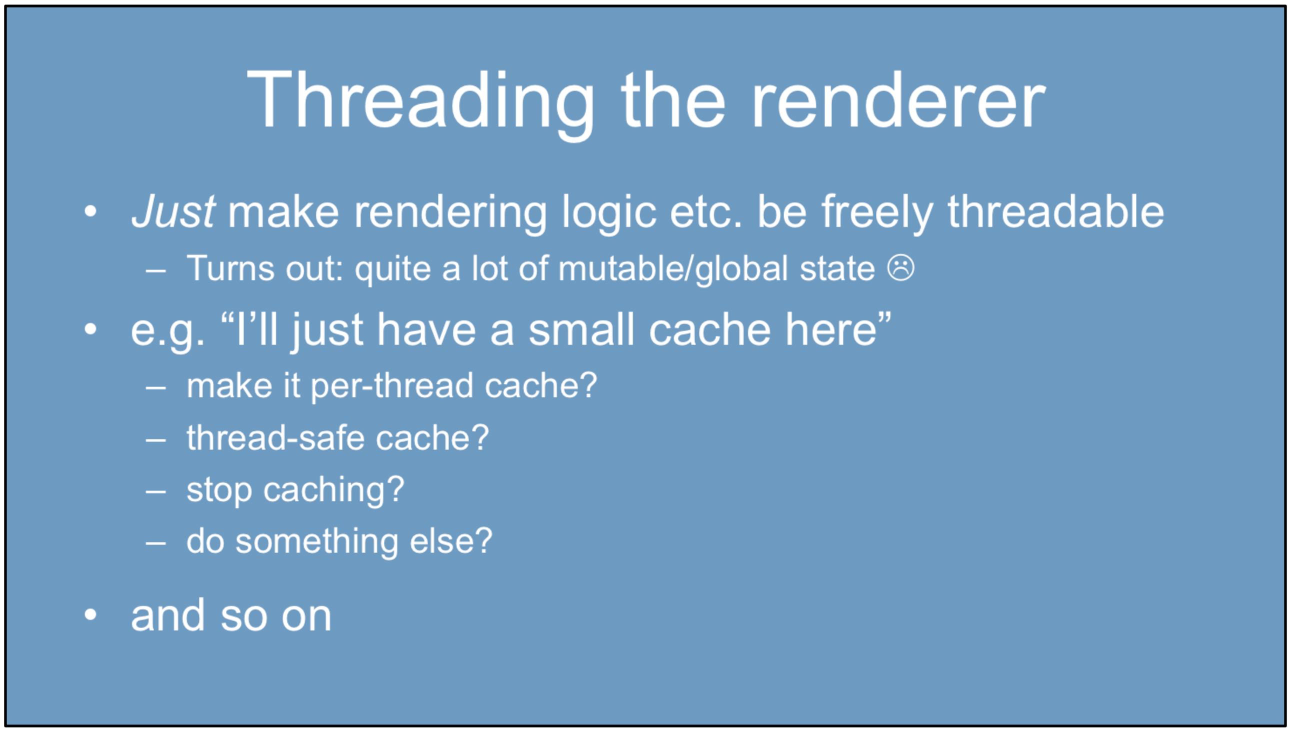
Threaded Work Creation

- Command buffer “creation” and “submission” separate
- Can create from multiple threads!
  - Just like some consoles could for a while ☺



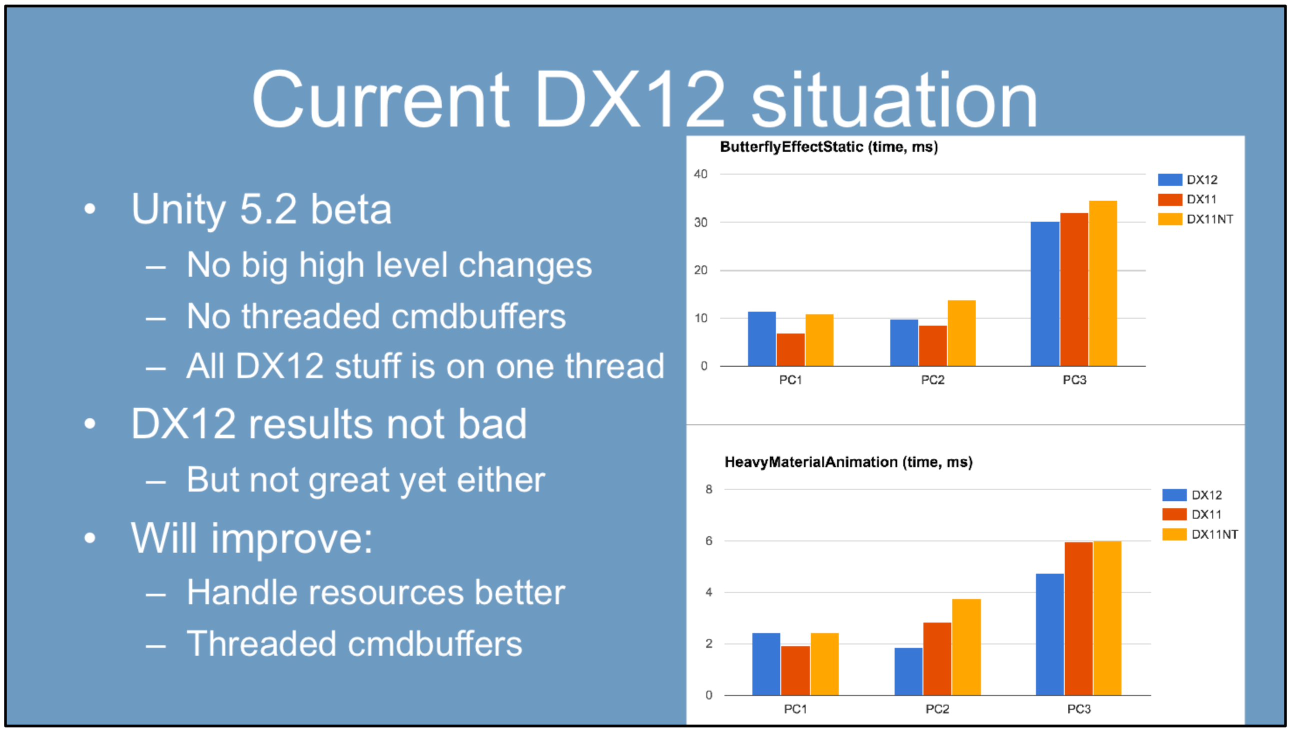
Current rendering (Unity 5.2)

- Dual-thread
  - MainThread: high level render logic
  - RenderThread: does API calls and a bit more
  - Ringbuffer with our own “command buffer” in between
- Works on all platforms/APIs
  - *…except WebGL since no threads there yet*



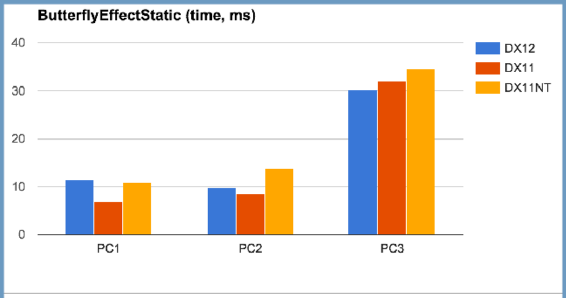
Threading the renderer

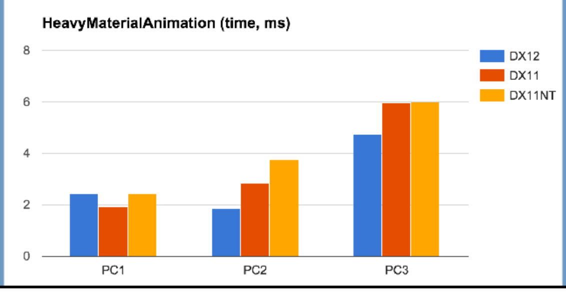
- *Just* make rendering logic etc. be freely threadable
  - Turns out: quite a lot of mutable/global state ☹
- e.g. “I’ll just have a small cache here”
  - make it per-thread cache?
  - thread-safe cache?
  - stop caching?
  - do something else?
- and so on



Current DX12 situation

- Unity 5.2 beta
  - No big high level changes
  - No threaded cmdbuffers
  - All DX12 stuff is on one thread
- DX12 results not bad
  - But not great yet either
- Will improve:
  - Handle resources better
  - Threaded cmdbuffers



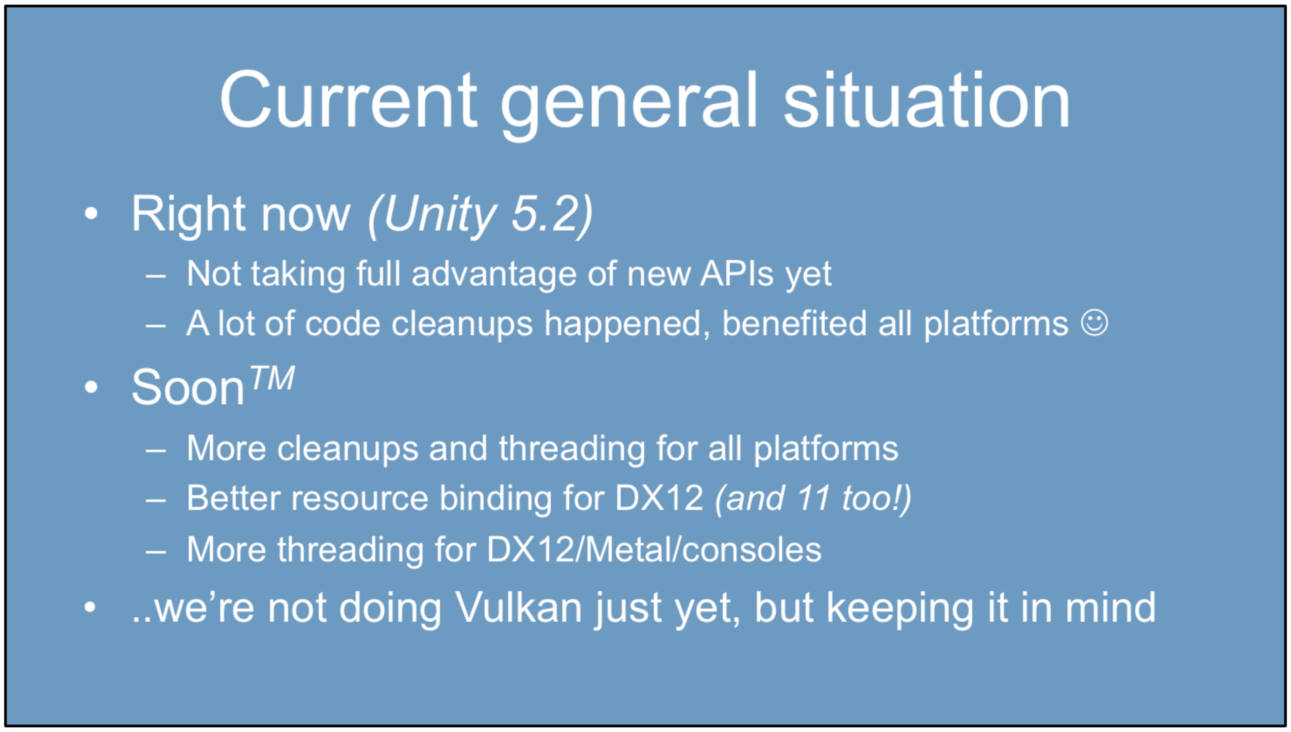


Current situation (Unity 5.2 beta) is that we did not do big higher level changes to the engine code yet. i.e. the graphics abstraction is still somewhere between "DX9 and DX11 level"; with quite many decisions done late. No efficient representation of constant buffers, no efficient representation of shader binding tables that never change, etc.

Also, all the command buffer creation is still done from one thread (the "main thread vs render thread" split, but not "all cores can create command buffers").

This is somewhat different from DX11, where even to the app it looks like everything is on one thread, most (all?) drivers internally will create additional threads to offload some of the "driver work" that they have to do. It's possible to prevent this with D3D11_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS device creation flag, which is what the "DX11NT" ("no driver threads") column represents.

The numbers are from two different benchmark scenes we have (one mostly static; another where a lot of materials animate their parameters). Tests were done on 3 PCs with different hardware configurations.



Current general situation

- Right now *(Unity 5.2)*
  - Not taking full advantage of new APIs yet
  - A lot of code cleanups happened, benefited all platforms ☺
- Soon*TM*
  - More cleanups and threading for all platforms
  - Better resource binding for DX12 *(and 11 too!)*
  - More threading for DX12/Metal/consoles
- ..we're not doing Vulkan just yet, but keeping it in mind

Question you might have: is all this trouble worth it? So far we're not taking advantage of the new APIs to full extent yet, however just by trying to make it better we've already been able to improve quite a lot of things. For example, part of cleanups & general optimizations realdy done in Unity 5.2:

http://aras-p.info/blog/2015/04/01/optimizing-unity-renderer-1-intro/

http://aras-p.info/blog/2015/04/04/optimizing-unity-renderer-2-cleanups/

http://aras-p.info/blog/2015/04/27/optimizing-unity-renderer-3-fixed-function-removal/

…and a bit more that I haven't blogged about (yet?)