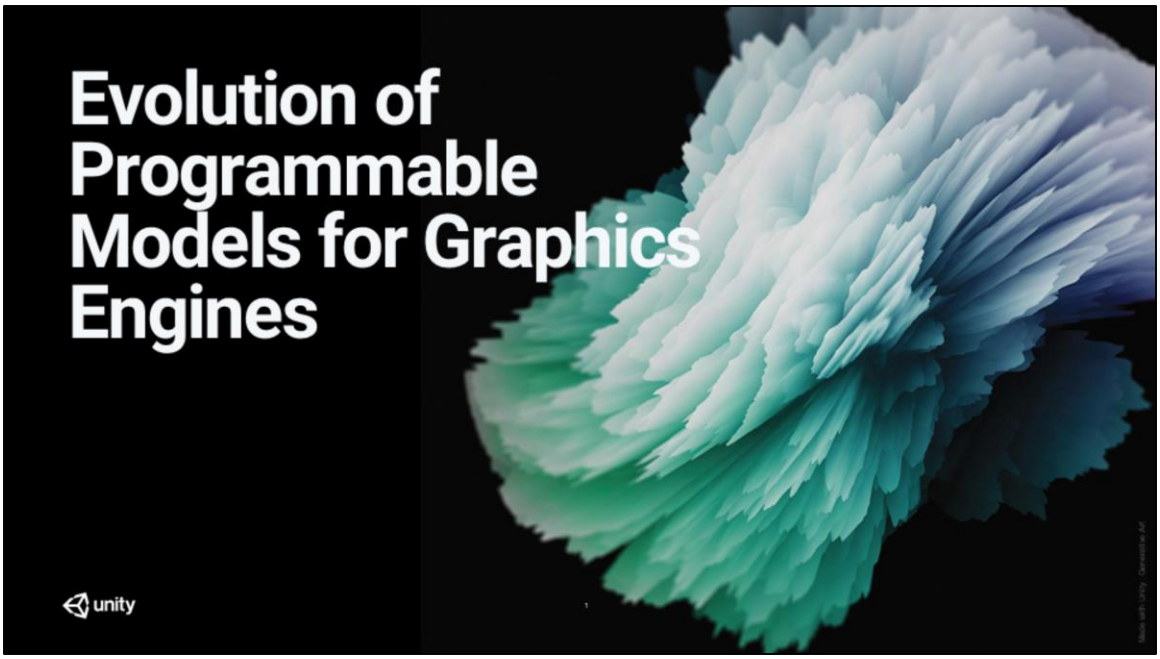


Hello and welcome! Today I want to talk about the evolution of programmable models for graphics engine programming for algorithm developing

My name is Natalya Tatarchuk (some folks know me as Natasha) and I am director of global graphics at Unity

I recently joined Unity…

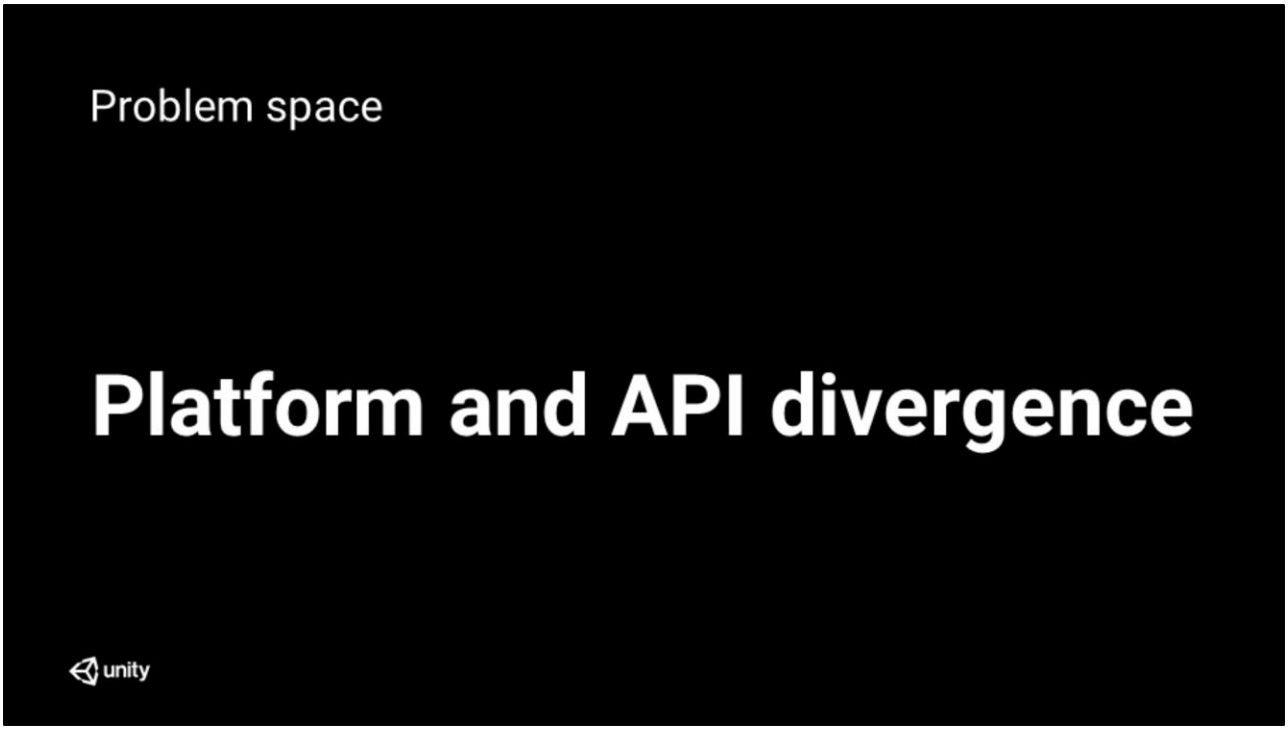
…right after having helped ship the original Destiny @ Bungie where I was the graphics lead and engineering architect …

and lead the graphics team for Destiny 2, shipping this year.

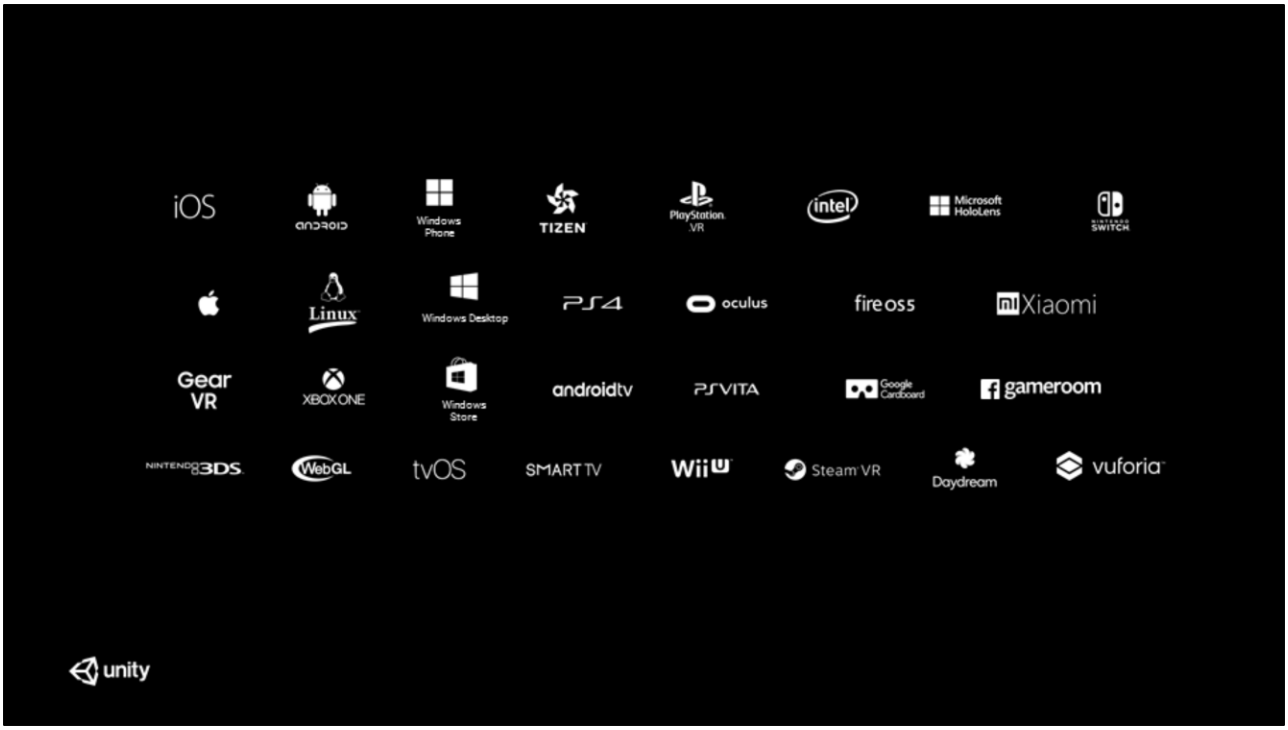
Before that, I led the graphics research and demo team @ AMD, helping drive and define graphics API such as DirectX 11 and define GPU hardware features together with the architecture team. Oh, and I developed a bunch of graphics algorithms and demos when I was there too.

At Unity, I am helping to define a vision for the future of Graphics and help drive the graphics technology forward. I am lucky because I get to do it with an amazing team of really talented folks working on graphics at Unity!

In today’s talk I want to touch on the programming models we use for real-time graphics, and how we could possibly improve things. As all in the room will easily agree, what we currently have as programming models for graphics engineering are rather complex beasts. We have numerous dimensions in that domain:

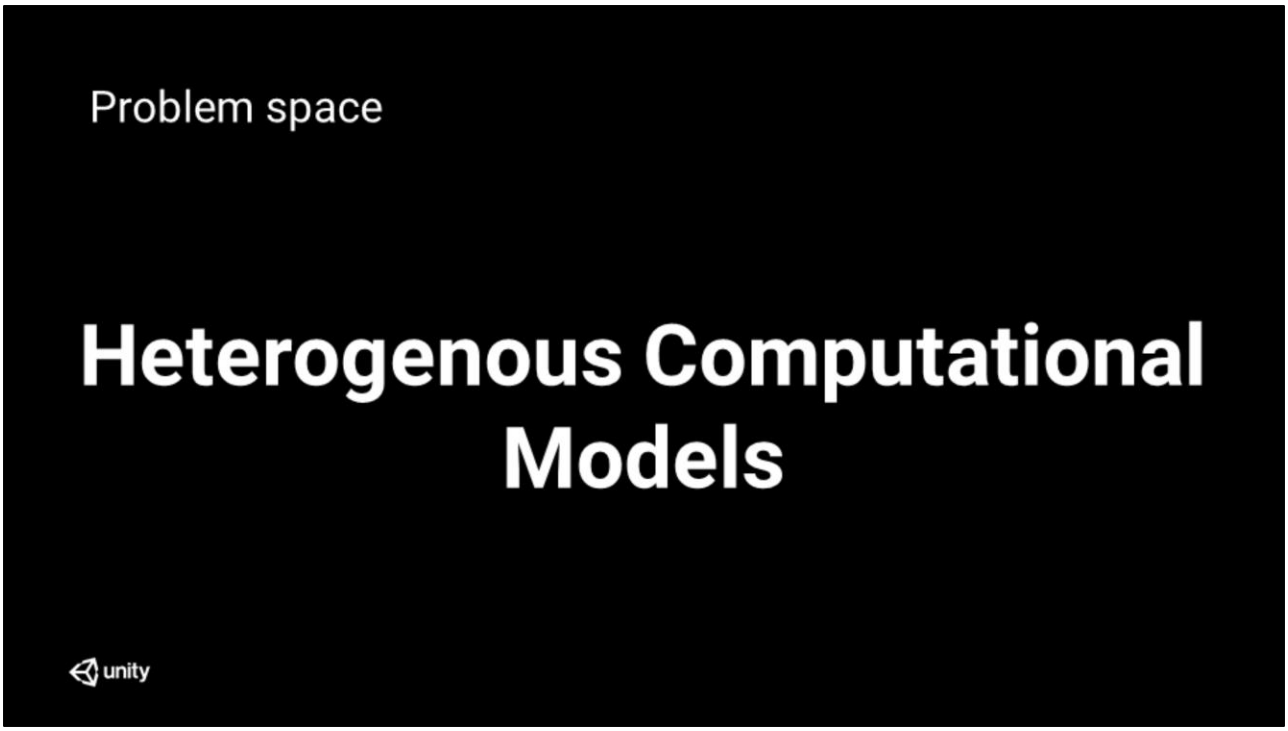


Model graphics programming lives on top of a very fragmented and complex platform and API ecosystem

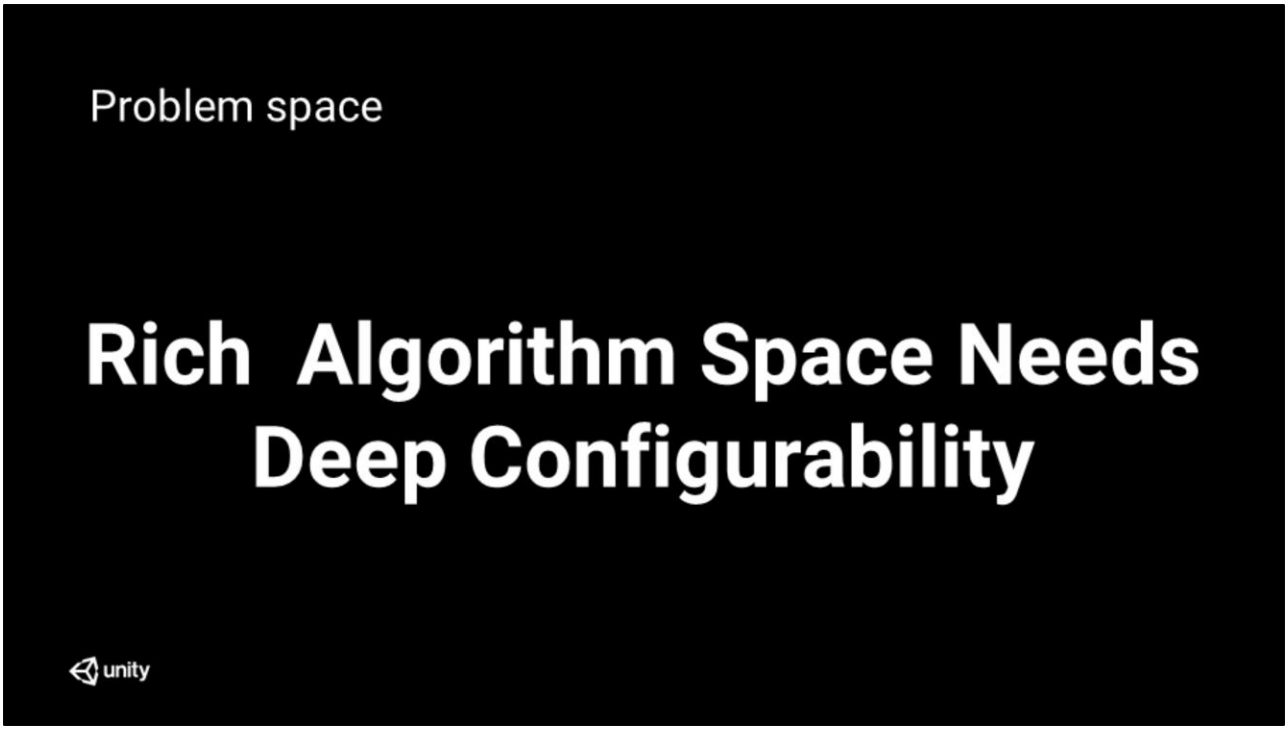


For example, this is snapshot of all the more than 25 platforms that Unity supports today, including PC, consoles, VR, mobile platforms – all with varied hardware, divergent graphics API and feature sets.

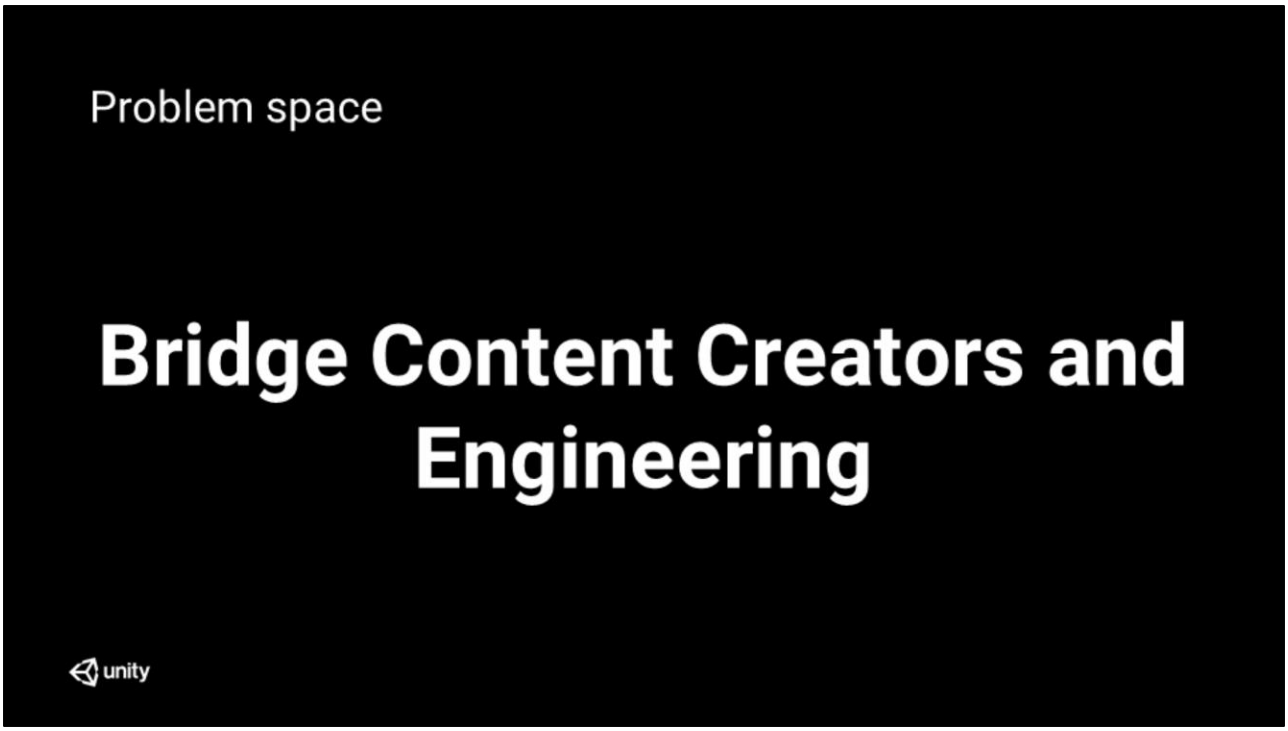
To do anything related to graphics, you have to somehow get the resources (textures, shaders, meshes, animations, etc.) . This often can mean a lot of boiler plate



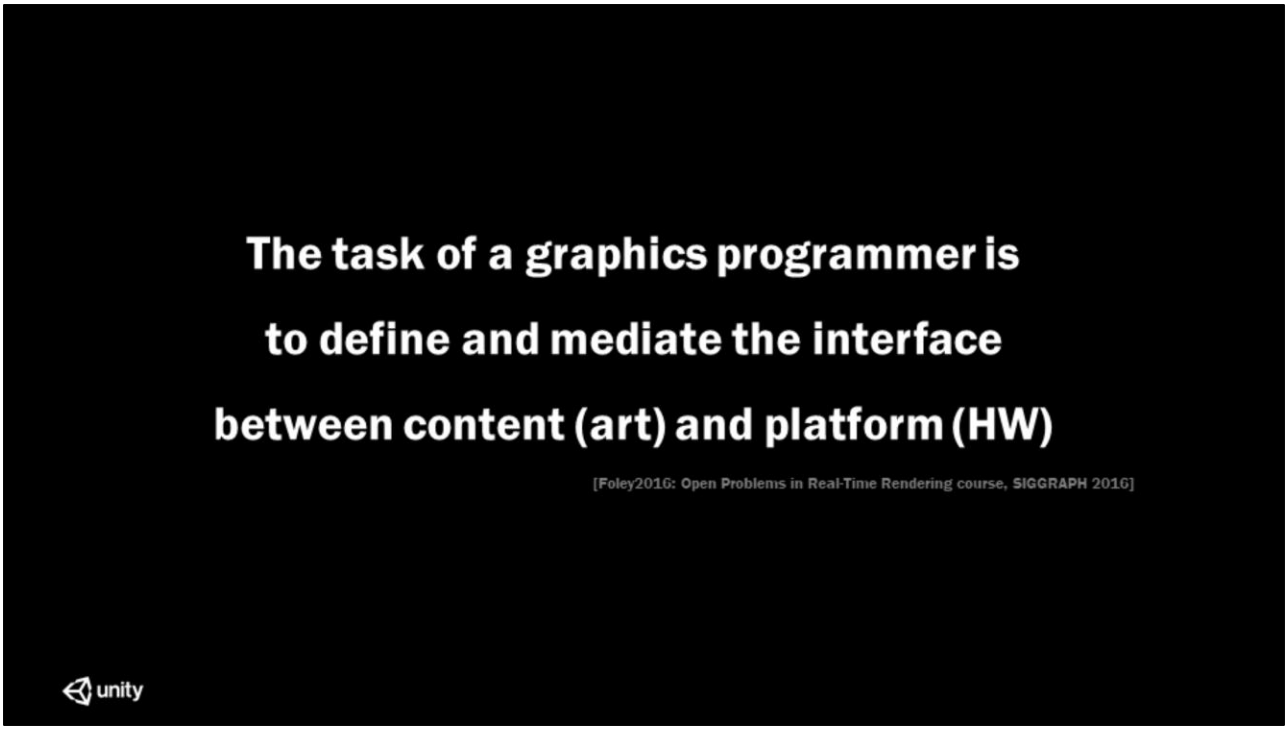
On top of it, we need to be thinking about programming to a heterogeneous machine – GPU and CPU have vastly different programming models and programming game and graphics engines needs to service all components. Ditto for any graphics algorithm research application.



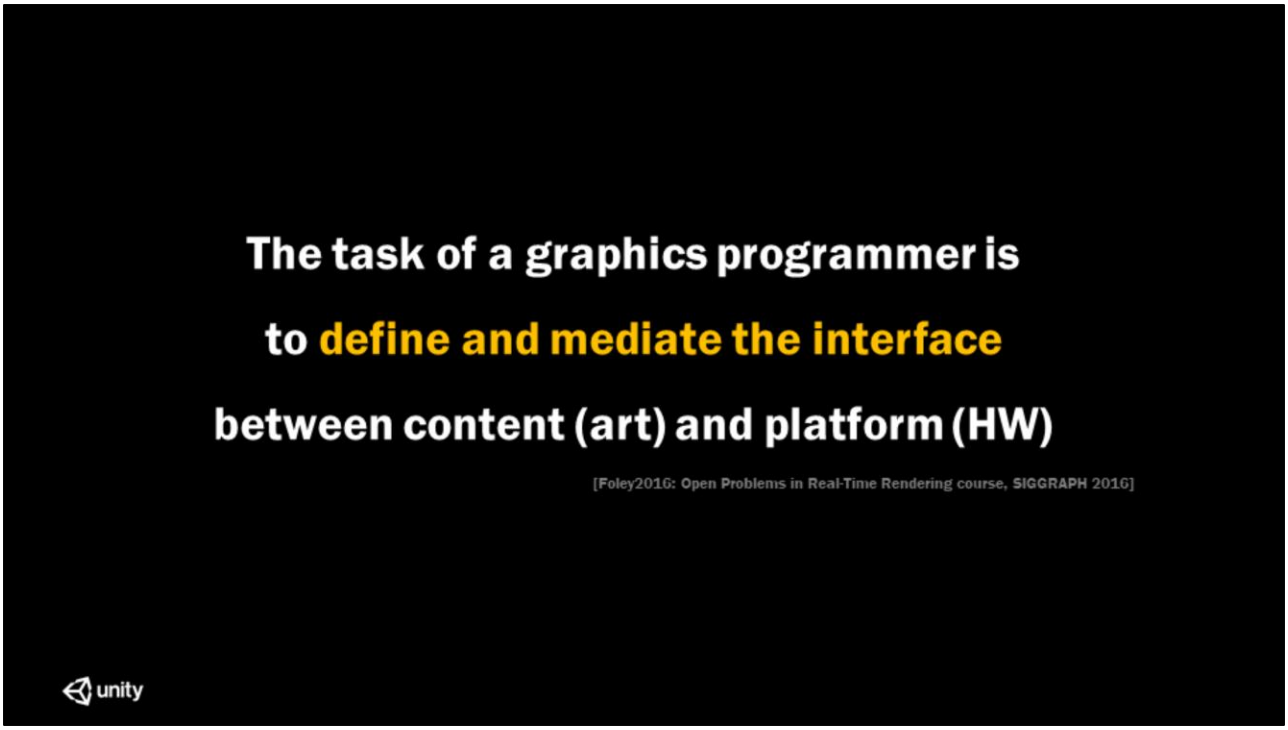
We now have an explosion of high-level pipeline algorithms used for real-time graphics – forward rendering, forward plus, deferred rendering, clustered algorithms, z-binning, GPU occlusion management, variety of tiled approache, like fine-prune tiled lighting examples, and so forth. All of those have their utility for specific pipeline goals, for example, VR benefits from single-pass forward renderer, while many games on consoles employ deferred rendering. All of these algorithms require deep configurability of the underlying engine if it is to support it.



Of course, game engines and graphics applications often serve as a bridge between content and engineering ..

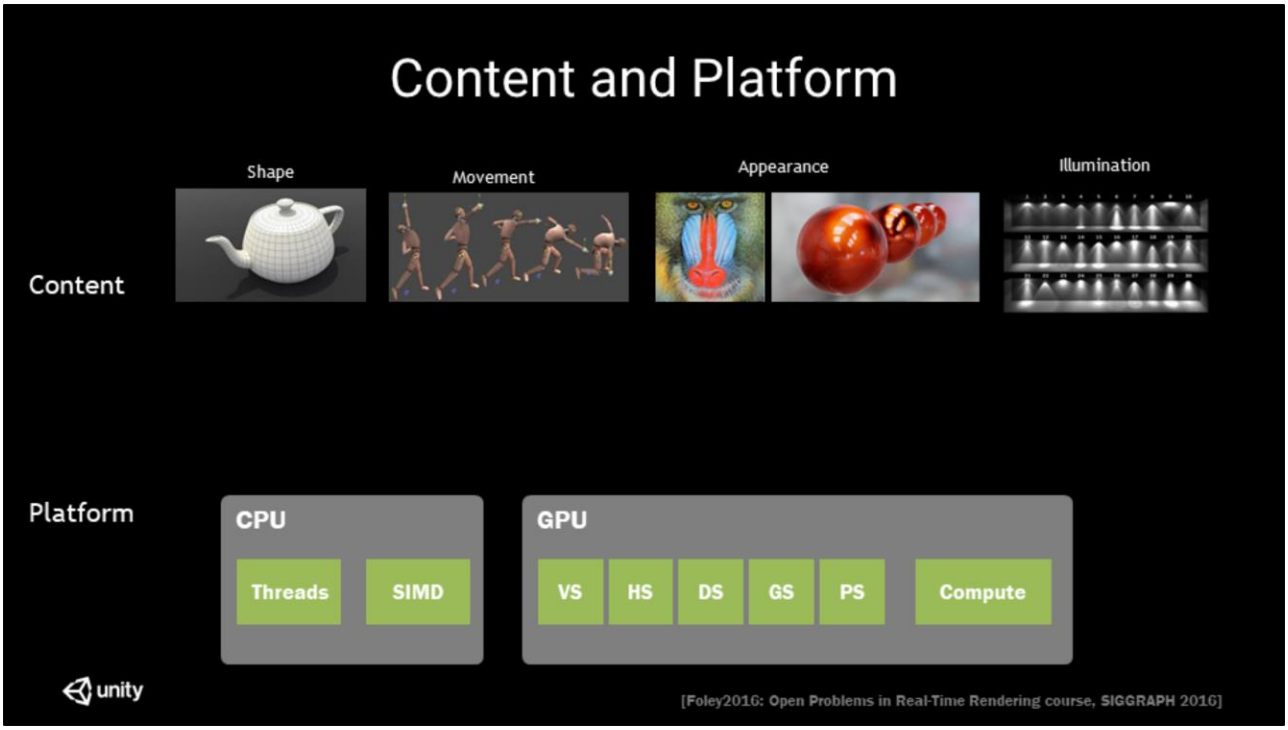


Tim Foley has made a point about this particularly well in his talk in the Open Problems at SIGGRAPH last year
We can think of the goals for graphics programmer – in other words, what we are all trying to do –



is to *define* and *mediate* the interface between art content, and one or more HW platforms.

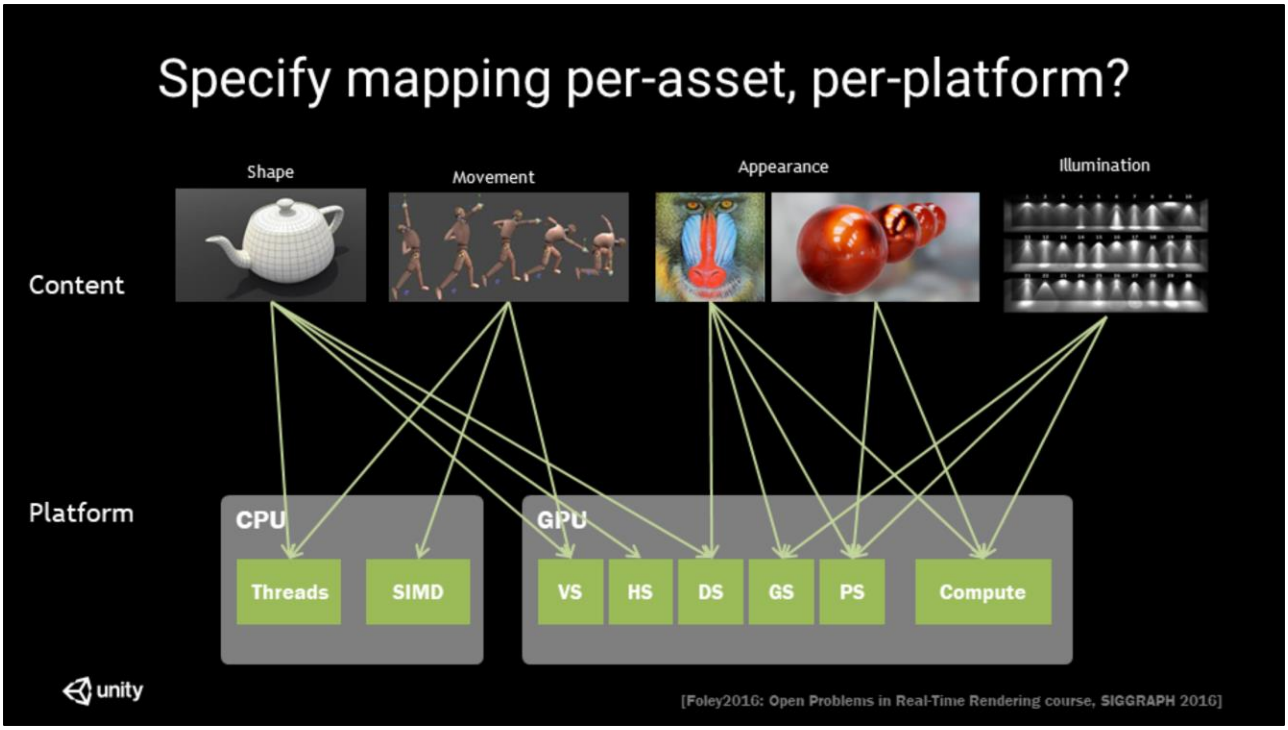
Let’s look at some pictures to make that a bit more concrete…



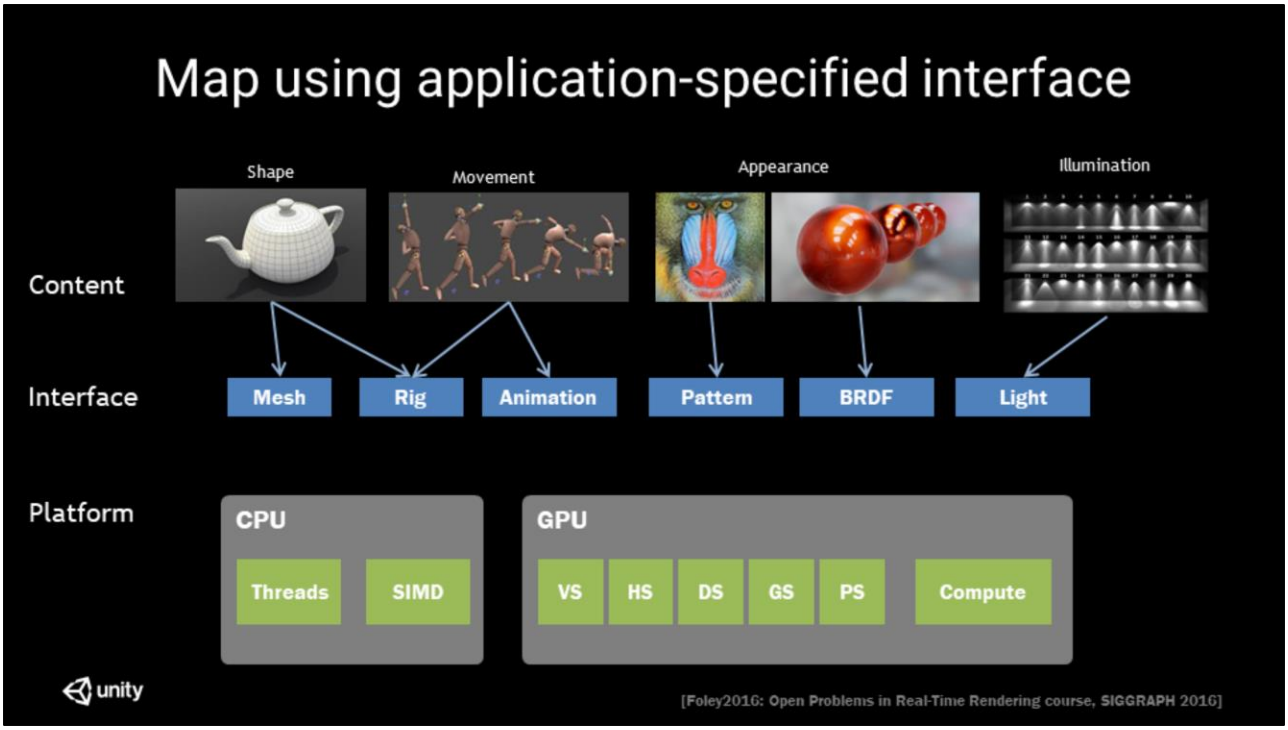
Here at the top of the slide we have *content*, which is all the concepts that artists are going to understand and author. We have stuff like shapes, movement, appearance, and illumination.

At the bottom we have the *hardware platforms* that we target, which these days look kinda similar: we have CPU cores with support for threads and SIMD, and GPU hardware that supports both a traditional rasterization pipeline, and some kind of general-purpose compute.

In order to get the stuff an artists produces up on the screen, we need to have a plan for how to map the concepts in the top row to the hardware platform on the bottom.



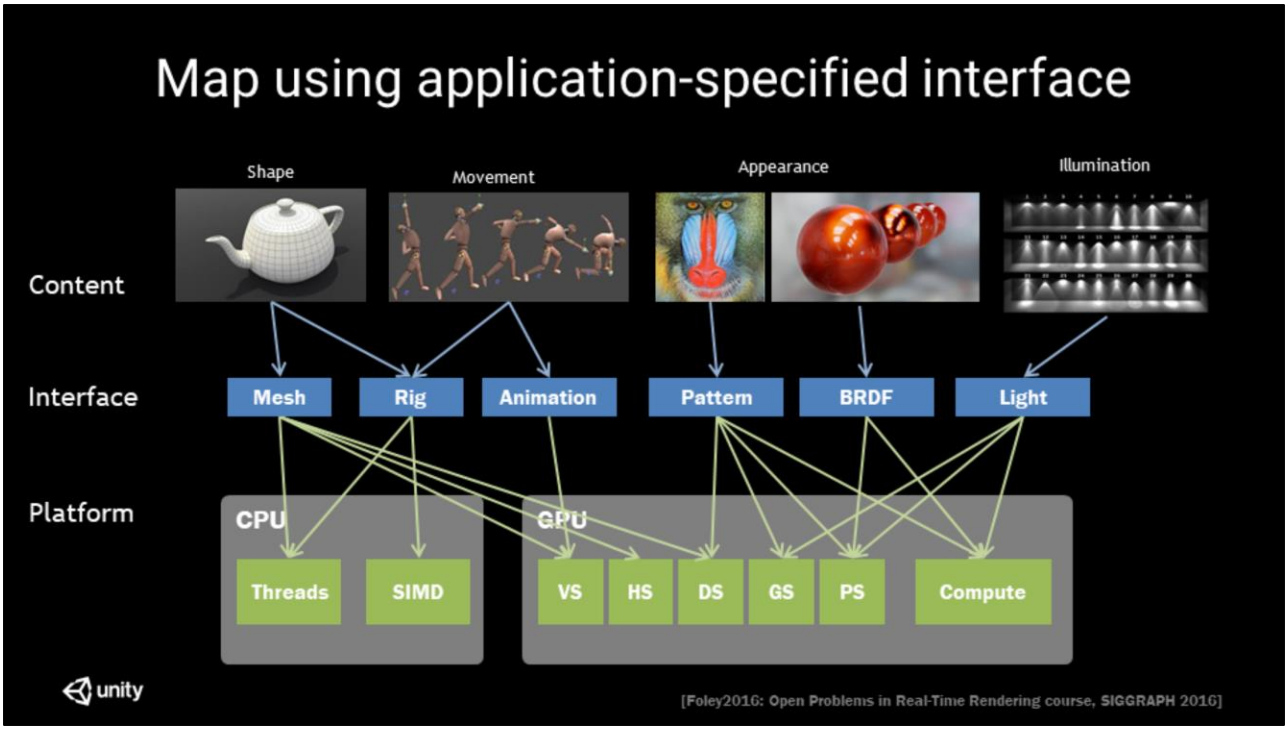
One (bad) way we could do that is by writing code on a per-asset, per-platform basis.
This obviously wouldn't scale, and so it isn't what we actually do.
And that gets us to the heart of the definition…



What we claim we actually do in practice is this:

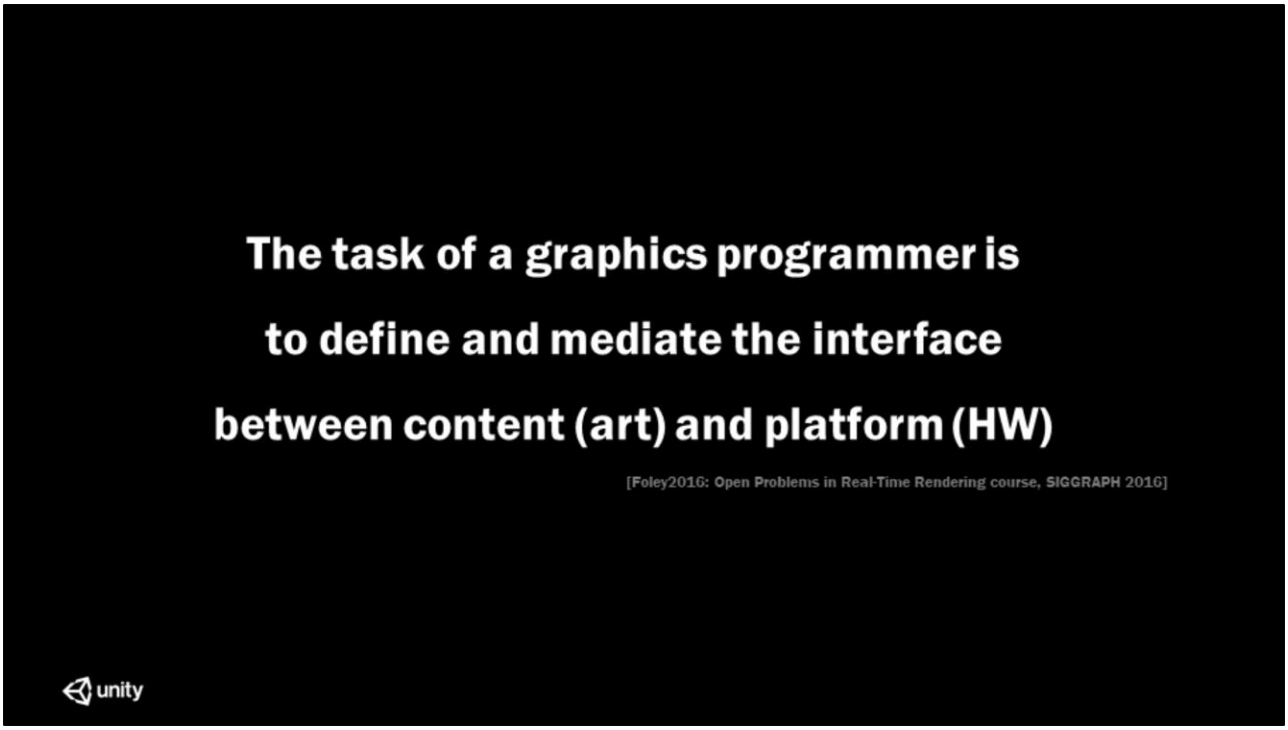
A graphics programmer defines an *interface* - a set of concepts that are specific to a particular engine or production.
This is how we will express an animation rig.
This is how we will represent reflectance functions.

Some of the representations in that interface might be pure data (e.g., for a mesh), and others might include code (e.g., if artists describe pattern generation as a “noodle graph”).



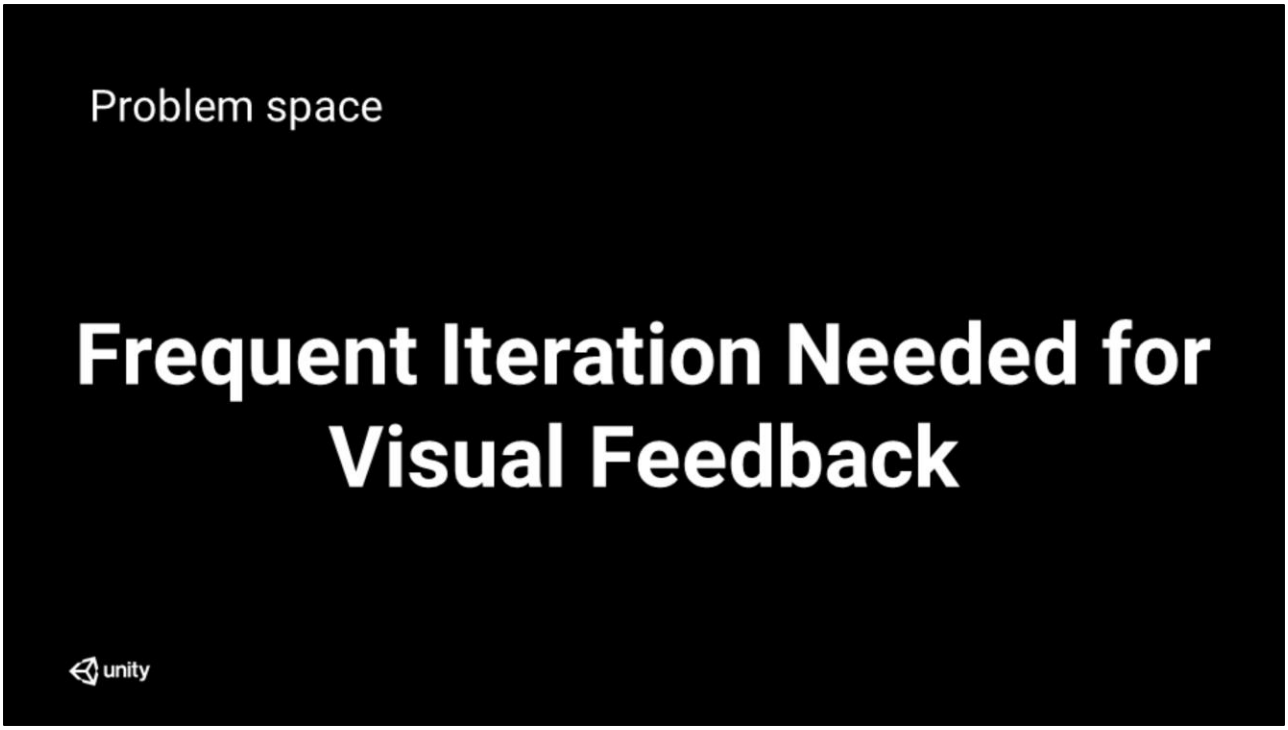
Either way, the task is to:

- map the assets produced by artists so that they conform to the chosen concepts in the interface, and
- map those concepts efficiently to one or more target platforms

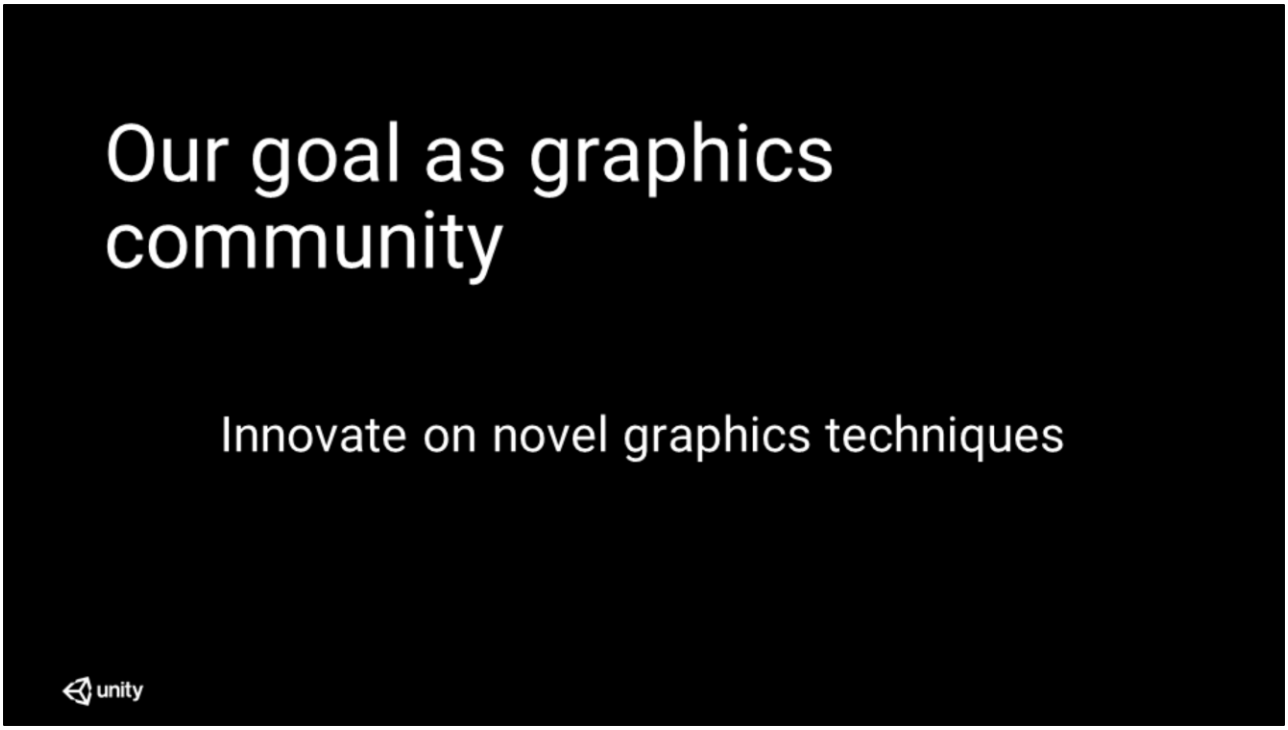


Now I’m coming back to this definition, and I hope it makes a bit more sense now what I mean.

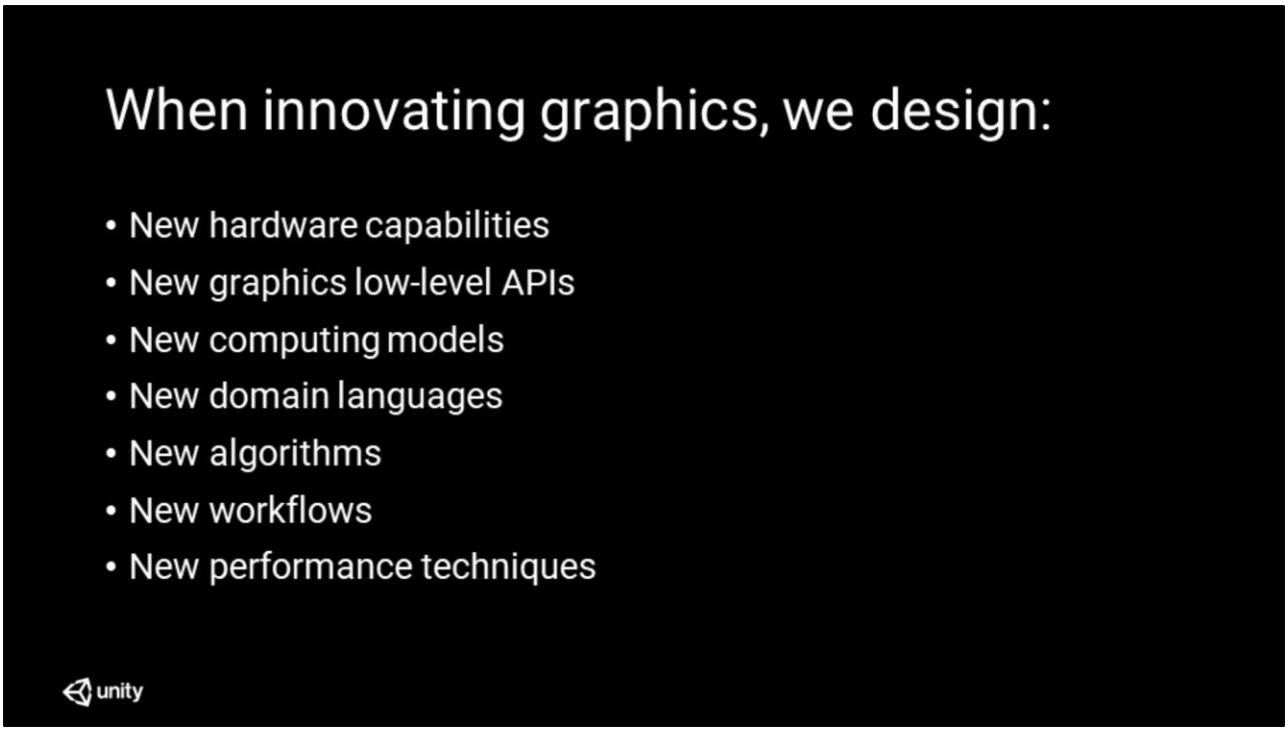
The task of a graphics programmer is to define and mediate that interface between content and the platform(s).



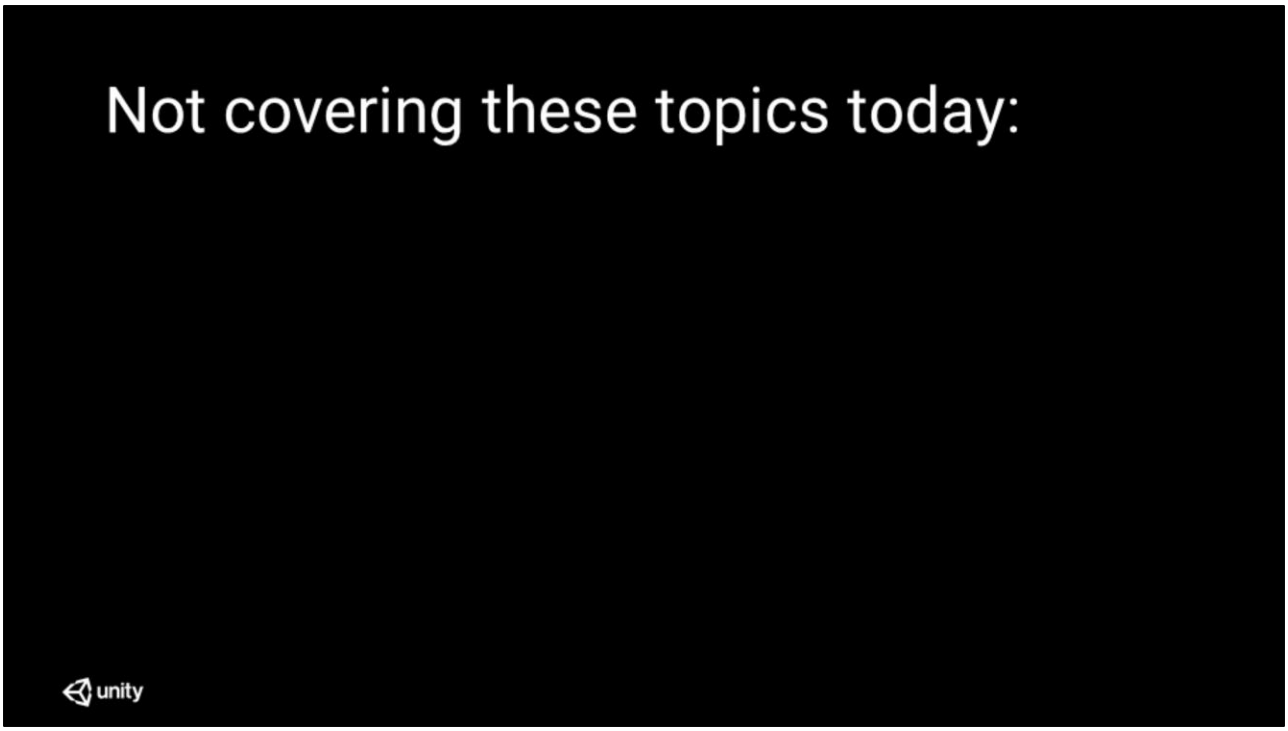
And of course, the more we can iterate on a technique, typically, the better it becomes. Thus, a must for *success of the developer directly* is their own iteration loop.



What is our goal, as graphics programmers, ultimately? Quite simply put, our goal as the graphics community is to innovate on graphics techniques



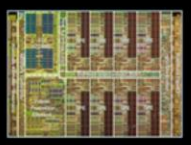
When doing that, we focus on
- Designing new hardware capabilities (CPU / GPU / etc.)
- Designing new graphics low-level APIs (ex: DirectX12, Gcn, etc.)
- Design of heterogeneous computing models
- Design new domain languages (HLSL, Cg, etc.)
- Design new algorithms (forward- versus deferred, FPTL, clustered, etc..)
- Design new workflows (content authoring: procedural vegetation placement, LOD systems, etc., and development workflows: developing the actual algorithms)
- Design new performance techniques

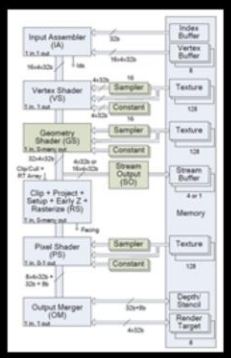


Because today I am focusing on the programming models for development in graphics / game engines, I will not cover the evolution or changes of ..

Not covering these topics today:

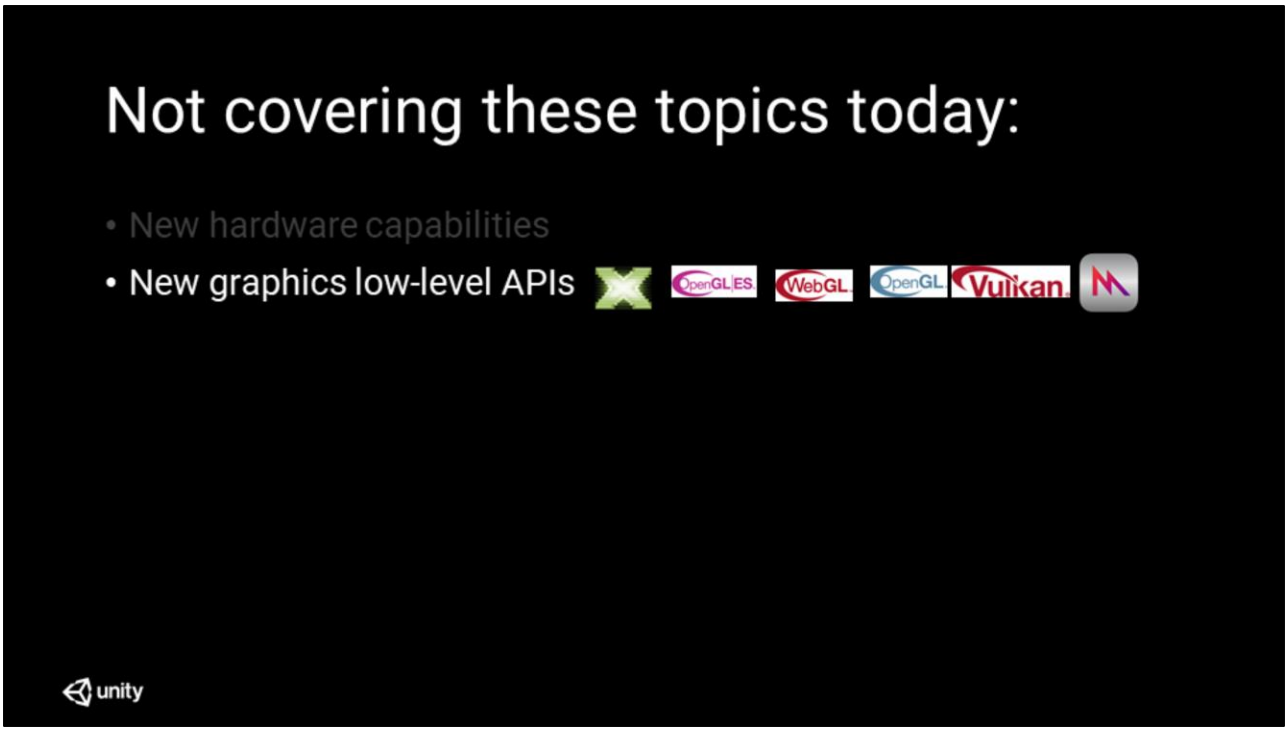
- New hardware capabilities



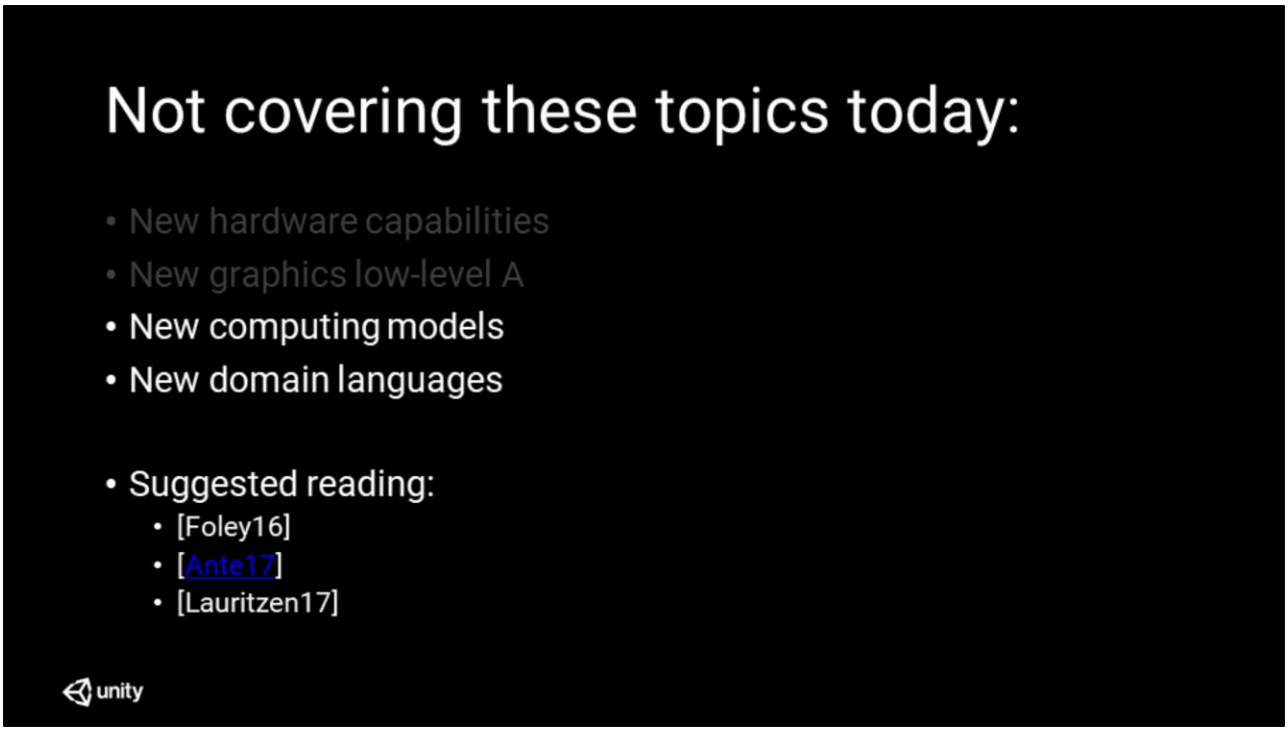


unity

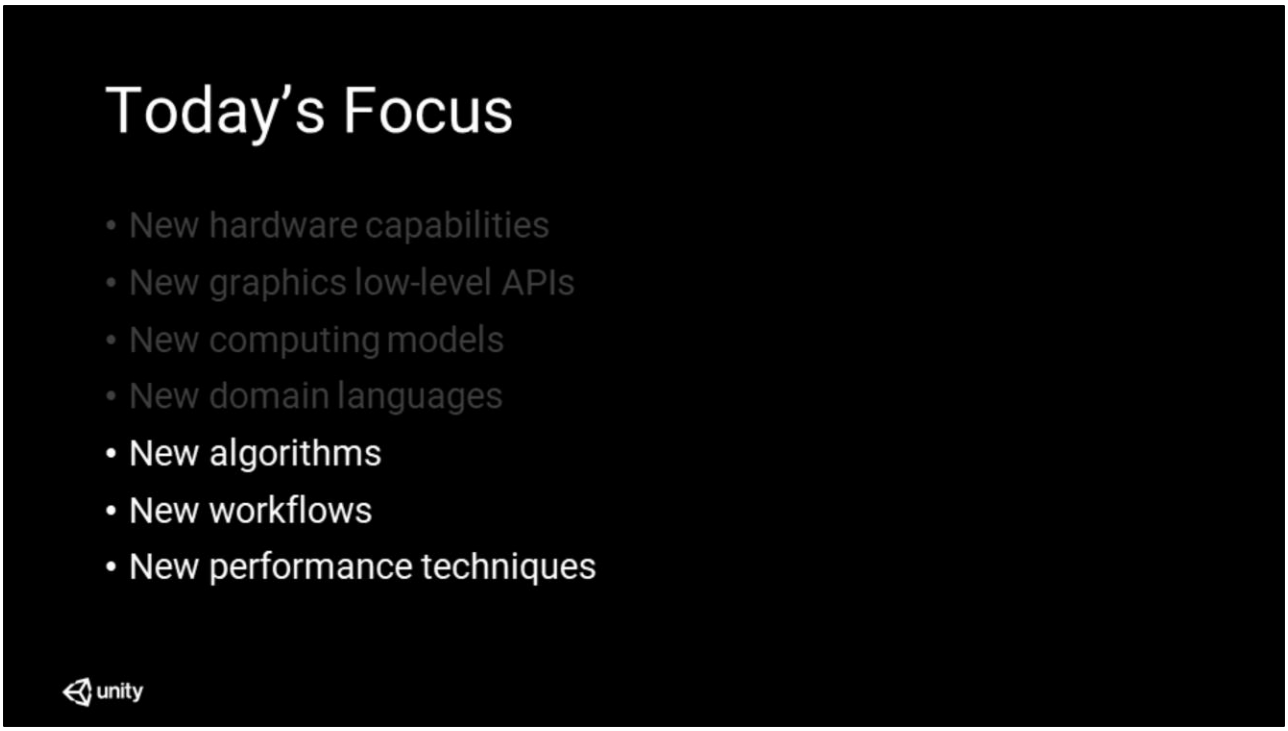
hardware capabilities or hardware programmable pipelines



Evolution of graphics low-level APIs, …



I am also not delving into new computing models, for example, how do we target efficiently cross CPU/GPU computation, or design of new domain languages. This is absolutely a very important problem space to innovate in. In fact, at Unity, we are actually doing active exploration of improving the programming model for this domain - with our development of C# to IL compiler (P2GC) to support C# jobification API, which allows writing thread-safe, concurrently executing jobs with a very simple API in C# (with all the benefits of C# iteration model), that, using this compiler, execute faster than native C++ code. This is done by compiling to super-vectorized efficient IL. But as exciting as this project is, it's not what I will focus in today's talk. But feel free to check out the reference from Joachim Ante's talk at this year's Unite.

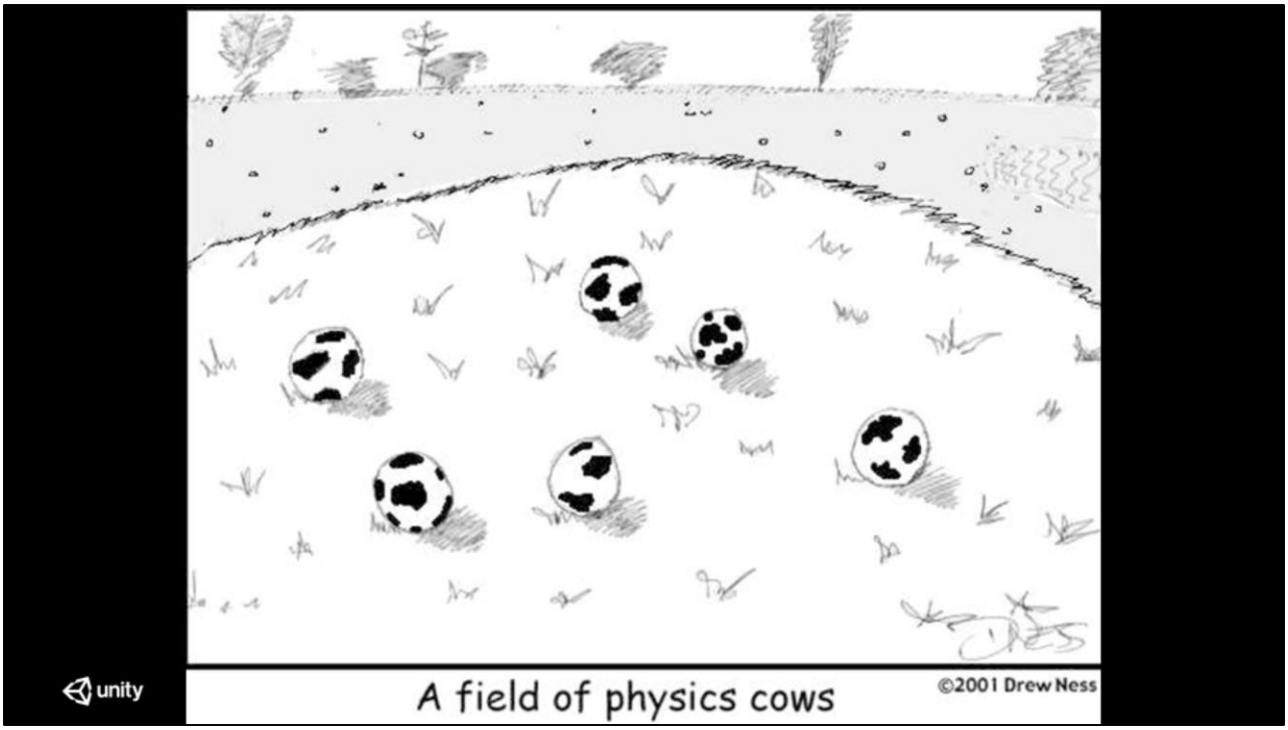


Today I want to discuss what we can do to enable better development of new algorithms by using new development workflows in graphics engines. In other words, the evolution of programming models for graphics game engines for designing new algorithms and rendering pipelines.

In other words, how we innovate on graphics *techniques, usability,* and *performance*

While constraining the other variables

To move forward, everything needs to define and understand its choice of constraints



The famous spherical cow of physics. What are our spherical cows?

Why are constraints important for evolution of programmable models?

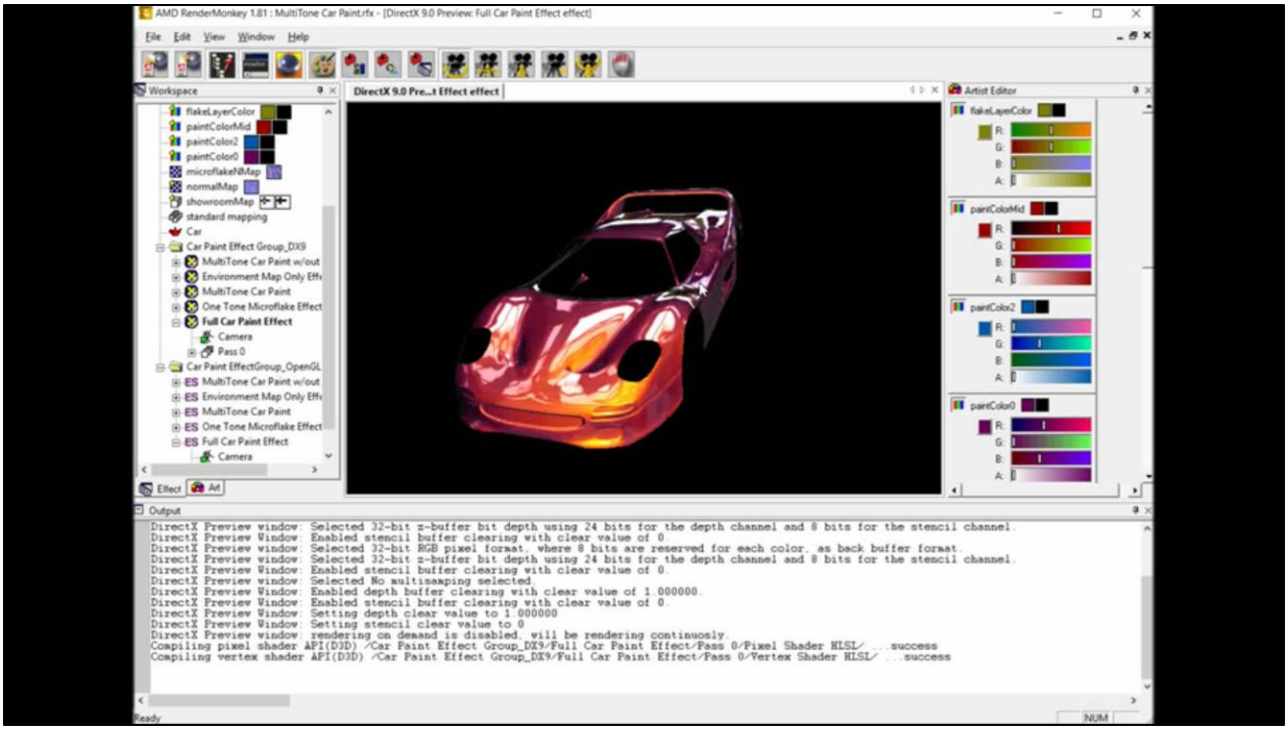
Constraints allow you to lock an axis and focus on evolving the rest of the stack
Low-level APIs constraint some aspects and focus on accessibility of others
Example: compute models focusing on accessibility of wavefront scheduling / dispatch, not on convenience of scheduling management – constraint is that GPU manages the exact details of the scheduling ,and we just focus on writing the kernels we're executing on the GPU
Domain specific languages benefit greatly from constraints – example: HLSL lacking general C++ programming features (allowed huge innovation in shader development by targeting the hardware directly)

Constraints Benefits

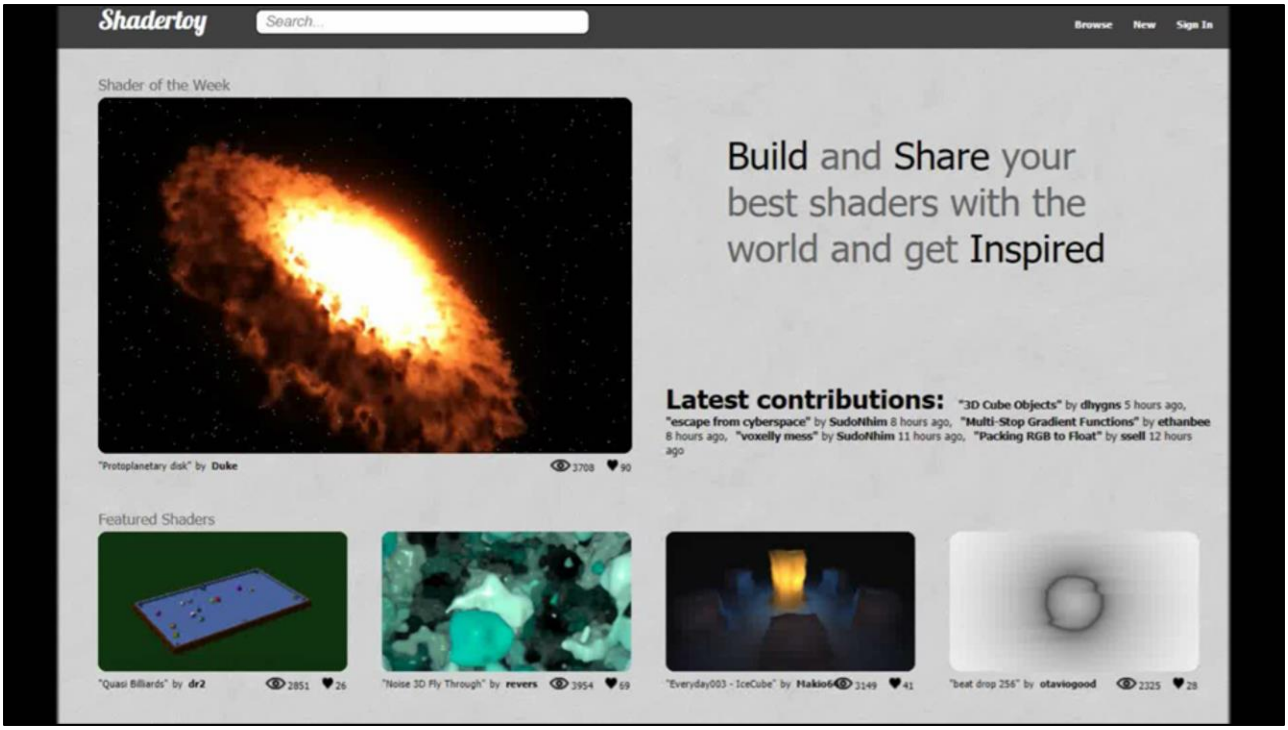
- Lock an axis and focus on evolving the rest of the stack
- Software architectures often use the same principles
- Constraining some aspects allows fast innovation in that domain

unity

Software architectures often use the same principles
Constraining some aspects allows fast innovation in that domain

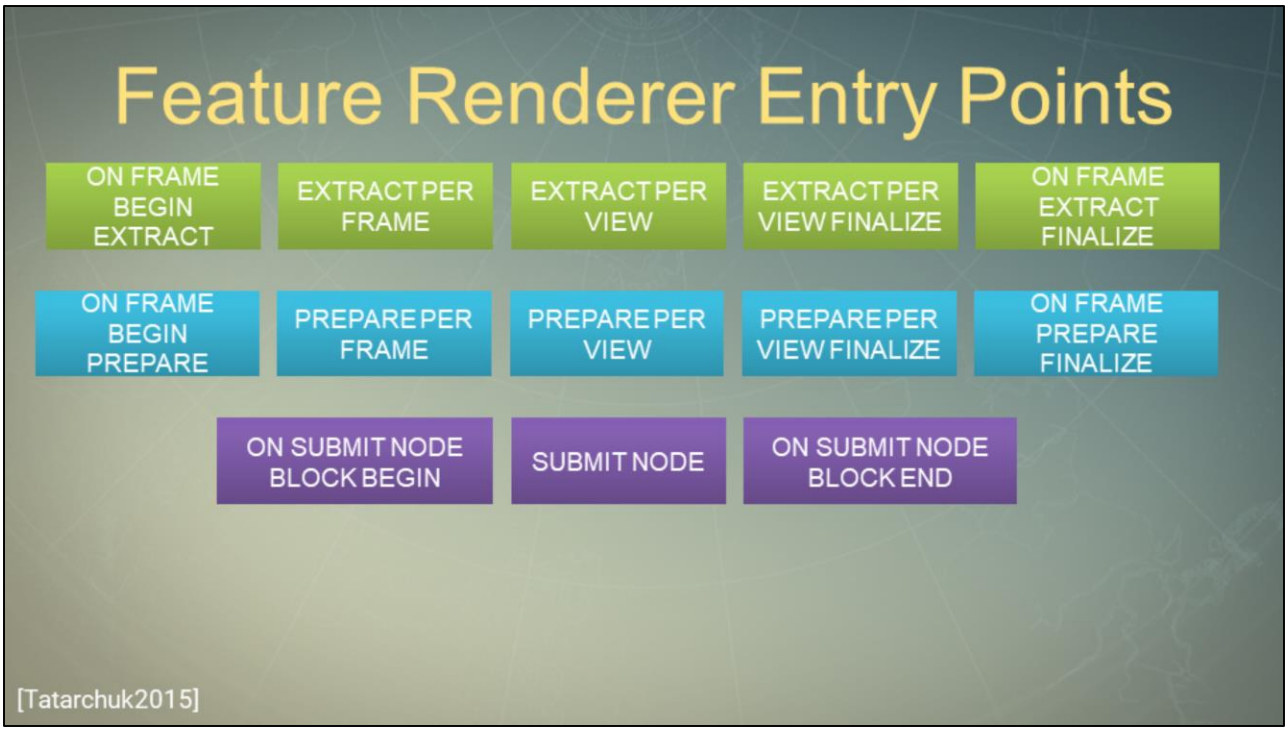


RenderMonkey (http://gpuopen.com/archive/gamescgi/rendermonkey-toolsuite/), which is over a decade old now, was a great example of constraining everything about the program except authoring pixels and vertex shaders, and then vastly simplifying the workflow for working on all the scaffolding elements required to execute these programs. It made it very easy to load geometry models, textures, create passes, etc. and as a result we saw a great deal of shader techniques being developed in RenderMonkey (even still people email about working in it).

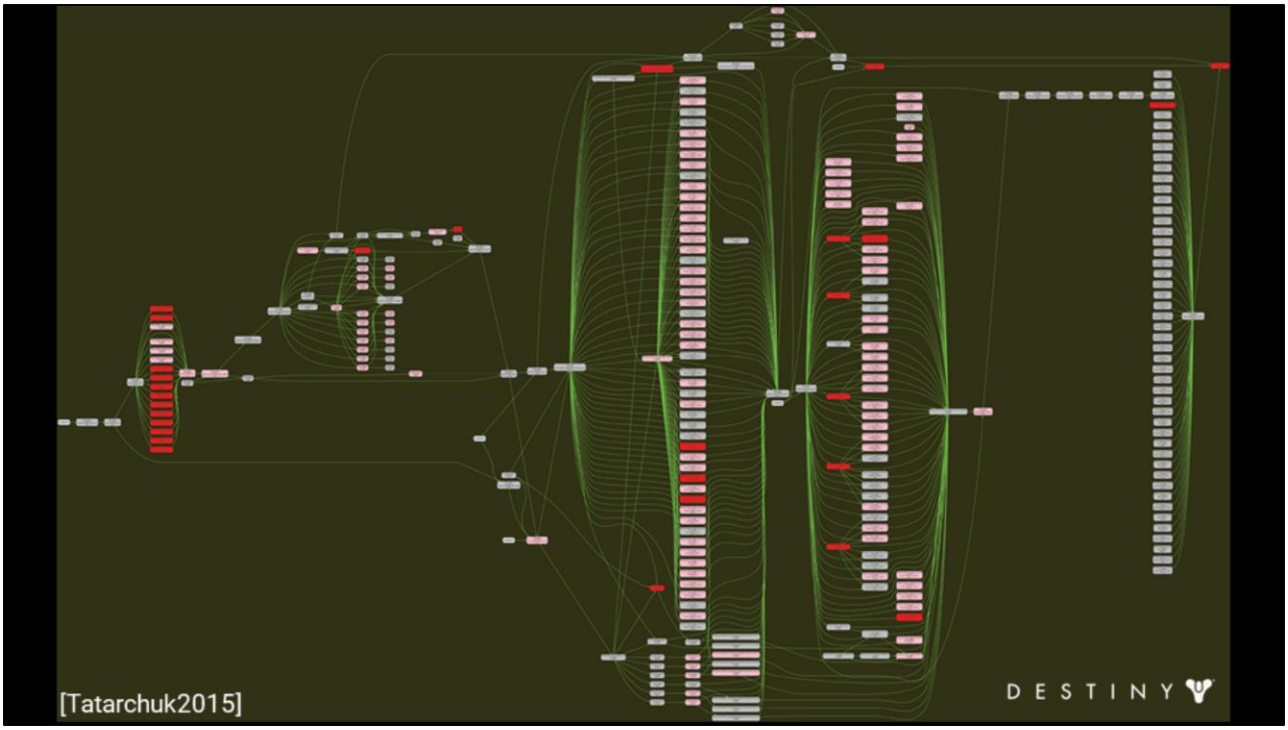


ShaderToy (https://www.shadertoy.com/) is another great example of constraining the programming model (you only work in the pixel shader) and as a result saw an explosion of amazing innovative techniques- if you haven't seen it, I urge you to check it out (there is even a competition this siggraph)

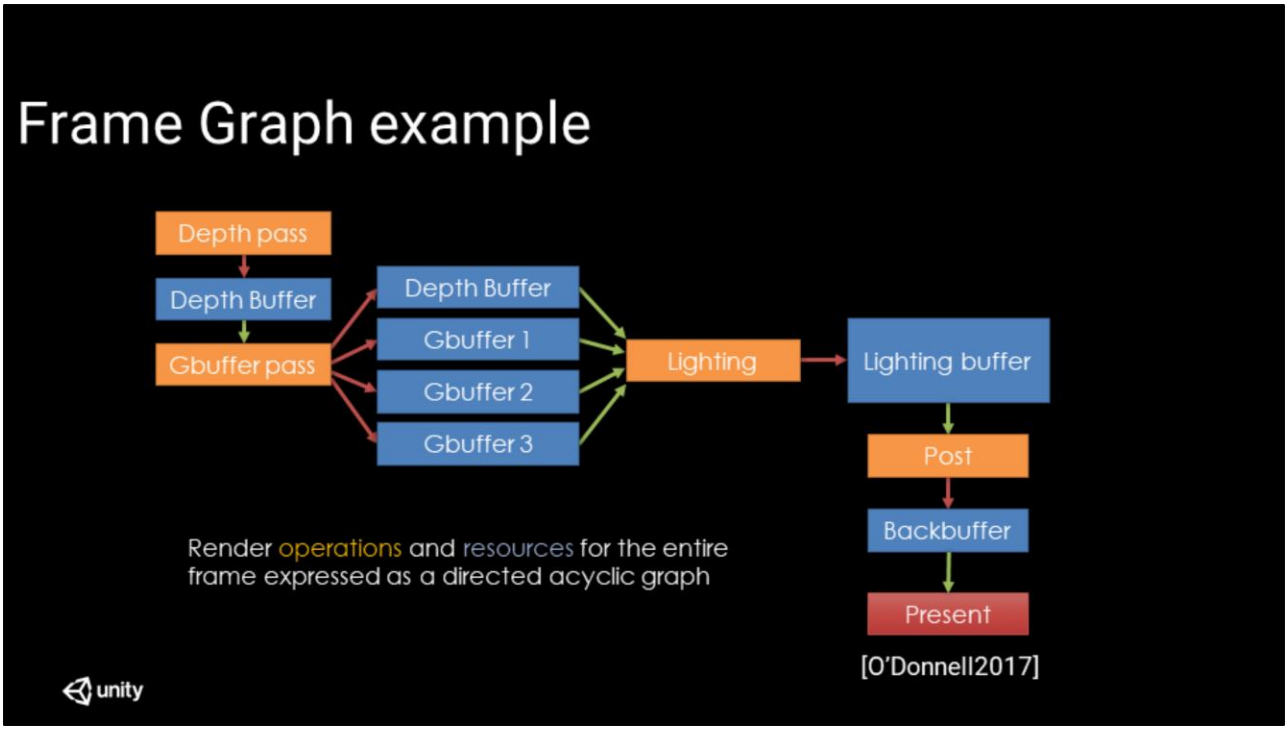
Destiny feature renderer architecture [GDC 2015, http://advances.realtimerendering.com/destiny/gdc_2015/index.html] was another example, where



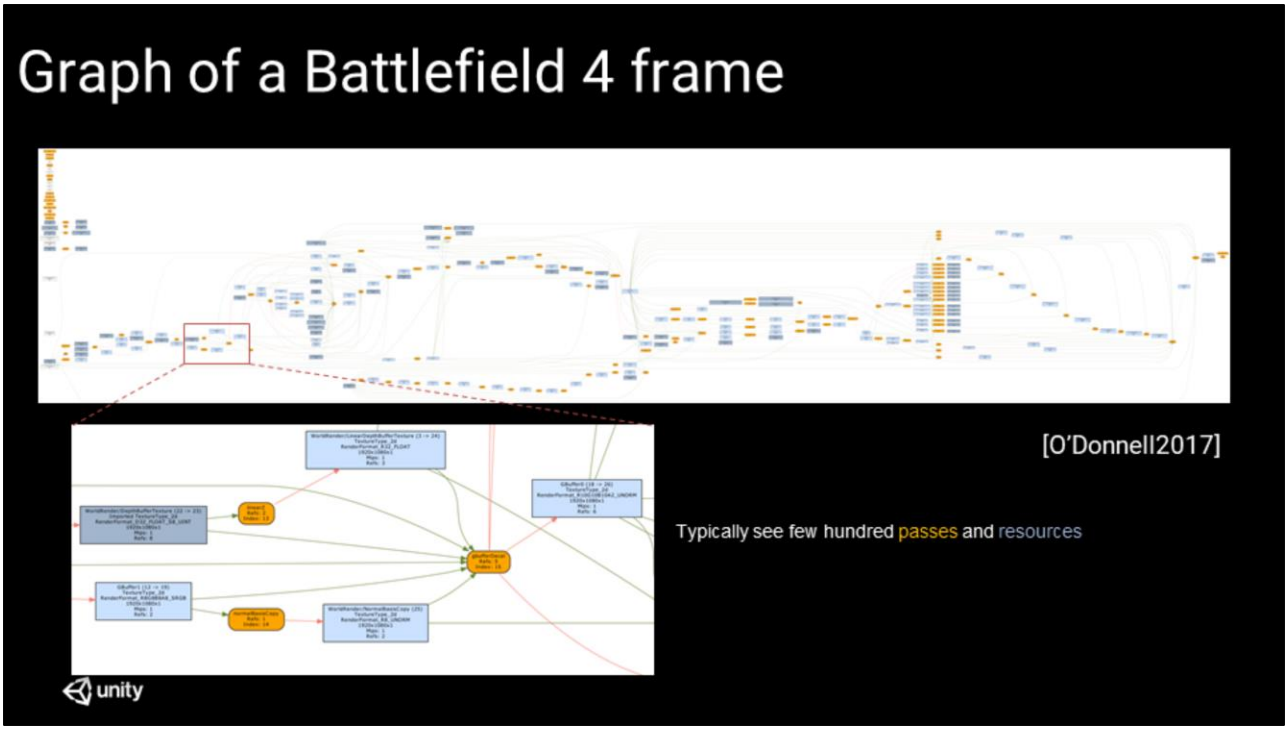
By providing a constrained API for writing guaranteed thread-safe units…



It enabled fast iteration for writing multi-threaded graphics features (the feature renderer architecture)
Each node in this diagram is an actual job (this is a shipping frame from the Destiny Cosmodrome level on Xbox One)

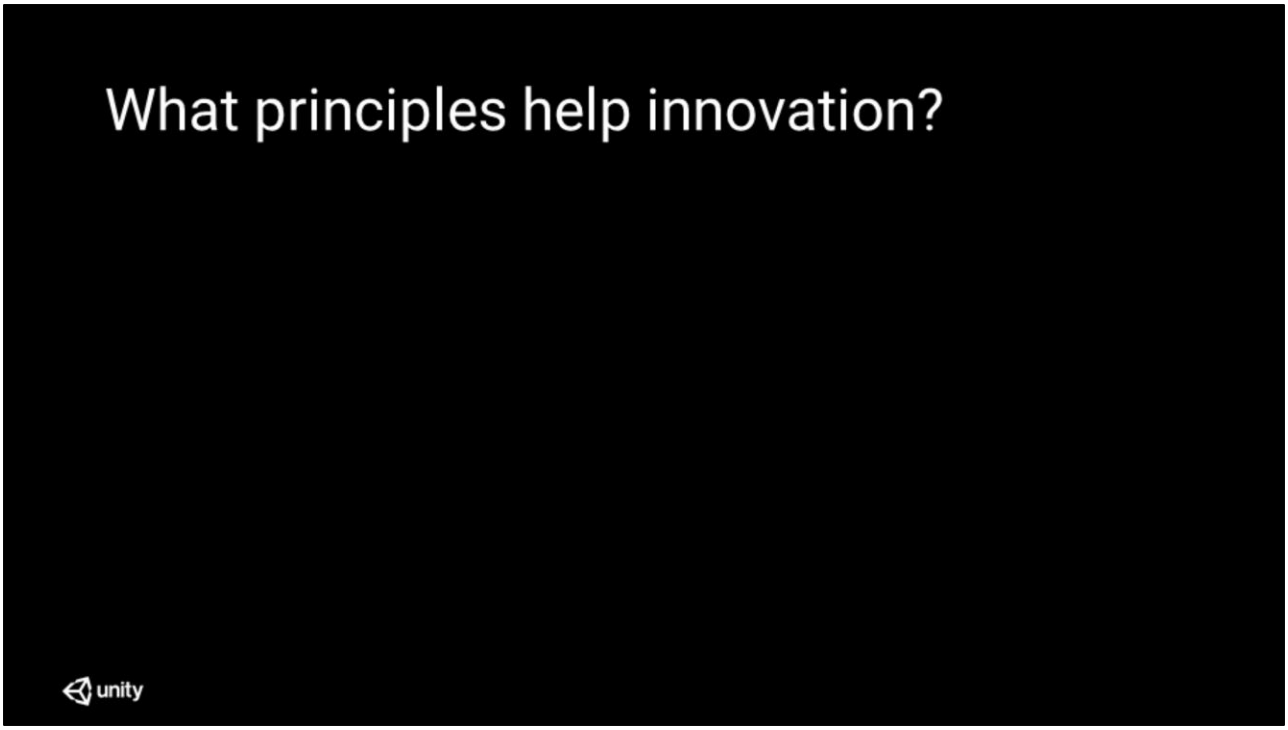


DICE FrameGraph [O'Donnell 2017]: Providing a focused constrained API for specifying render passes allowed easy iteration for virtualizing resource management
Here is a toy example of a frame graph that implements a deferred shading pipeline from Yuriy O'Donnell's presentation at GDC 2017.
The graph contains render passes and resources as nodes.
Access declarations / dependencies are edges.



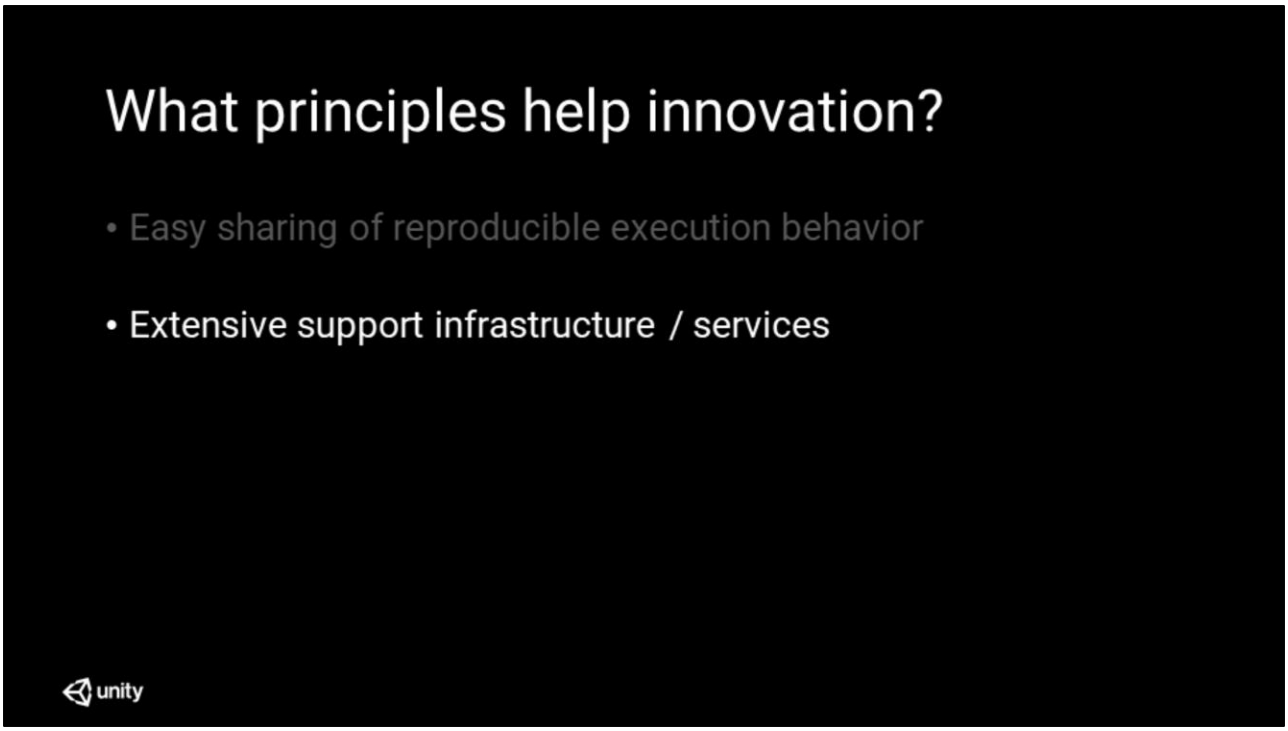
This architecture yielded memory savings due to resource aliasing, aided design of async compute execution, and simplified DICE's render pipeline configuration. Note, however, they still programmed this in C++

Of course there are numerous examples of such constrained innovation jumps throughout all of the graphics field.



What are the properties of frameworks that support better innovation?

Easy sharing of reproducible behavior (for example, sharing full shader packages for RenderMonkey shaders with code and content) speeds up learning throughout the community. Of course the best is end-to-end runnable executable with full content



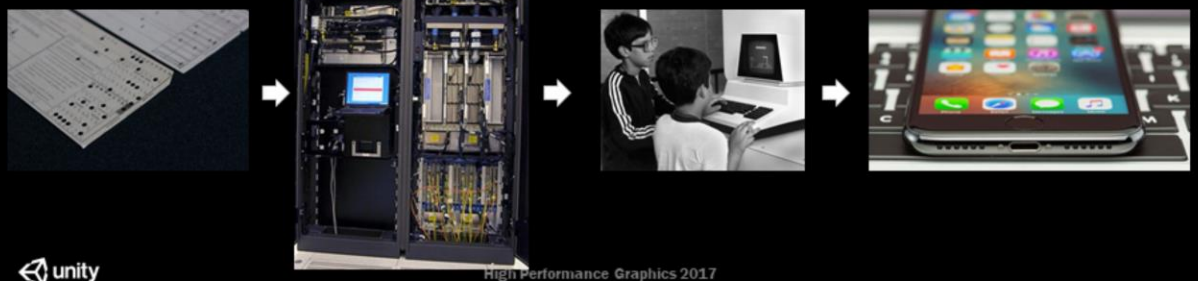
Frameworks that help innovation should provide solid infrastructure that allows users avoiding boiler plate code. For graphics, this means providing asset and object management, threading, low-level platform support, etc.

Well, the good news is we are now at the point where the game and graphics engine provide that functionality directly. For example you can easily use the free Unity personal edition and distribute projects with full content, where the engine will manage object loading / unloading, resource management, like creation of textures, shaders, meshes, etc., and runtime layers for rendering.

But is that enough? Maybe. But what's also important is the while the frameworks like game engines give you all that support structure, they also need to not constraint you. In some ways, having to modify monolithic, heavy-weight complex engine architecture layers can be quite inhibiting to innovation, as the amount of work you have to sync to modify the layers can be disproportional to the solving of the problem you've set out to do.

Mustn’t Let Constraints Become Shackles

- Mostly care about constraints when we can’t handle the whole enchilada
- As capabilities grow, the trend is always toward generalization
  - To ease development



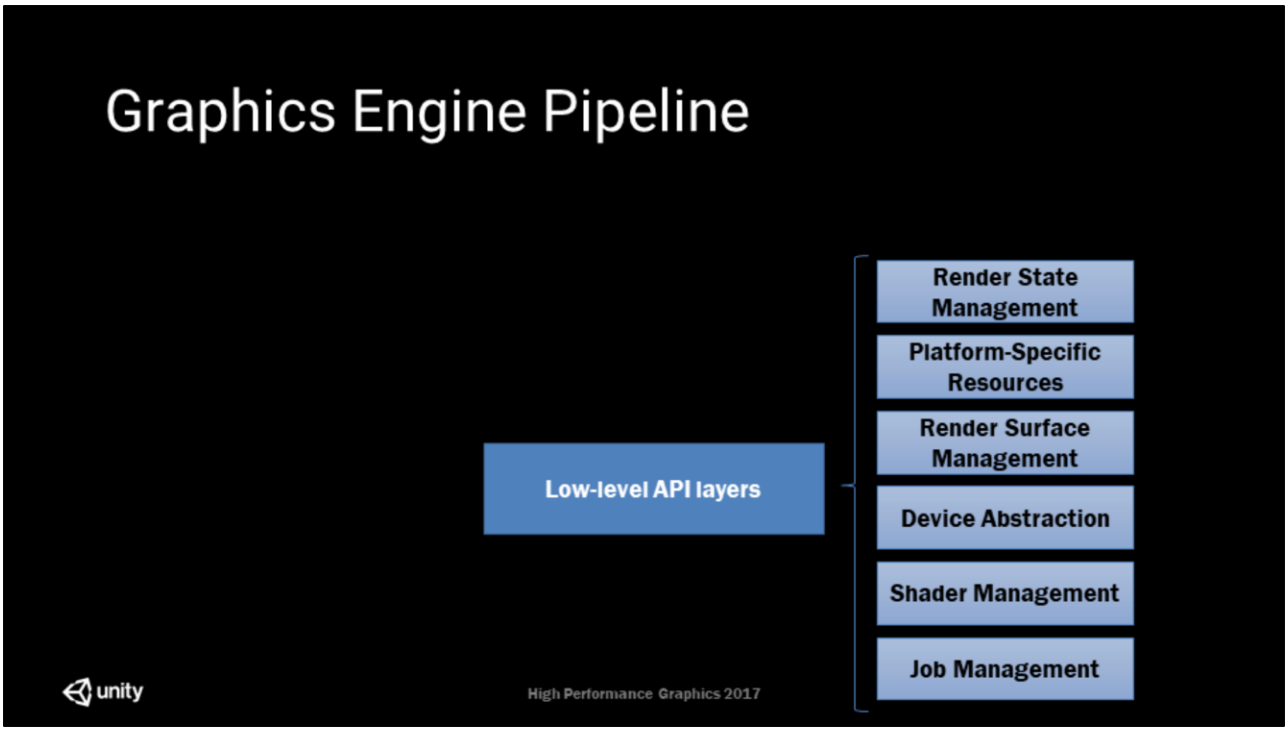
unity

High Performance Graphics 2017

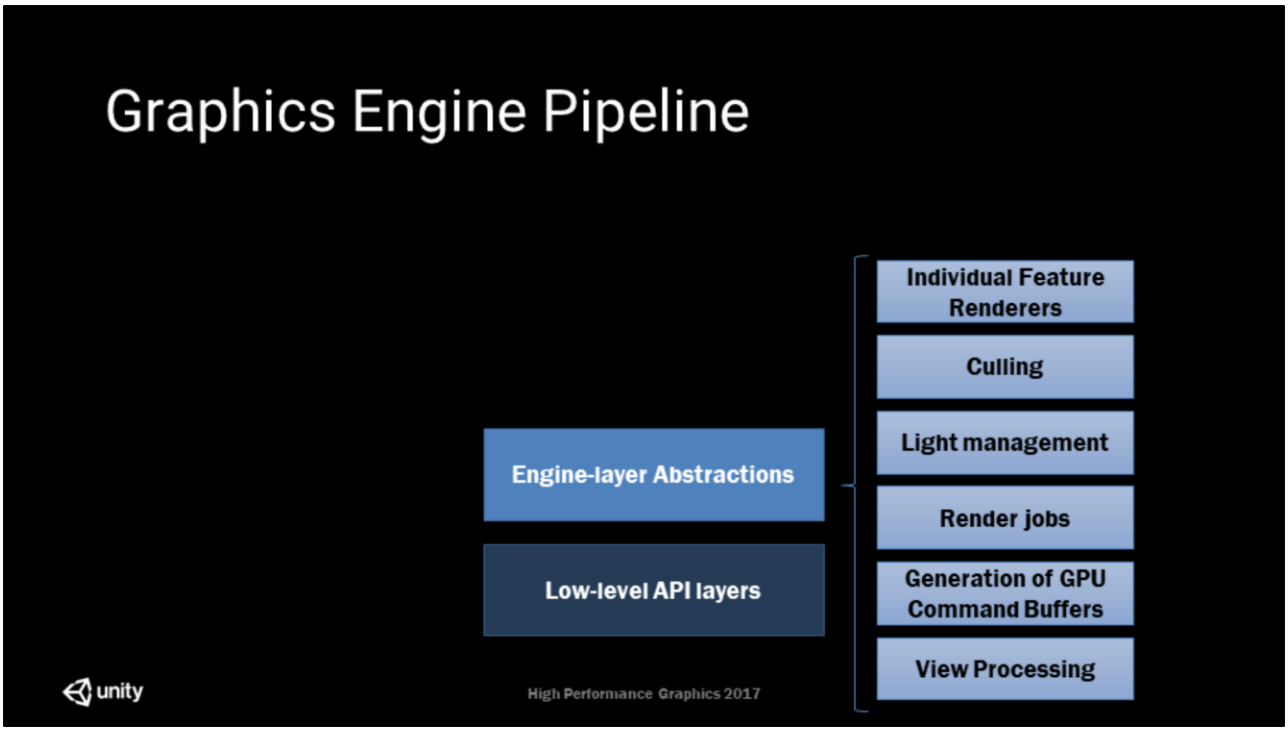
With respect to the constraints themselves, we also should note that all constraints are good when the problem is too big to address (thus the spherical cows of early grades physics). We note that as system capabilities grow, we always generalize to create more flexible models because our system now supports this amount of flexibility (due to growing computing power, for example), which makes development more accessible.

Coming back to our discussion, namely the evolution of programmable models in graphics engines to help with innovation, what we need to do is use Opportunistic Generalization for better workflow and process when performance matures.

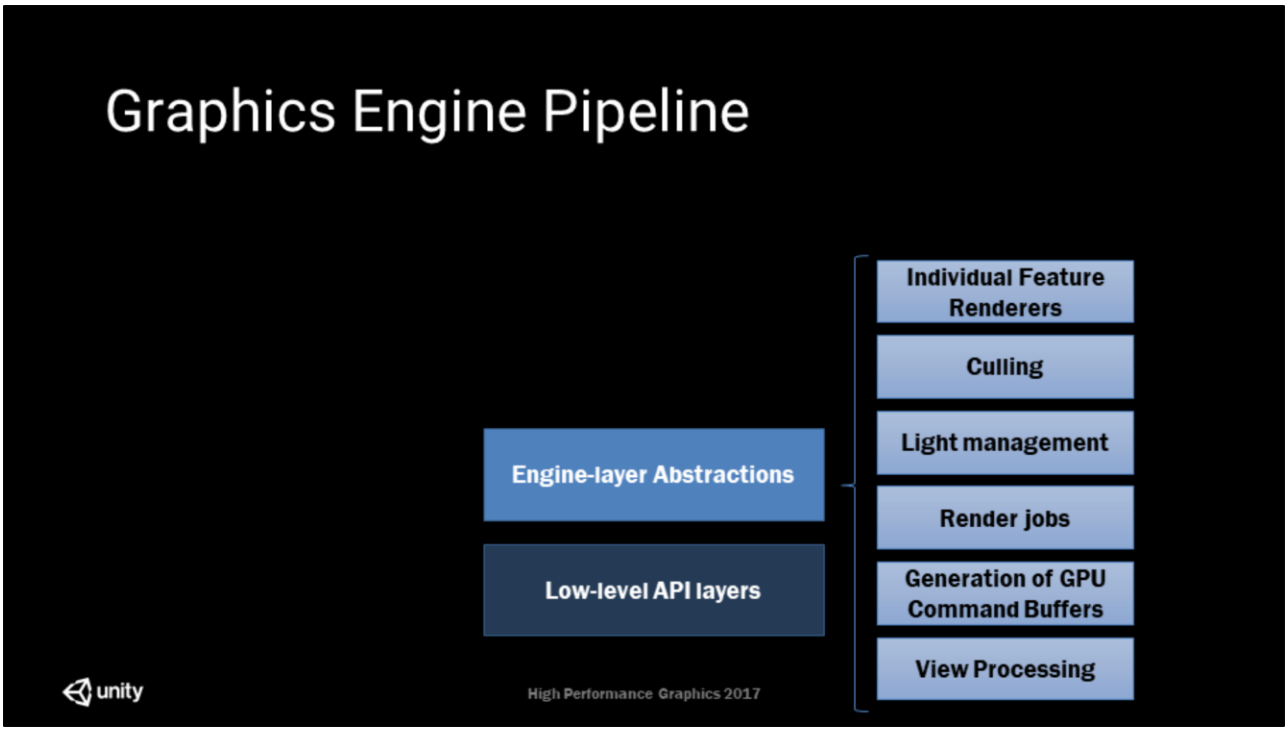
As a starting point, I'd like to spend some time talking about the state of the art in real-time graphics engine programming today.
This is largely based on my experience when I go and write application code.



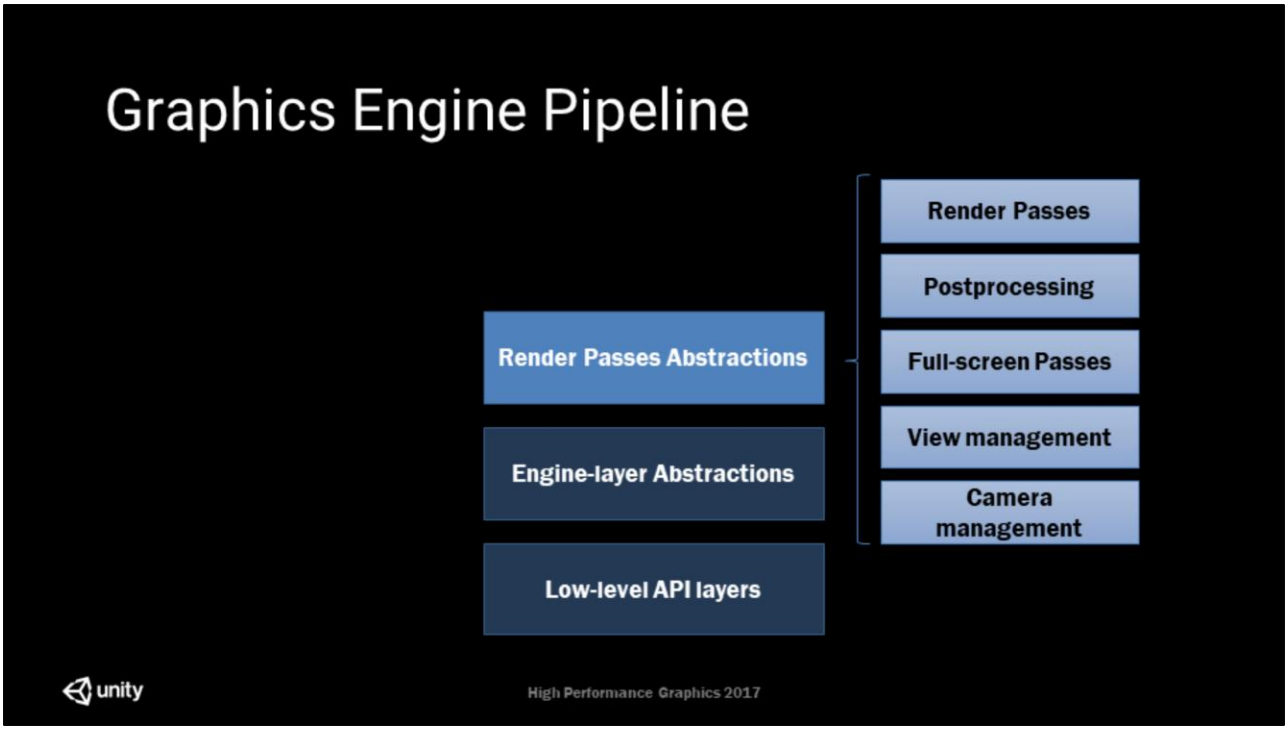
We typically start by writing low-level API abstractions: this is where device abstraction happens, platform-specific resource abstraction, render surfaces, render state management, shader management, etc. Low-level rendering layers. At this point, we work on the level that is frequently game-type or visual-style or algorithm-independent. Of course if you don't need to implement a particular abstraction (for example, if you didn't plan to use asynchronous compute) application developers may skip implementing this abstraction in that case.



Next we typically work on a layer for engine-level abstractions
Write higher level code on top of that
- Individual renderers
  - Skinned, static, instanced, etc.
- Culling (visibility, light list management, etc.)
- Jobification of render jobs and threading for generation of drawcalls and GPU command buffers
- This is also where we handle processing of views
- Phase management (CPU / GPU synchronization, or managing asynchronous compute scheduling)



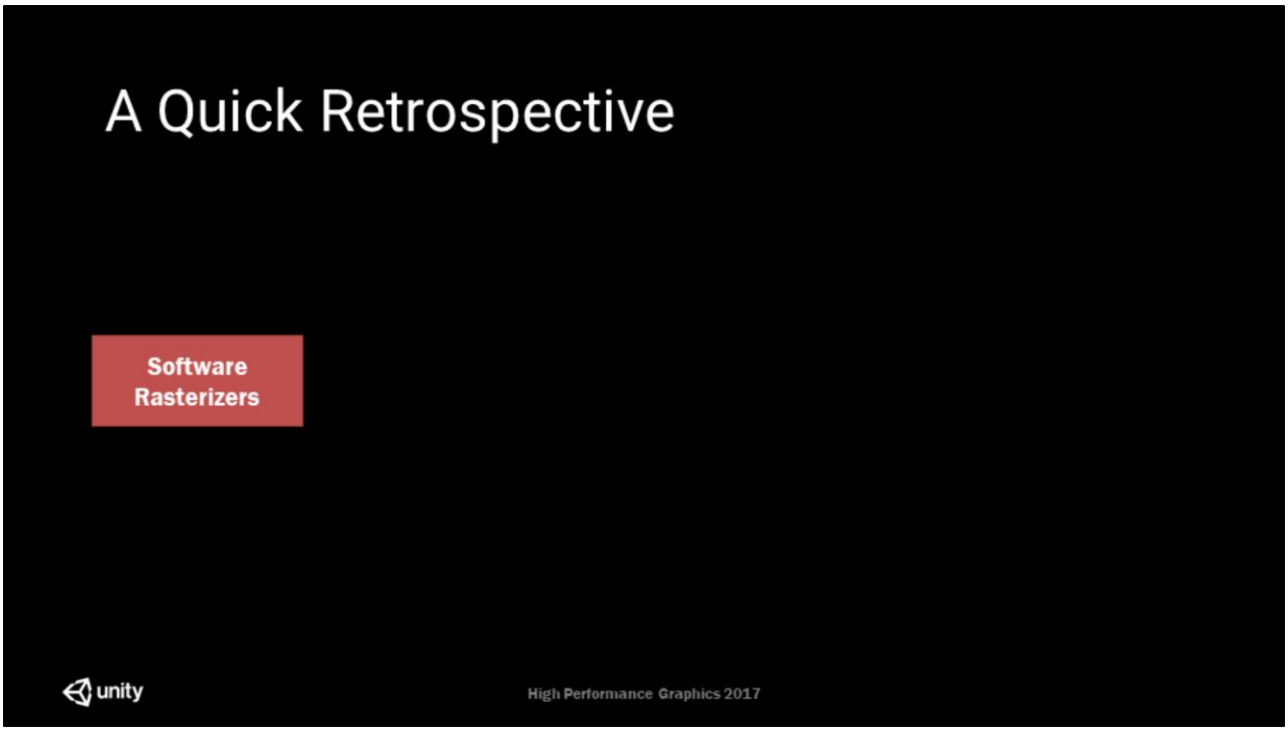
At this point, we are still fairly on a deep end. We haven't introduced any specific algorithms related to frame rendering, nothing terribly game specific (aside from forking the path whether we're doing things in 2D or 3D for optimization reasons). If we architect this layer correctly, we don't even need to think whether we're submitting single-frame or stereo.



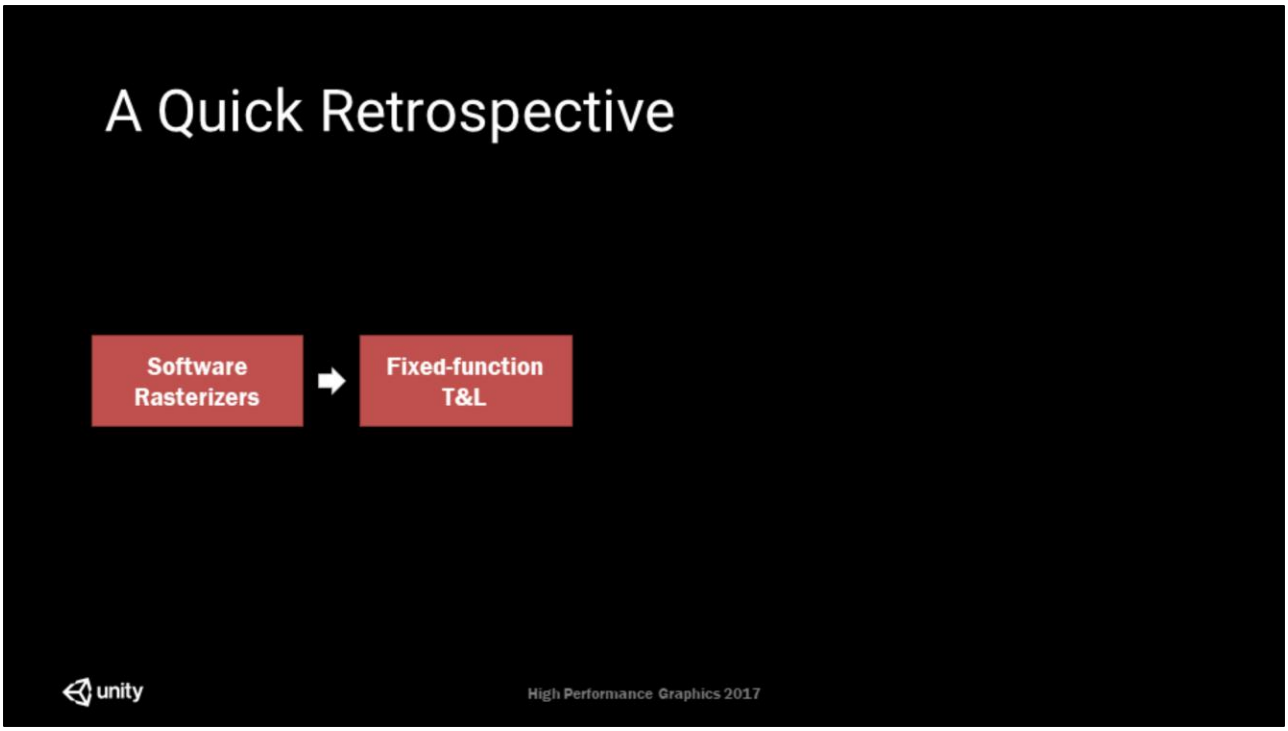
Next we write even higher level render loop control flow code. This is where we write our render passes, postprocessing, full-screen passes such as lighting or shadow application, shadow generation techniques, this is where we set up the various views, and do camera management. Many people thing of this step as the actual render pipeline element, especially when writing for a specific game or a platform (such as deferred pipeline, or a VR forward pipeline).

But this brings us to having to take a look – what example is the complexity of this step? What does it mean to write render pipeline passes?

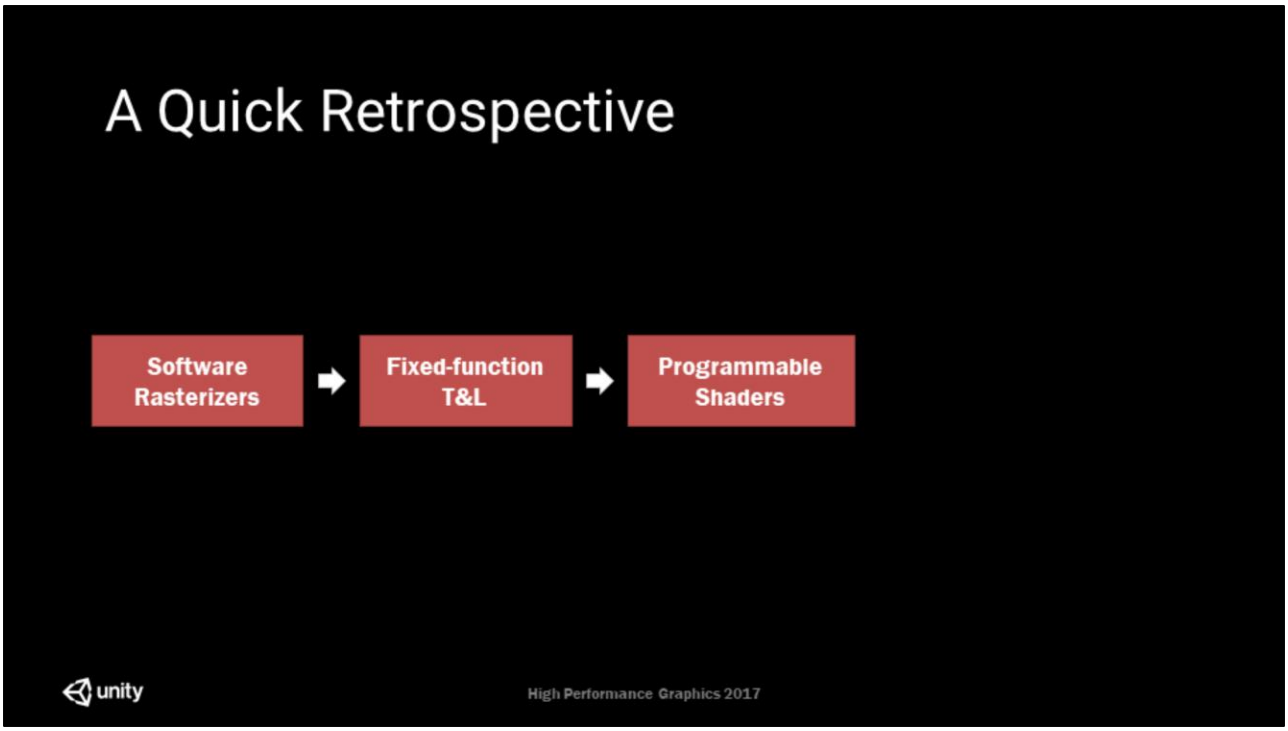
To do this, let’s do a quick retrospective into the evolution of real-time rendering engines.



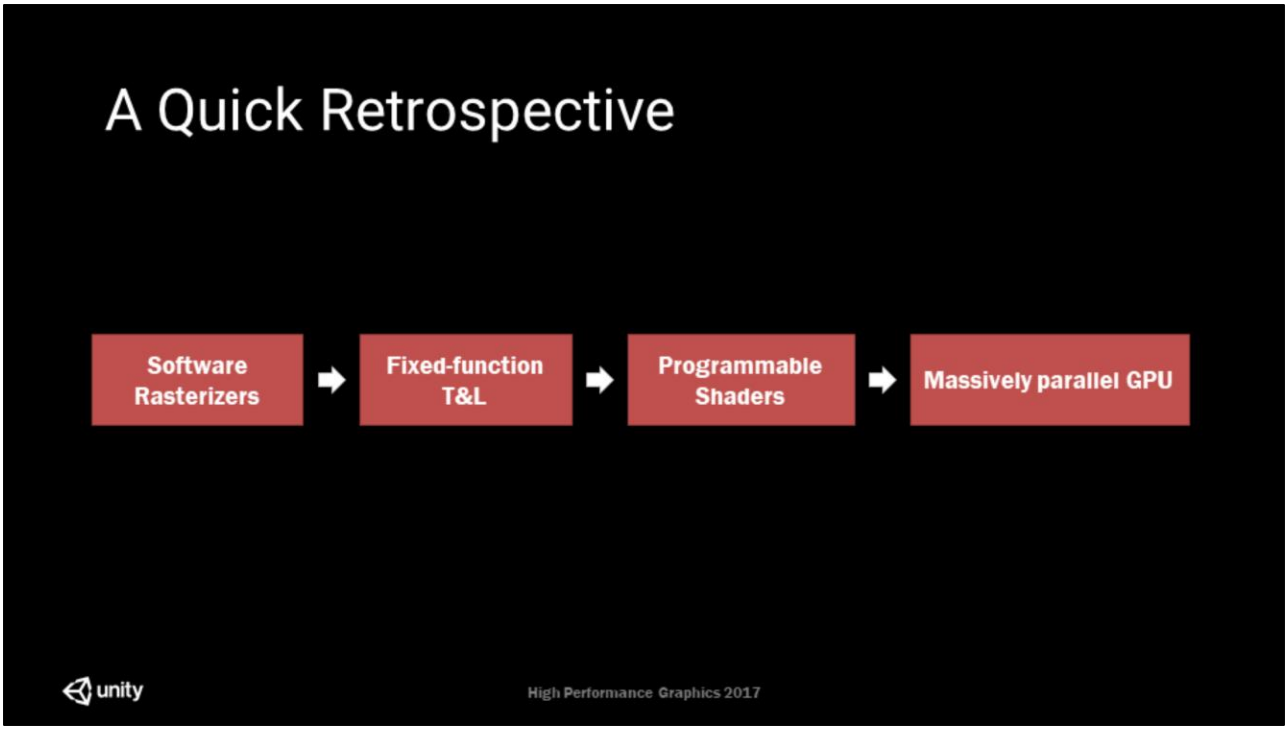
Of course graphics started a while back by writing software rasterizers. At this point, you had full flexibility but low performance. Everything executed on the CPU and you had full control over what you wanted to express. It was just slow.



Next we introduced one level of parallelization where we got fixed-function hardware to do texturing and lighting. You gave up some control (a very specific rendering method) but you got some performance back for it. But games really looked the same at that time and there was fairly little innovation in material models or lighting techniques during that time frame in real-time domain.



Then progressively we started gaining control back.. First by introducing programmable shaders – and we started evolving the visual fidelity of the resulting rendering…
Vectorized unified shader pipelines in hardware



And of course with the recent evolution of compute-based massively parallel GPUs with scalar execution and far better performance we're seeing huge amounts of innovation in material models, shadows and lighting techniques, image quality and many other aspects of real-time rendering. Visually, there is a lot of different looking styles in games and real-time applications now

But how does that relate to the render passes?

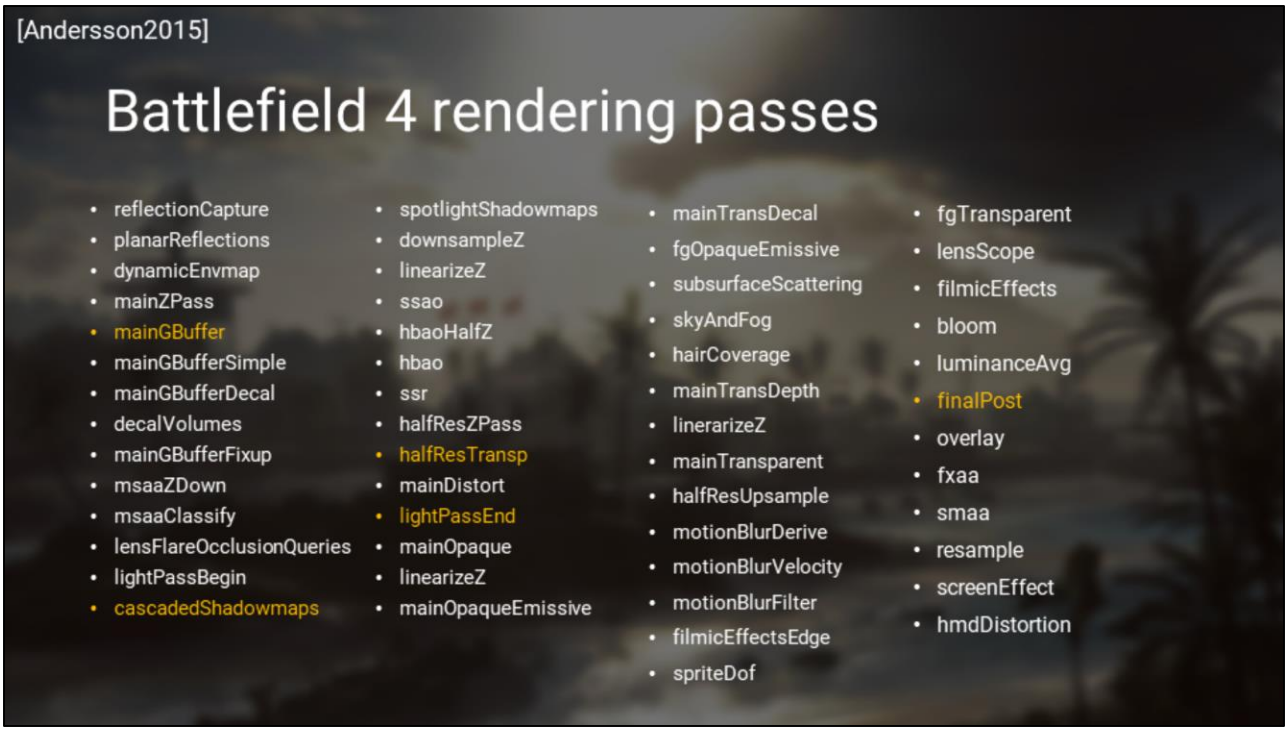
If we go back to the earlier stages, for example, when Halo: Combat Evolved came out on the original Xbox – the programming model for graphics back then did not allow a lot of flexibility in both the capabilities of the system nor in the degrees of freedom – there was still a lot of fixed-function elements in the graphics programmable model then.

Interestingly enough, Halo: Combat Evolved had a good resume of the variety of features (a terrain system, foliage, water, ice, snow, particles, etc.) but they were all much simpler. And the control flow of the overall game rendering was much less complex.

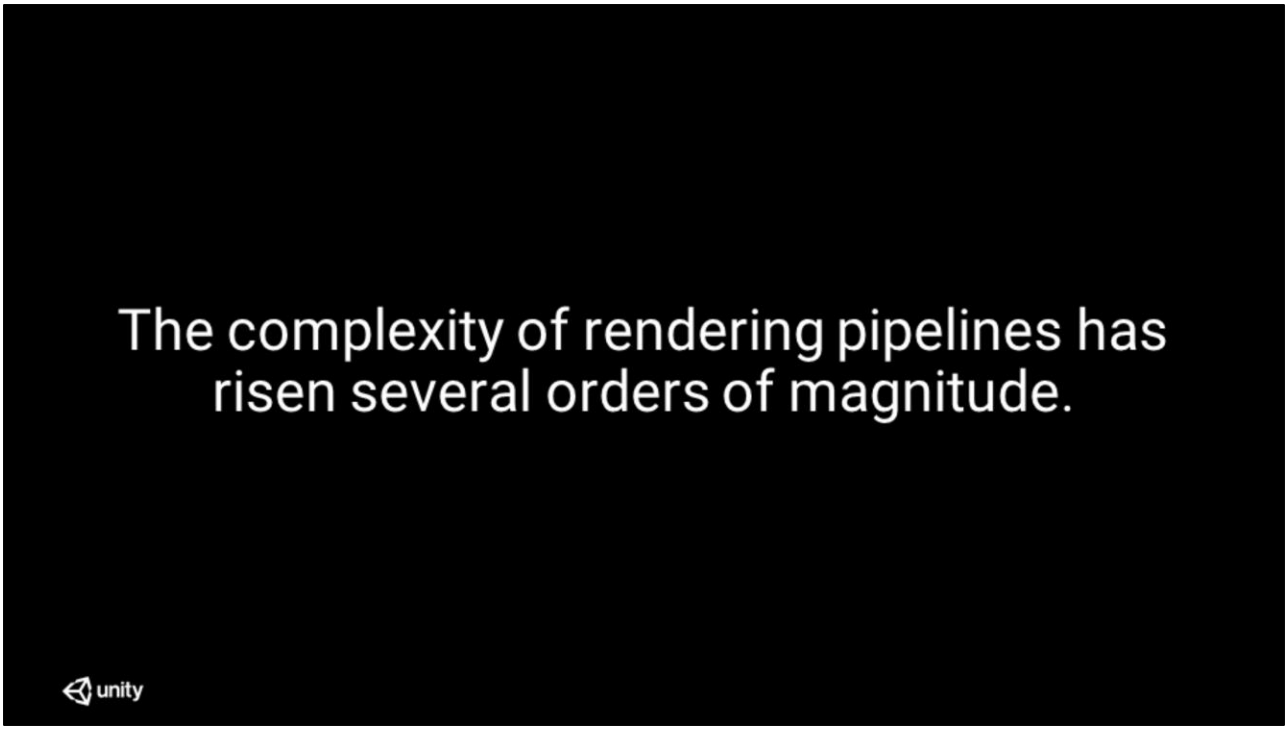
These early game features were fairly fixed-function. That made sense – it's a good route for small project – *don't overgeneralize without need*.
They were cheap to implement, were fast, and relatively easy to test. However, they were not expressive, and had limited functionality and art controllability.
But we've come a long way from there to now…

In the current world of player expectations, for AAA blockbuster games (just like movies), the expectations have risen drastically.
Technology has been progressing by leaps in the last decade or so and consumers expect modern games to showcase their capabilities accordingly. Modern game rendering pipelines are now very complex. There is a huge variety of render passes for any given frame. Furthermore, the render passes have
- Frame-dependent structure
- They are Game-type dependent
- They are Content-dependent
- And they can be Platform-dependent

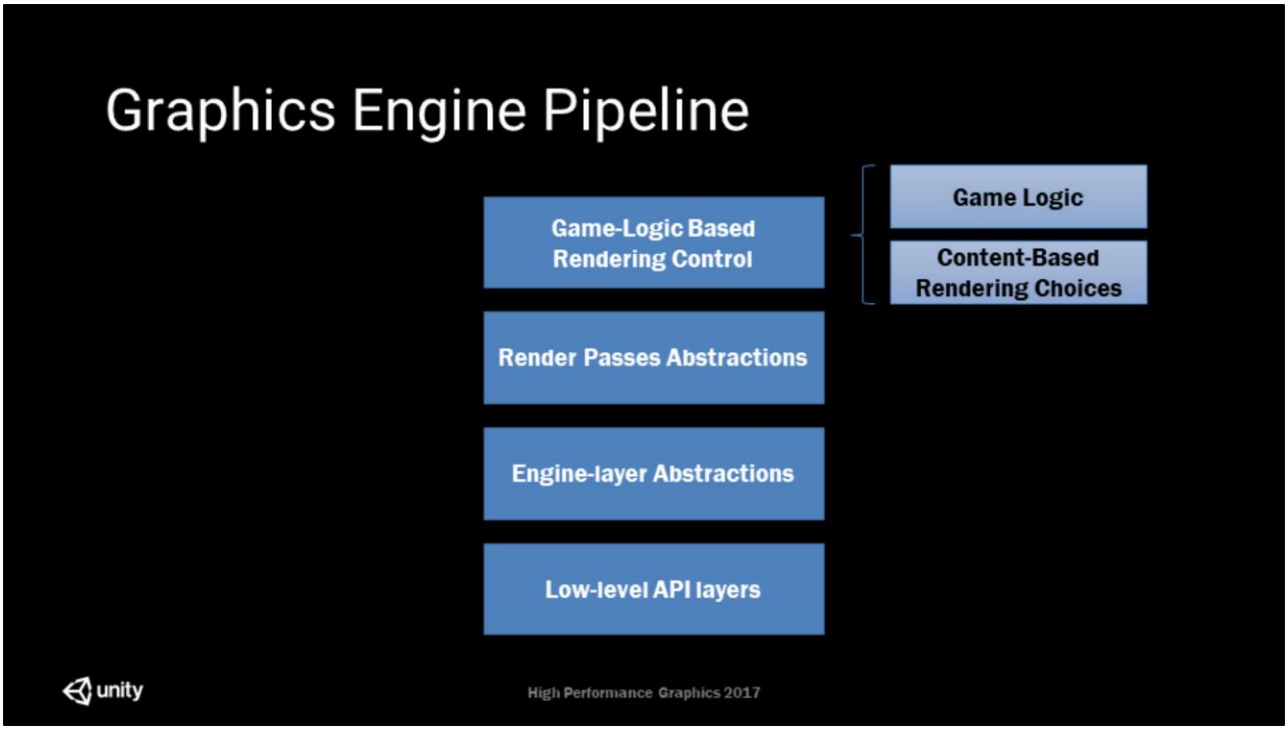
Here is an example from Destiny (from the GDC 2015 presentation) - If we look at breakdown of a frame, you'll quickly see that we build the frame from <a number of passes> (as you see in this list on the right). Note that these passes are different depending on the state of the game and the content visible.



This is the rendering passes DICE had in Frostbite a few years ago for Battlefield 4. Similar or more complexity is found in Destiny, and in most modern games.

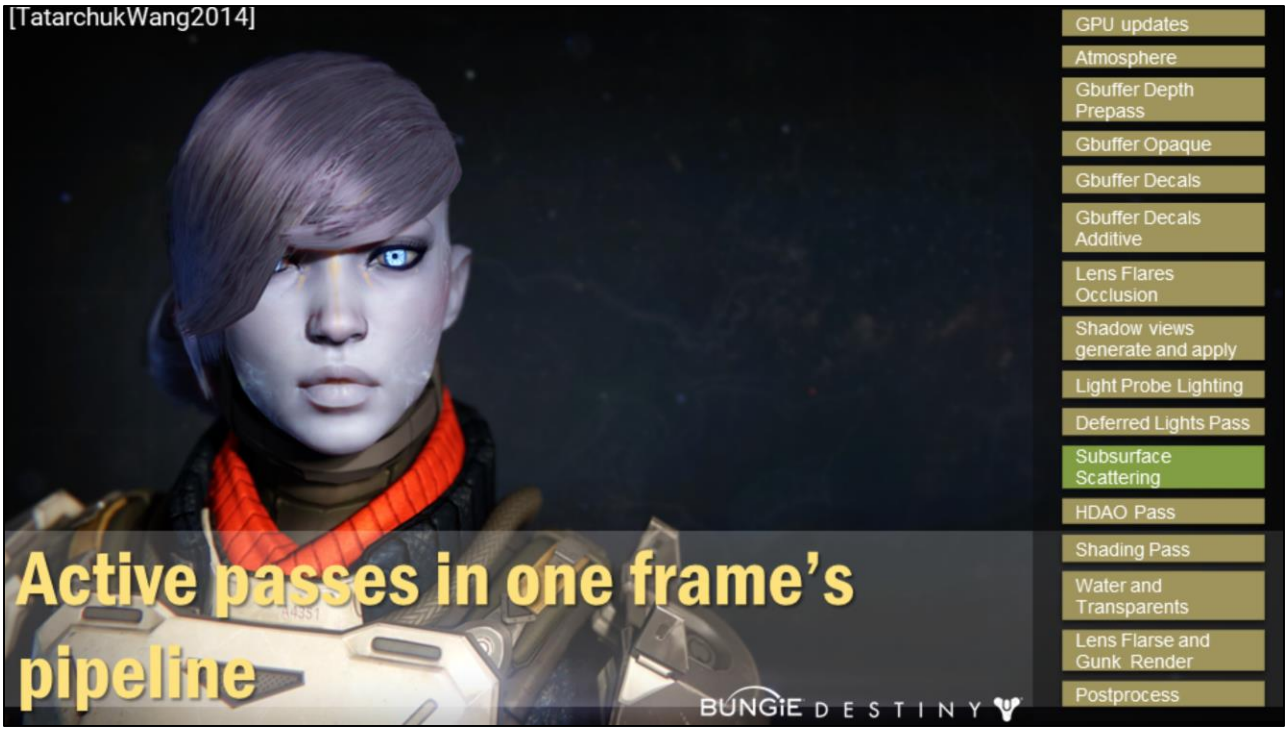


So as you can see the amount of render passes and the feature complexity has come a long way since the early days of fixed-function hardware. The complexity of rendering pipelines has risen several orders of magnitude

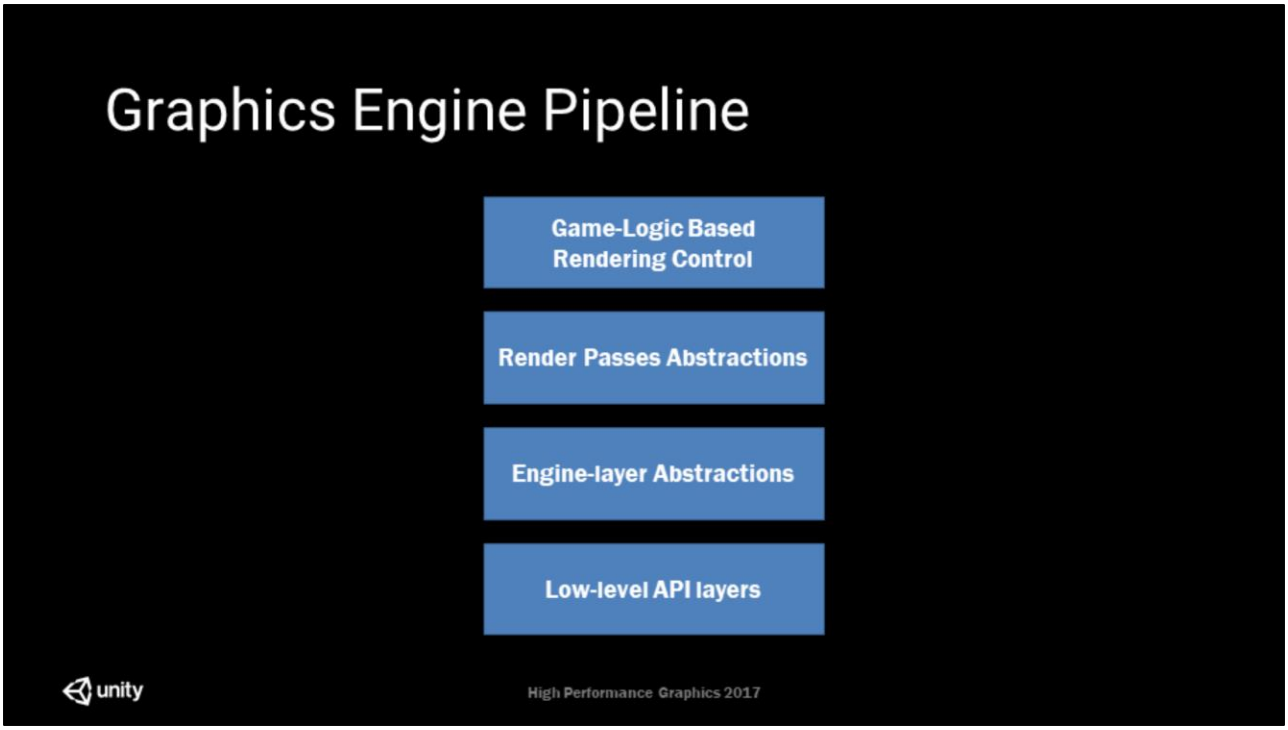


Coming back to our graphics engine pipeline, with this understanding in mind, at the top layer we start looking at choices that the game type and game itself is driving. These are rendering choices that change because of a particular content element that's visible or a particular gameplay behavior (particle effects, postprocessing, etc.

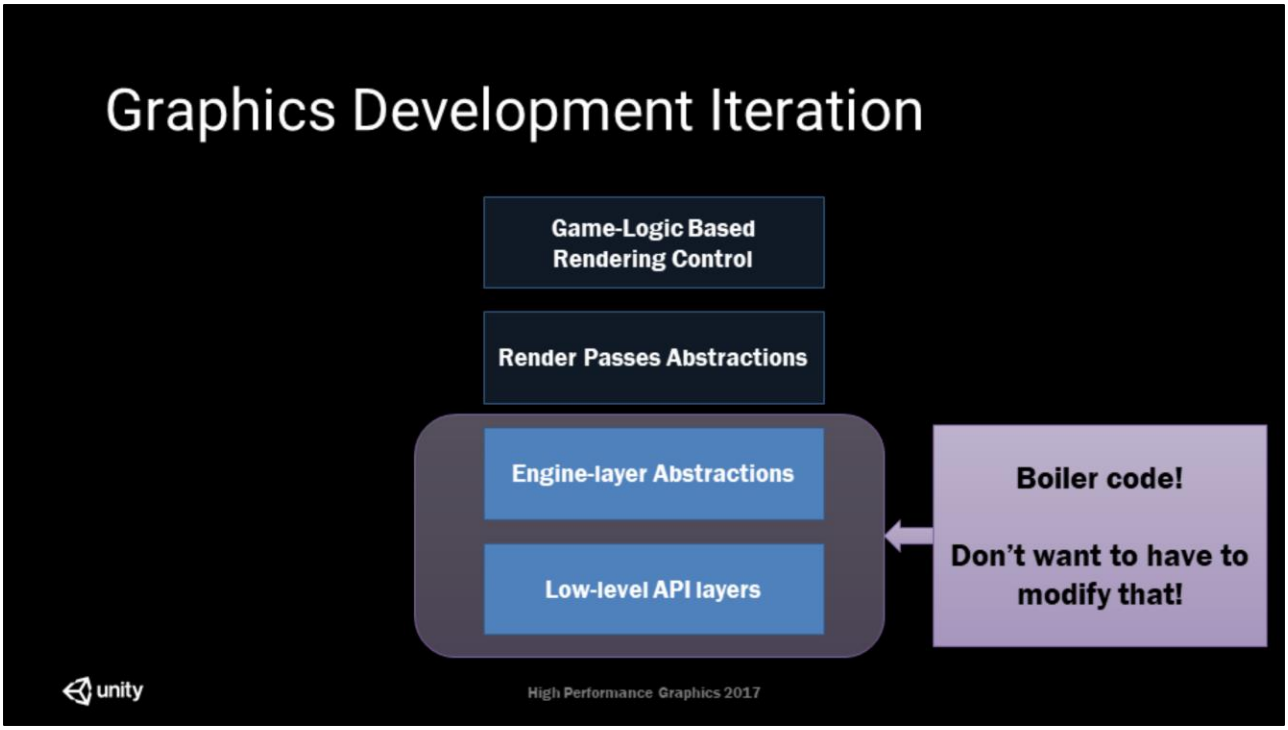
The structure of the frame can be very content-dependent – for example, if in this frame we don't have any translucent or skin materials, the subsurface scattering passes will be sikipped. This may also be true for transparents if there are none.



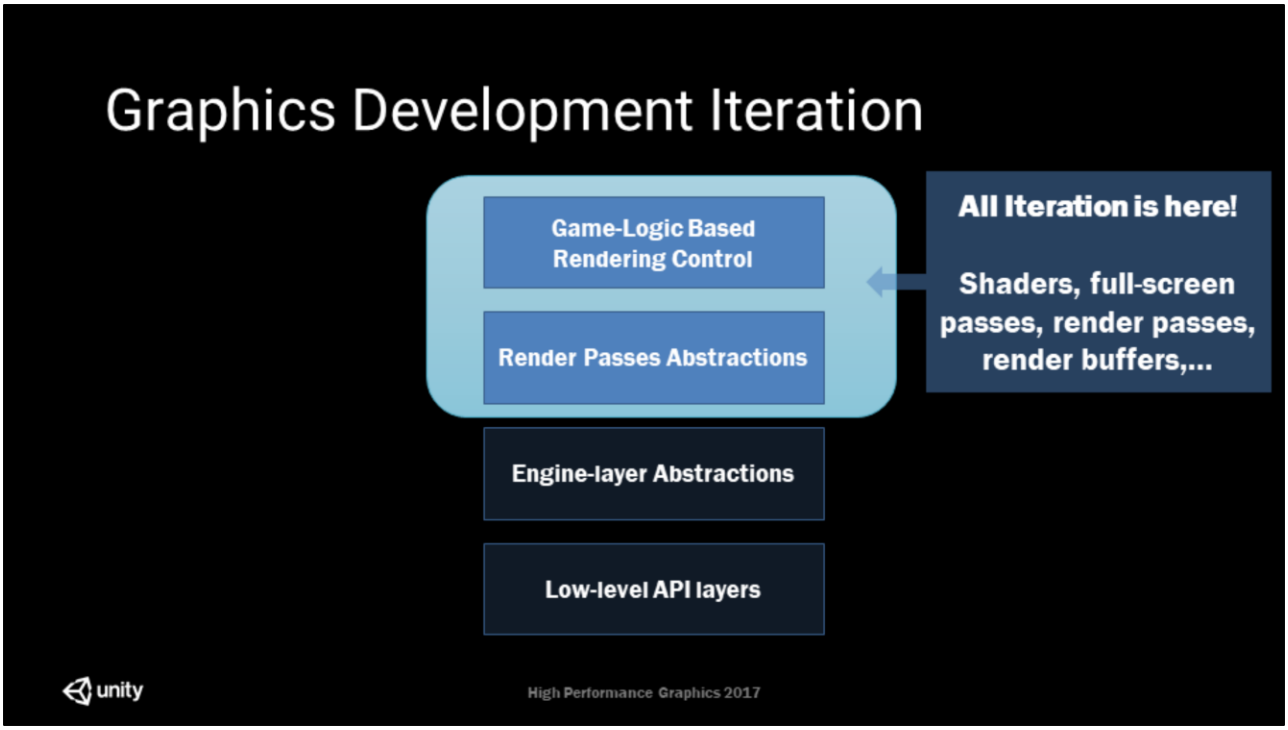
Where is in a different frame, when these elements are present, like this face in an in-game cinematic for the investment screen, we will see the appropriate subsurface scattering passes enabled



So now that we have the big picture of the different pieces of the graphics engines architecture, what does it mean to iterate on them while developing a particular graphics algorithm? For example, what if we want to explore a new physically-based material model representation which requires some changes to the material shaders and G-buffer layout plus lighting?

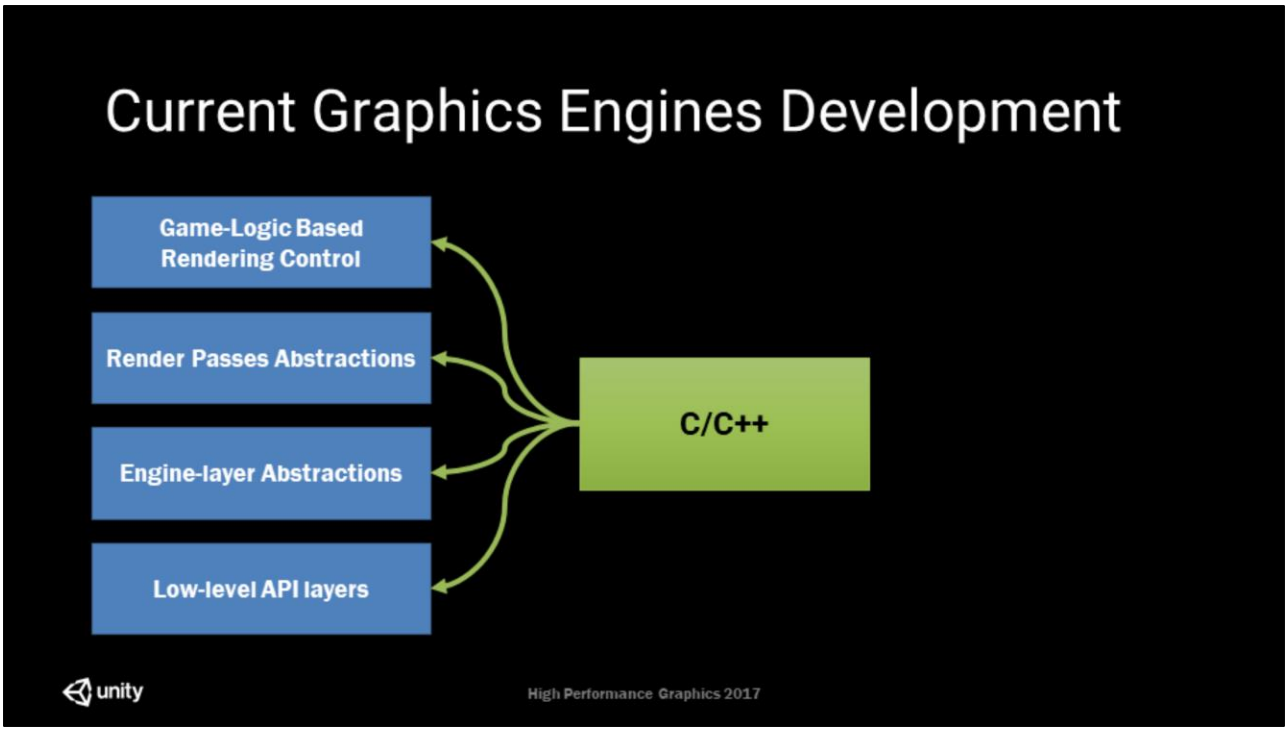


What we don't want to do when we're iterating on a new G-buffer layout is having to muck w/ the platform layers, resource loading, etc. And that's what most engineers outside of a game / graphics engine framework have to do. Often it's cut-n-paste, but it's still layers of boring boiler plate code. No thank you.

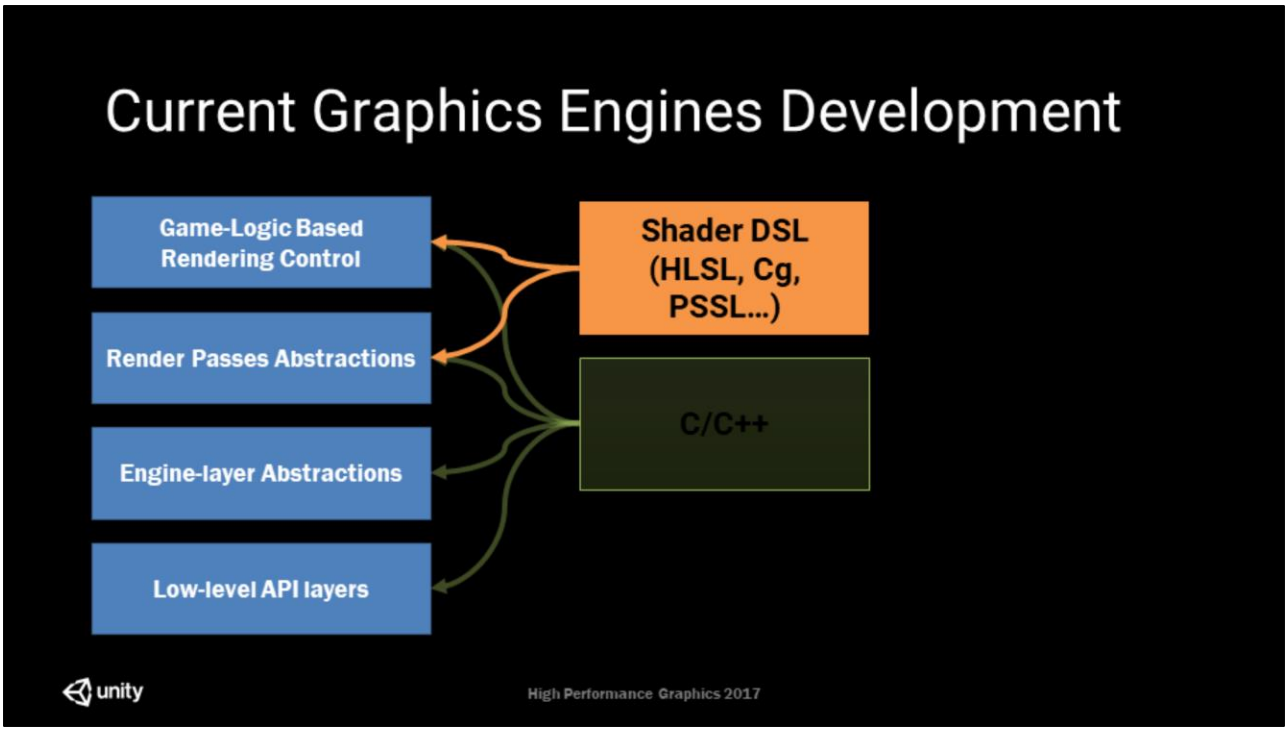


What we really want to be focused on is iterating on the shader passes, render textures layout, etc. The heart and soul of the algorithm – and that's where we want to spend our time.

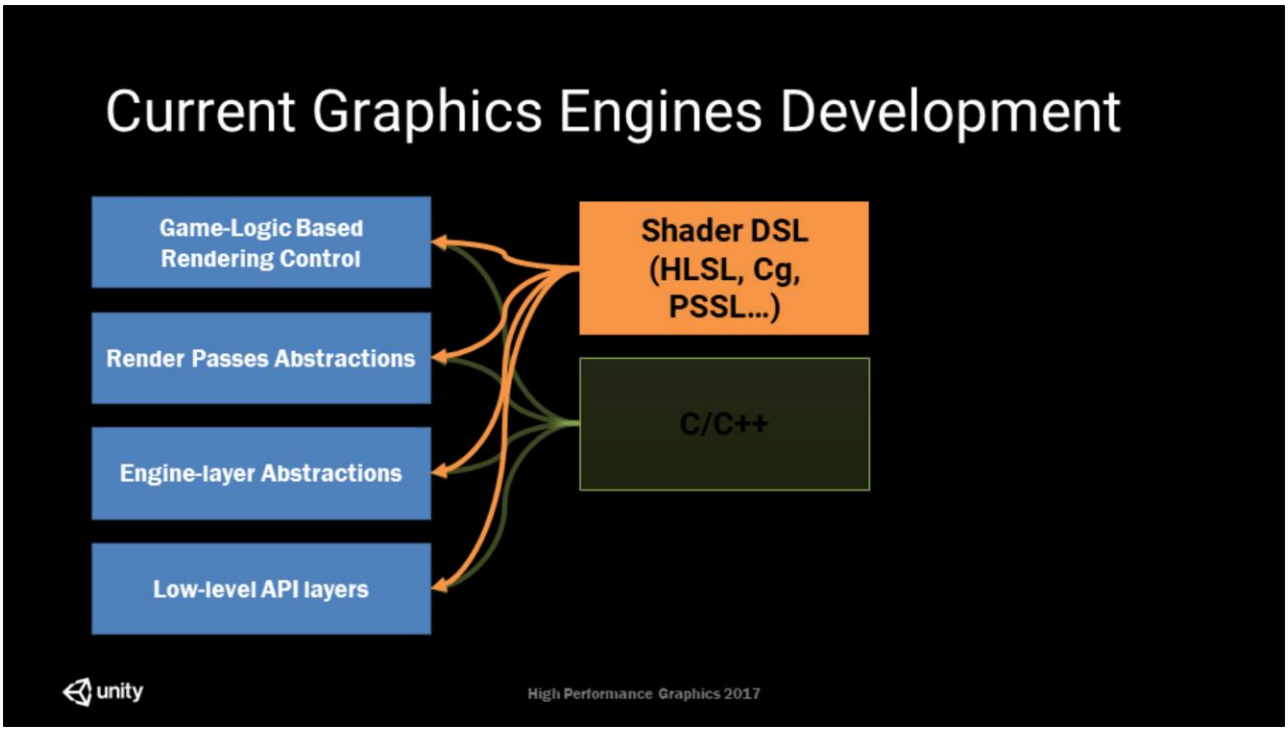
Let's take a look at what it means to program these individual layers in game graphics engines now.



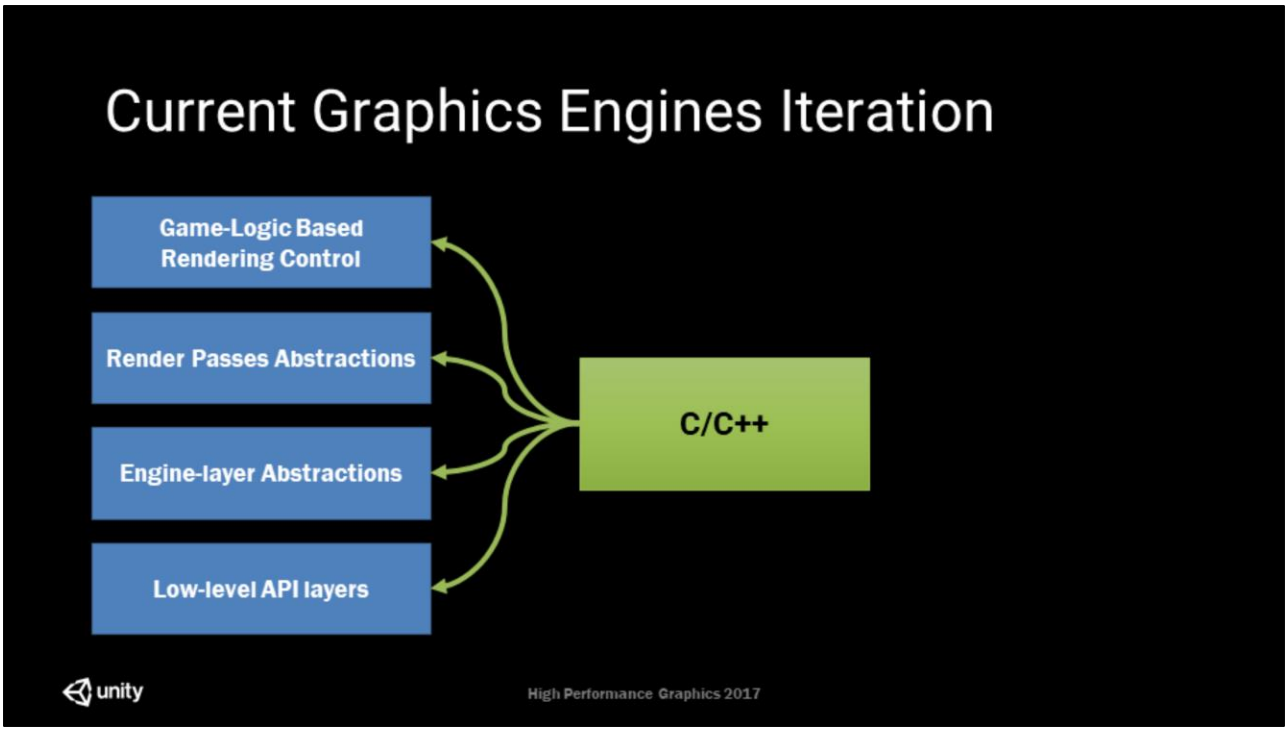
Mostly, when we're writing the game engine layers, we're working in c++ (let's pretend we forgot about PS3 and SPU Assembly…)



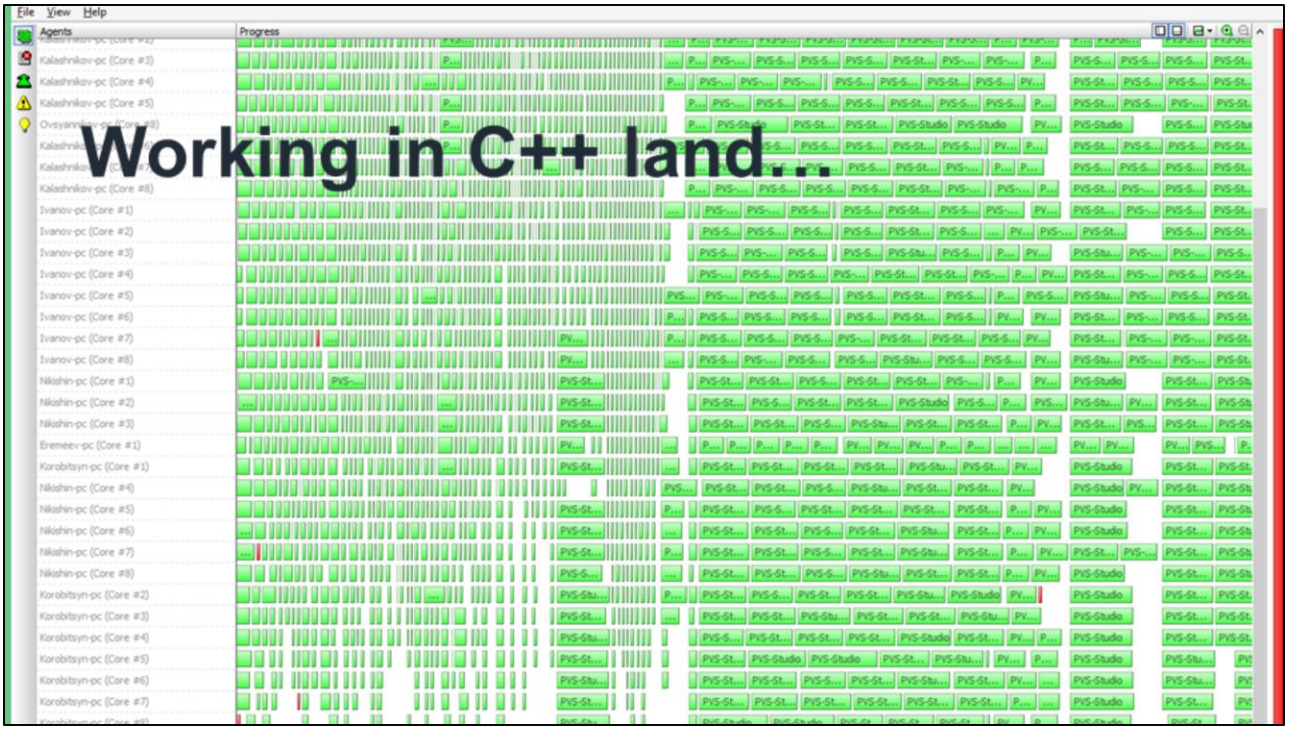
We also do some work using shader domain-specific languages (whether it be a platform-specific shader language like HLSL, Cg, PSSL, or, like we did at Bungie, a custom shader domain-specific language, like TFX). Typically that layer touches mostly the higher elements of this stack.



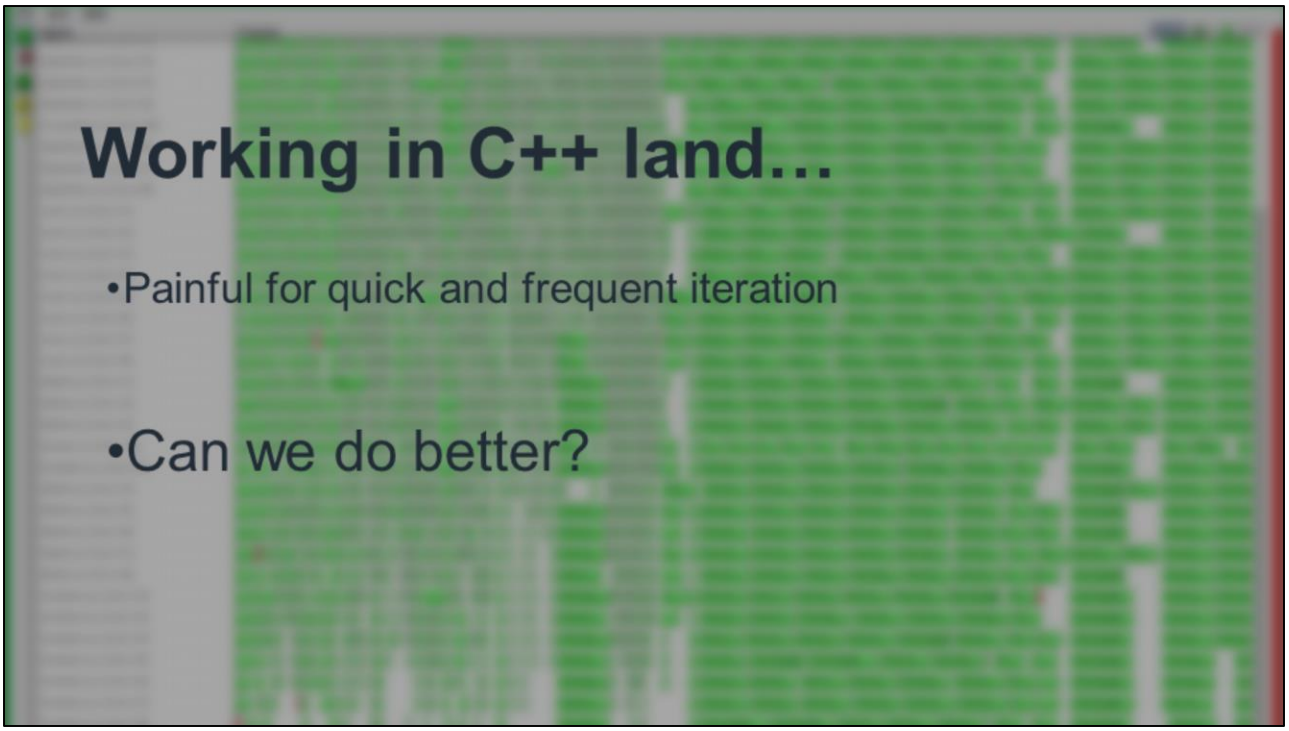
It's far more rare that dependency on the shader implementation penetrates into lower-levels of the stack like the engine layers or low-level API layers. Typically this occurs when the shader functionality is tied to the hardware-specific feature – like tessellation, instancing or asynchronous compute. In those cases, the engines lower-levels require deeper changes, in line with the actual shader changes. But those tend to be more edge situations for graphics algorithm development.



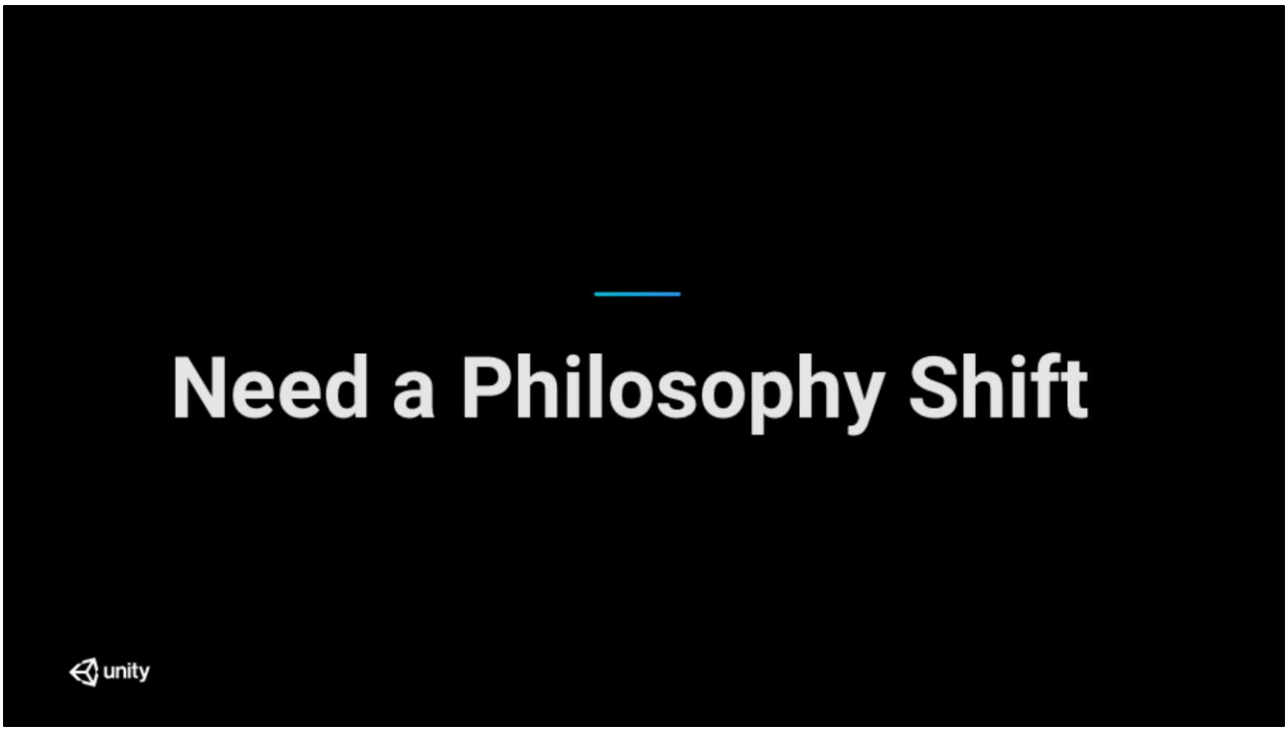
So what does C++ iteration look like?



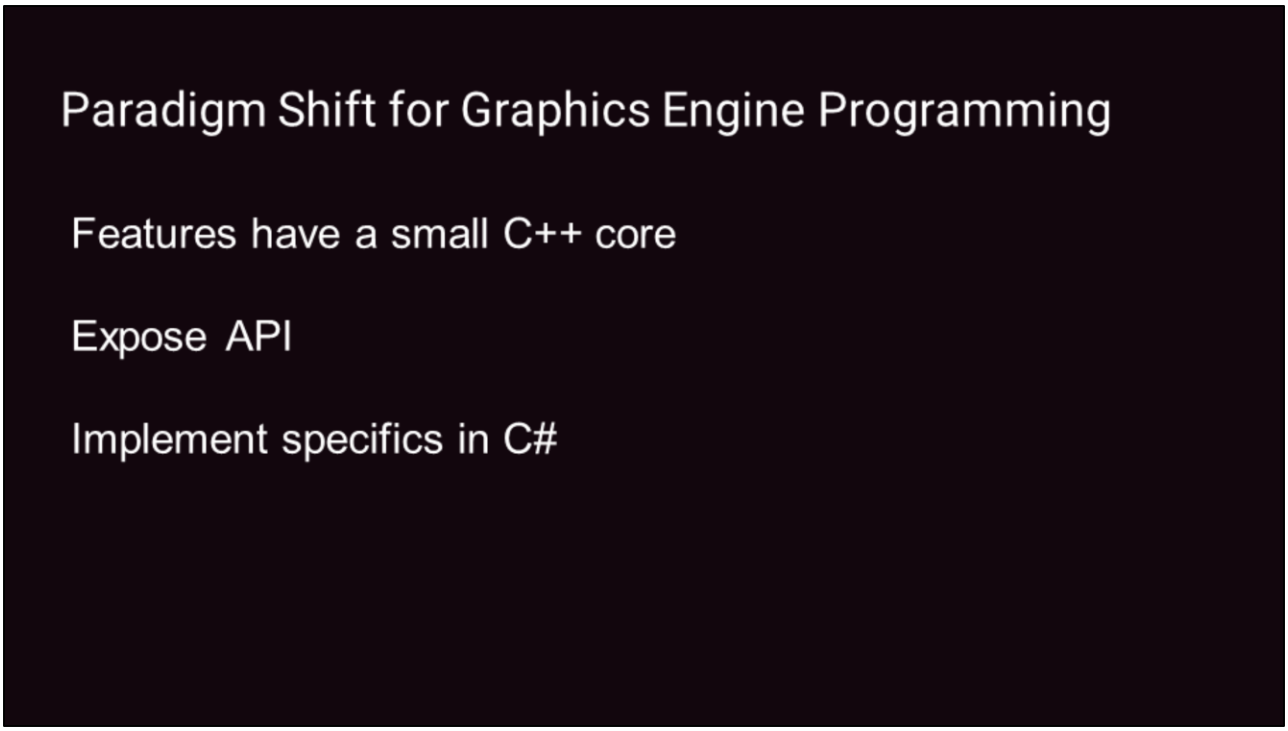
We know it well.. C++/build+link and continue. As engines increase in complexity, this means slower iteration time (this is just some random image of Incredibuilt from the interwebs, but those of us who worked on large game / graphics engines know this well). And even with incremental builds, we're still slaves to slow link times.



As all of us know, this is a fairly painful process. Most of us just take it for granted that this is the way it’s got to be for graphics development. But can we do better?



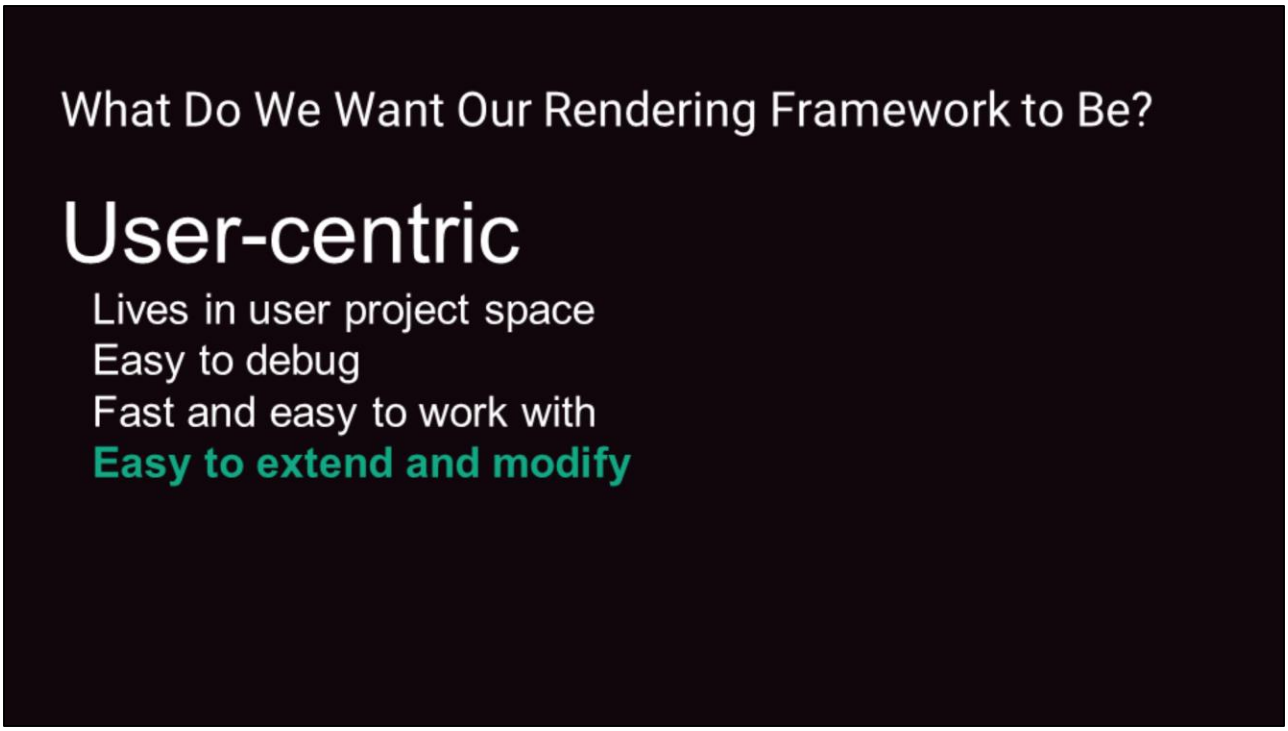
Which brings us to the need for a philosophical approach to programming models for rendering.



To address this we made a shift conceptually. What if we start thinking of the rendering layers as a small core of C++ functionality, move move into the scripting C# land, and expose the low-level API control

This would give flexibility of writing in script which can be modified, debugged, improved all without having to do an engine recompilation. SIGN ME UP!

We want this rendering framework to be lean – we want less overall C++ code to maintain. We also don't want to have to configure on the low level for each platform and algorithm (like deferred on consoles, forward for VR, etc.) – this logic should be moved into the configurable script layer.



So of course Unity's credo is to democratize game development. Well, we want to democratize Rendering!

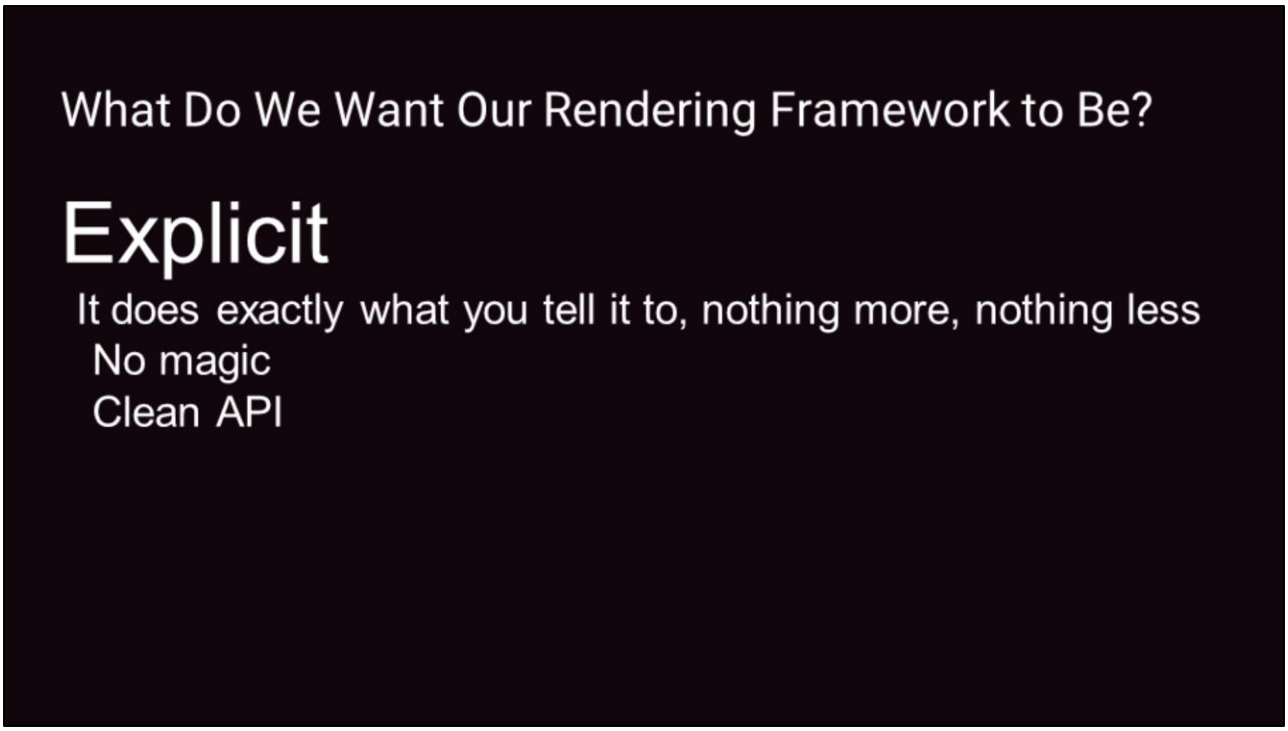
Our framework should be user (you, the algorithm developer) centric. The code changes should be accessible and modifiable by you, easy to debug, have fast iteration and easy to change. Really want to make people working with rendering algorithms more productive!

And we want to make sure that the framework is easily customizable for new algorithms and rendering types.

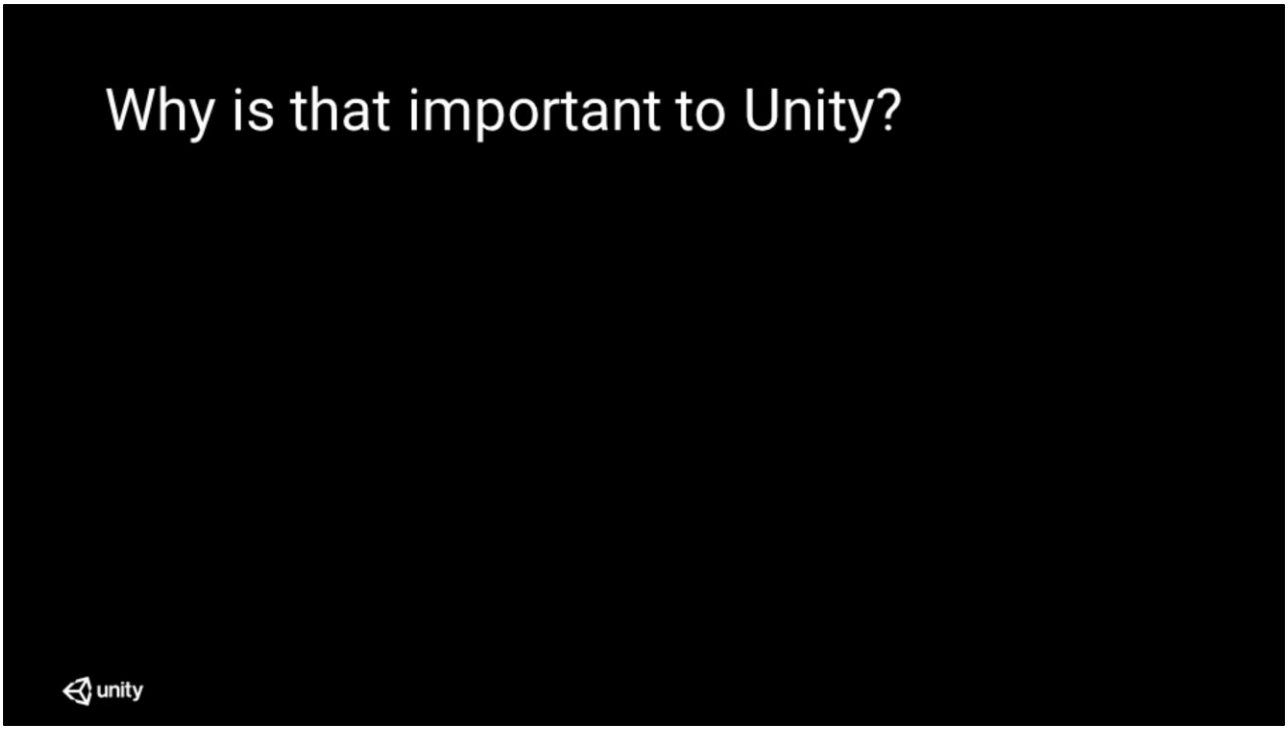
The goal for our new programmable model is that it needs to be optimal. Should not do work it does not need to do. Which means that we no longer want to even code layers of code for the particular rendering pipeline that it doesn't need to execute. Are you doing a 2D pipeline? Well, you should be able to program just that, and not have to *skip* all the layers of code that have been designed for 3D VR

And our framework needs to run in a heavily parallelized way, to make sure that it's performant.

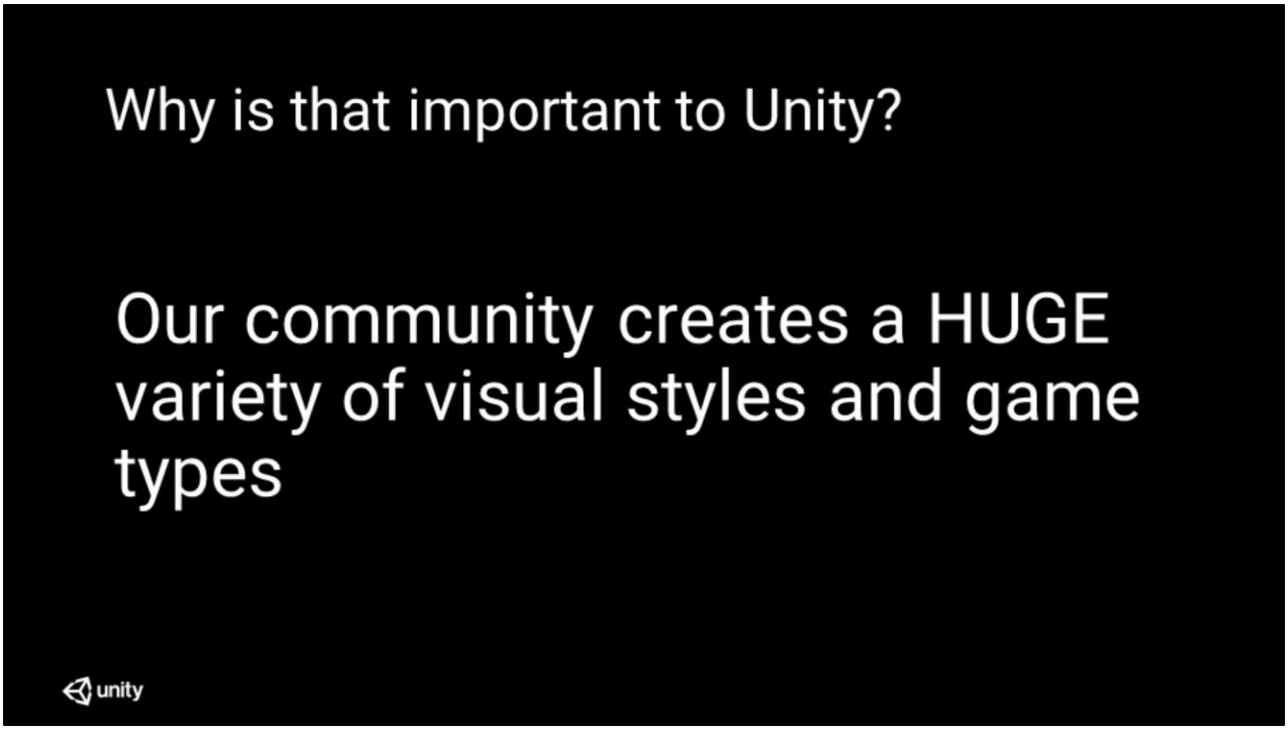
Supports jobs, and jobified rendering



The framework should have clean and easy to use API



Why is this in particular so hugely important to Unity?



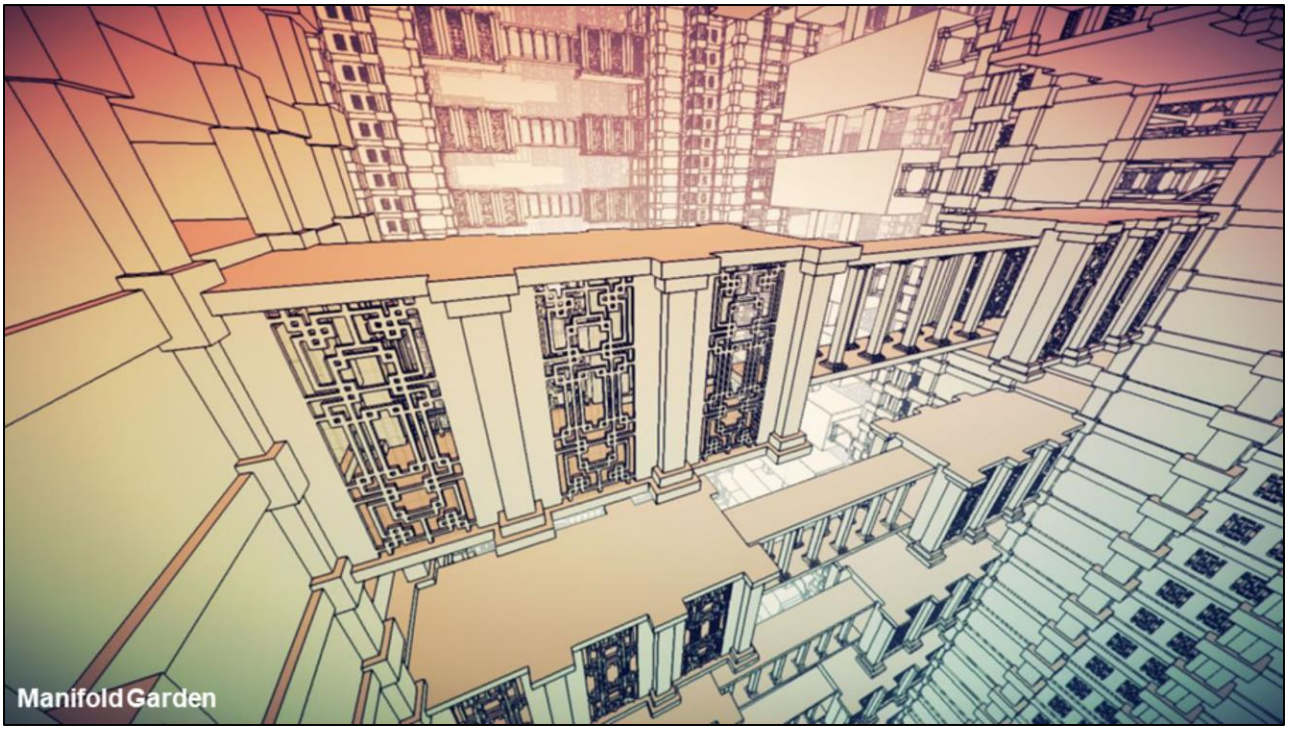
Unity community is creating a vast variety of visual styles and game types. We are the dominant engine on all mobile platforms, VR, and heavily present on consoles and PC and developers are creating a crazy variety of visuals with Unity, ranging from pure 2D rendering, to VR / AR, and high-end graphics. Let me show you some of the examples that the talented game developers have done with Unity.

So you can see some overhead rendering for Cities: Skylines

An extremely fast low-latency VR rendering like the job simulator game

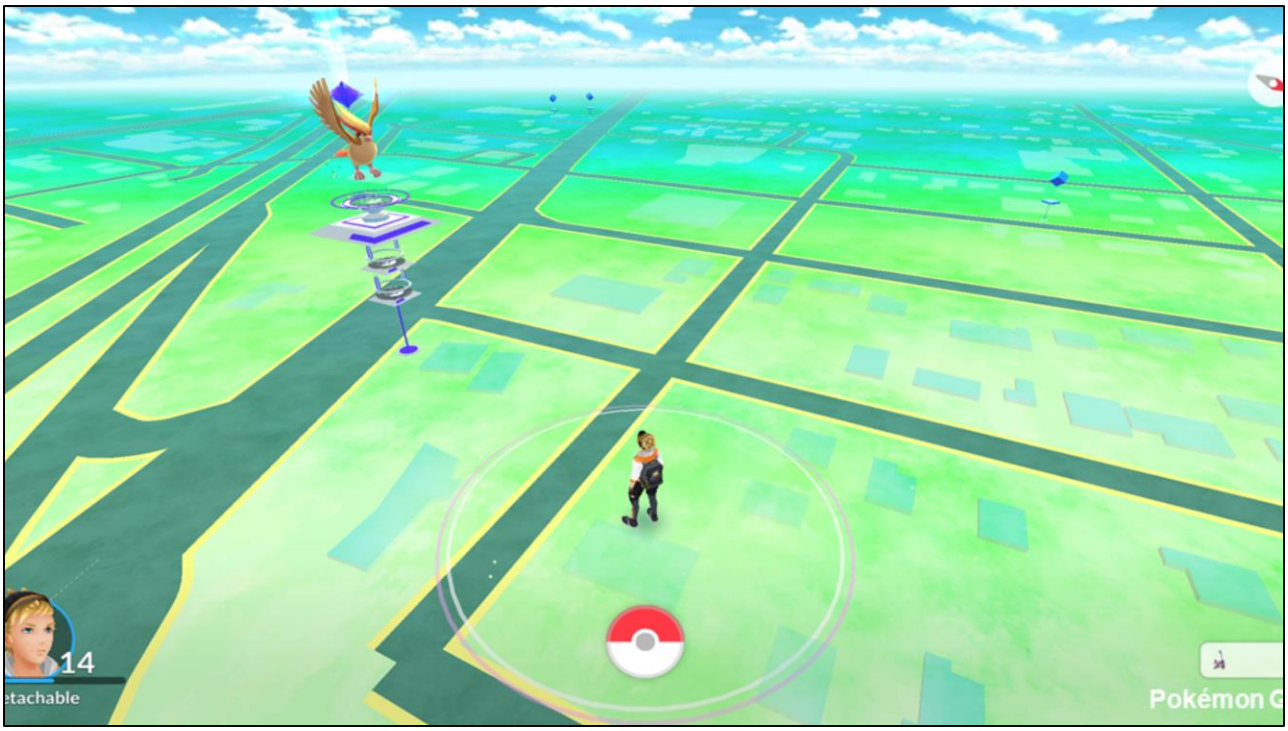
Or VR Eagle Flight game

Some 2 rendering styles

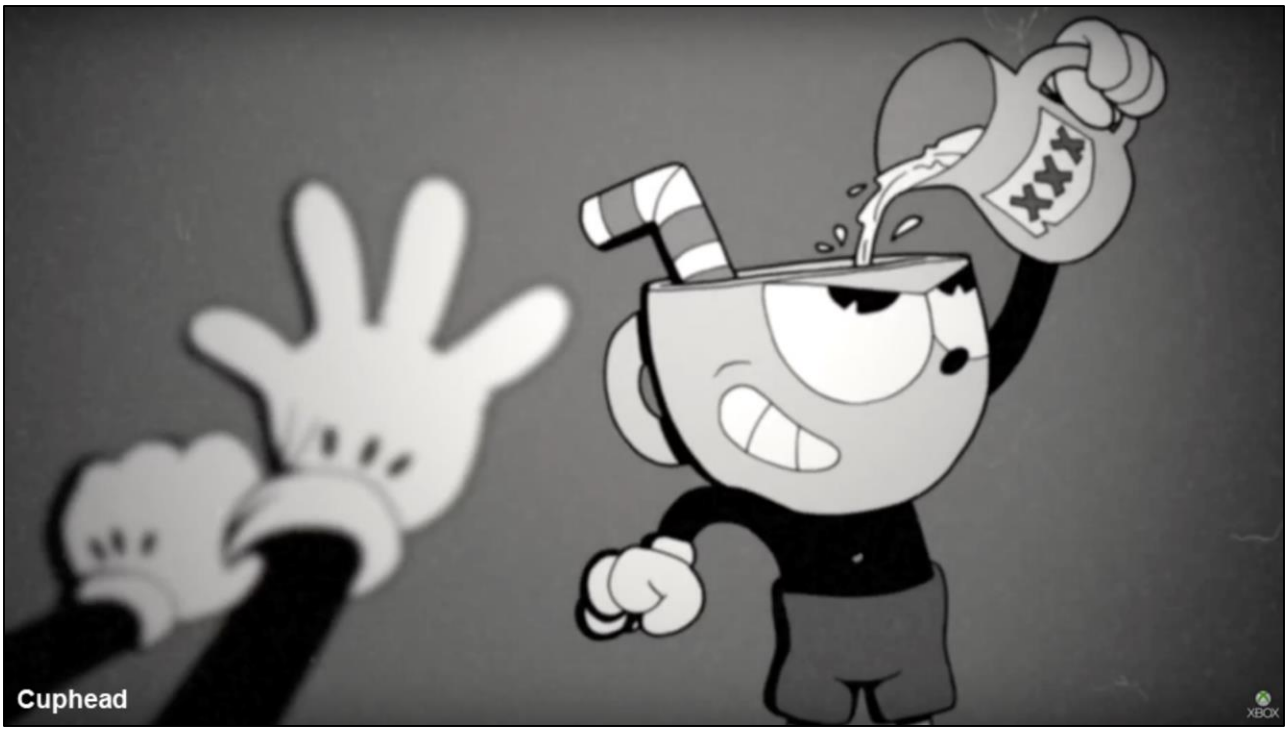


Heavily stylized Manifold garden game

A beautifully different stylized world of Inside



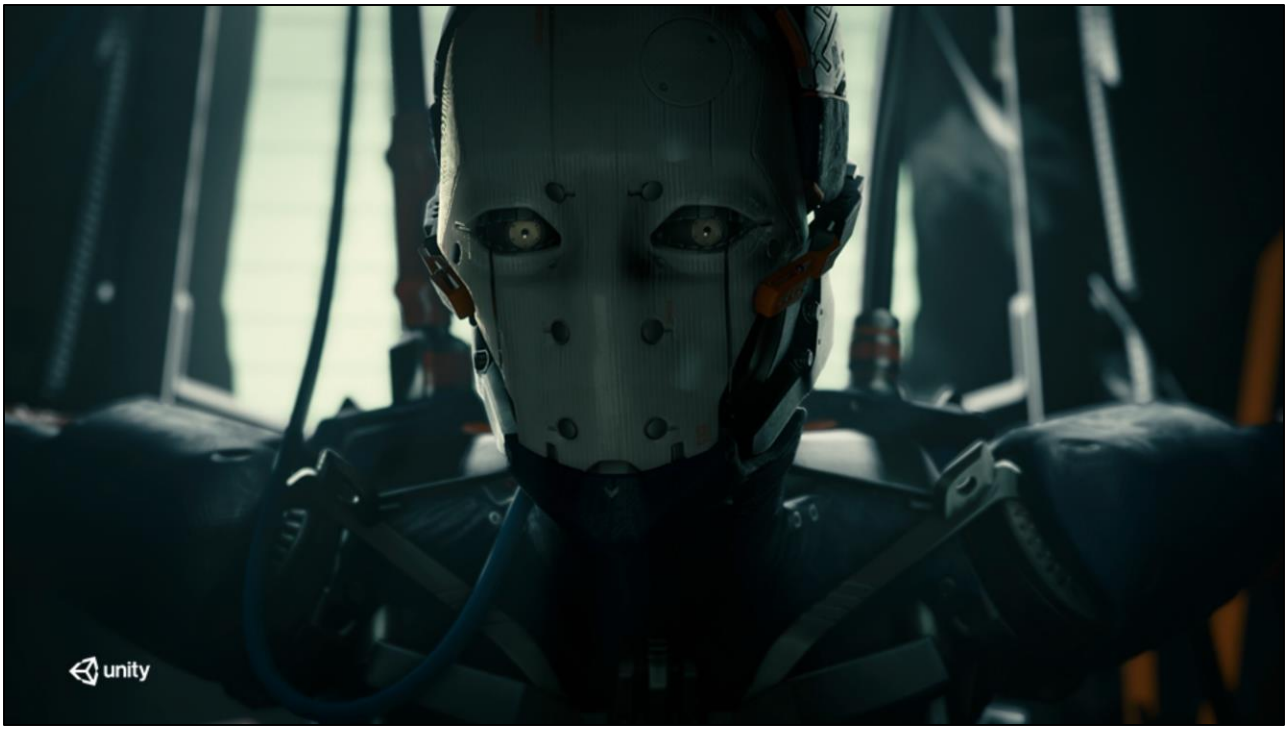
Augmented reality of Pokemon Go



A throwback to actual cartoons of Cuphead

An unspeakable beauty of the Ori series and the upcoming Ori and the Will of the Wisps

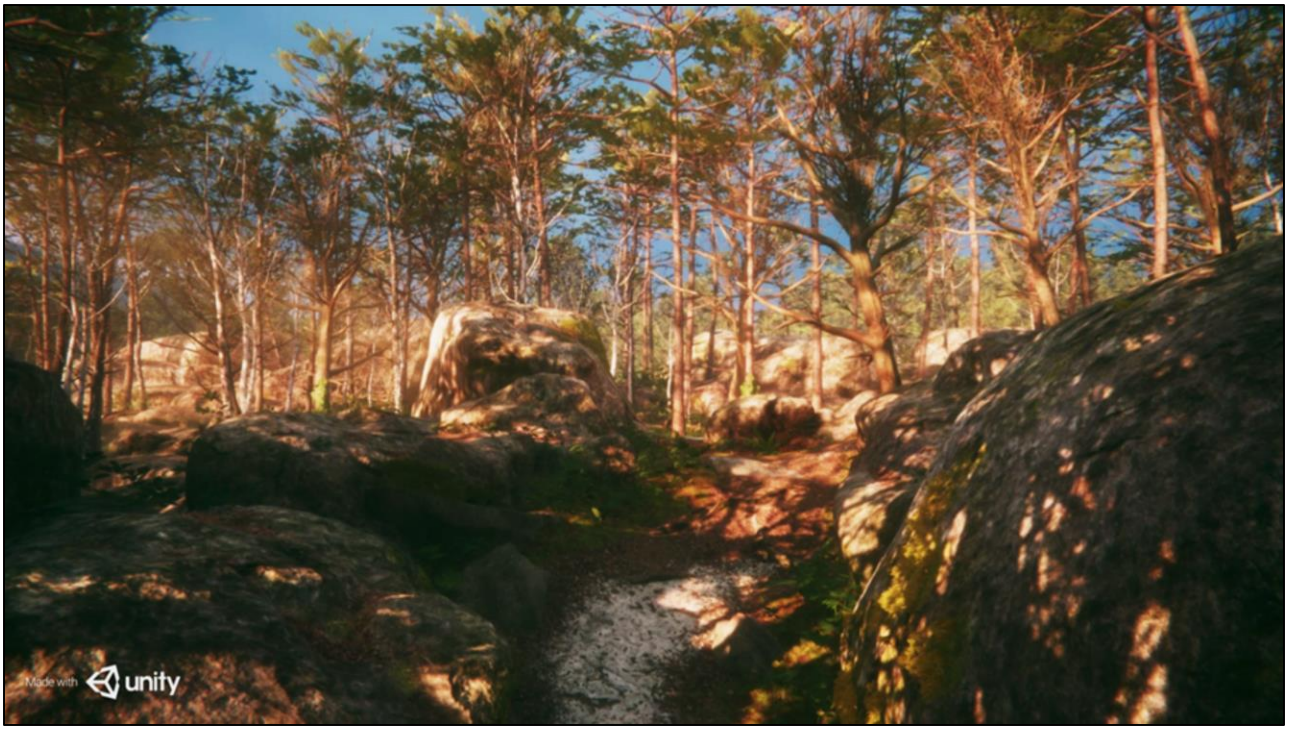
Amazing high-end visuals of the Adam demo from our own Unity demo team (https://unity3d.com/pages/adam)


unity

Example of our High quality rendering pipeline using scriptable render loops for the about to be released Photogrammetry demo from Unity

the photogrammetry solution page: https://unity3d.com/solutions/photogrammetry
Link of the photogrammetry video: https://www.youtube.com/watch?v=hCeP_XUIB5U
Siggraph 2017 talk about Photogrammetry workflow and the tech behind the de-lighting tool:
https://www.youtube.com/watch?v=Ny9ZXt_2v2Y&list=PLX2vGYjWbI0SawzXeOMl6F9cBcGUkvxY8


Made with unity


Made with unity

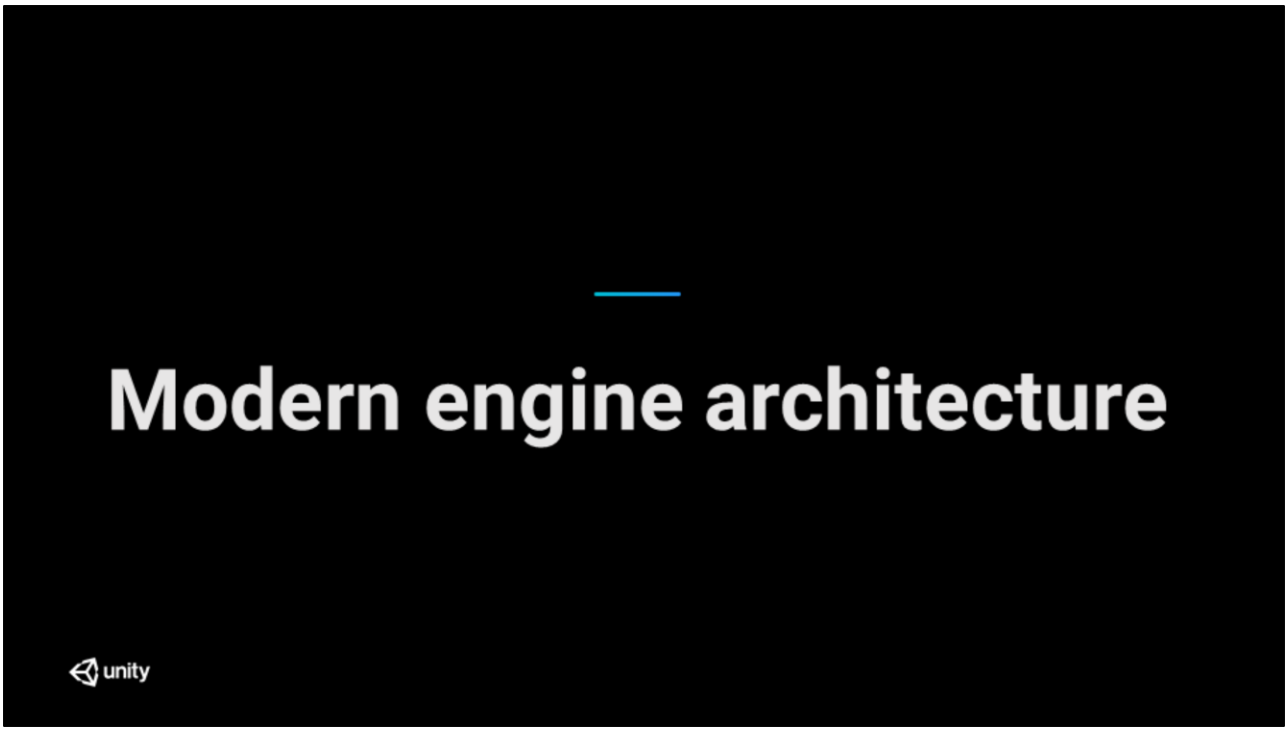
and even tiny game development studios can be making games with tremendous variety of environments, objects and visuals, like this game, D.R.O.N.E. – I'll let you take in all the different visuals and environments they were able to put together with a team of only five people using Unity

and even tiny game development studios can be making games with tremendous variety of environments, objects and visuals, like this game, D.R.O.N.E. – I'll let you take in all the different visuals and environments they were able to put together with a team of only five people using Unity


D.R.O.N.E.

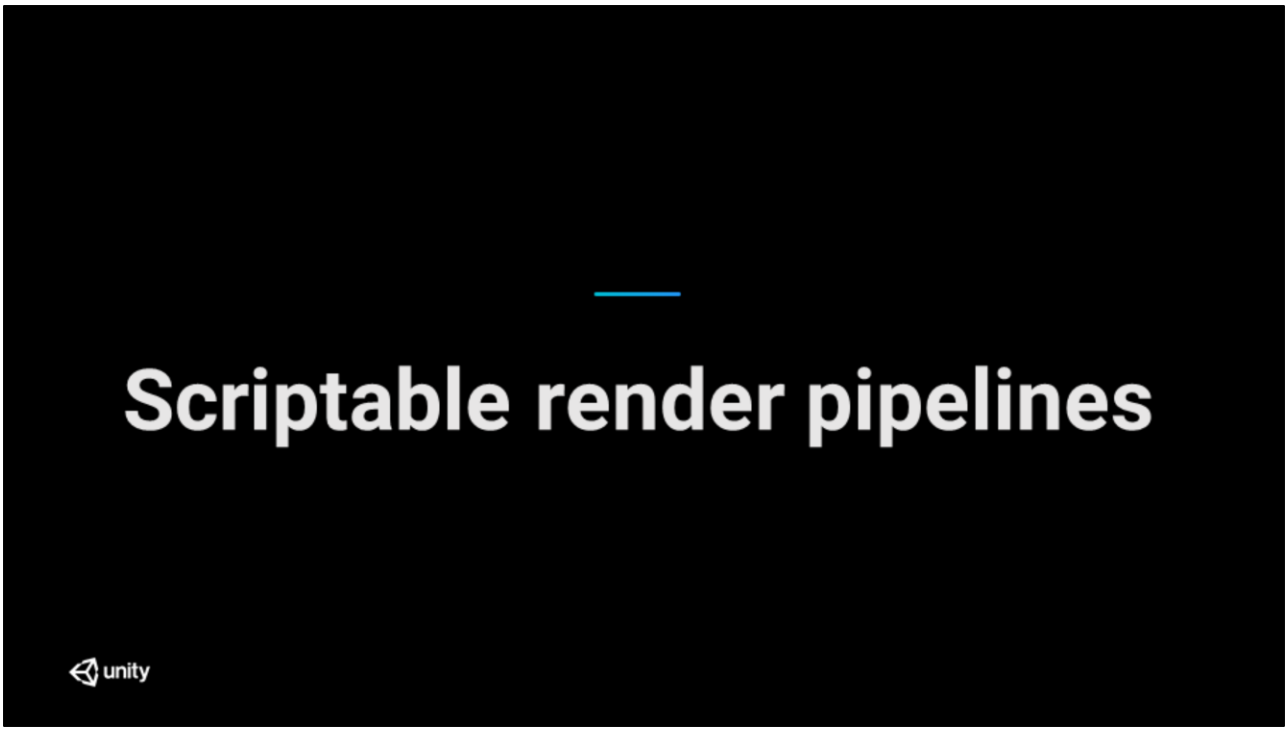

DRONE AREA
D.R.O.N.E.


D.R.O.N.E.

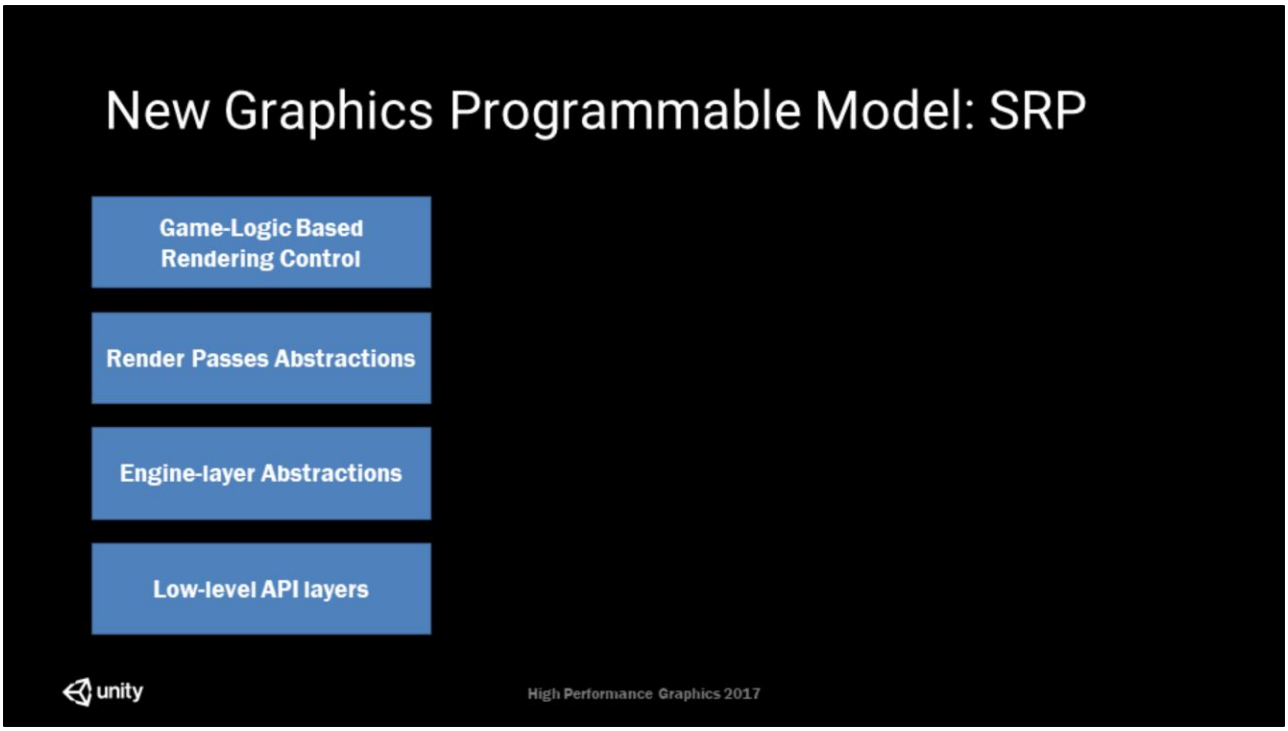


Coming back to our goals of evolving the game engine programmable model,
A modern game engine needs to be highly configurable, react dynamically to varied throughput, and make intelligent choices about platform constraints.

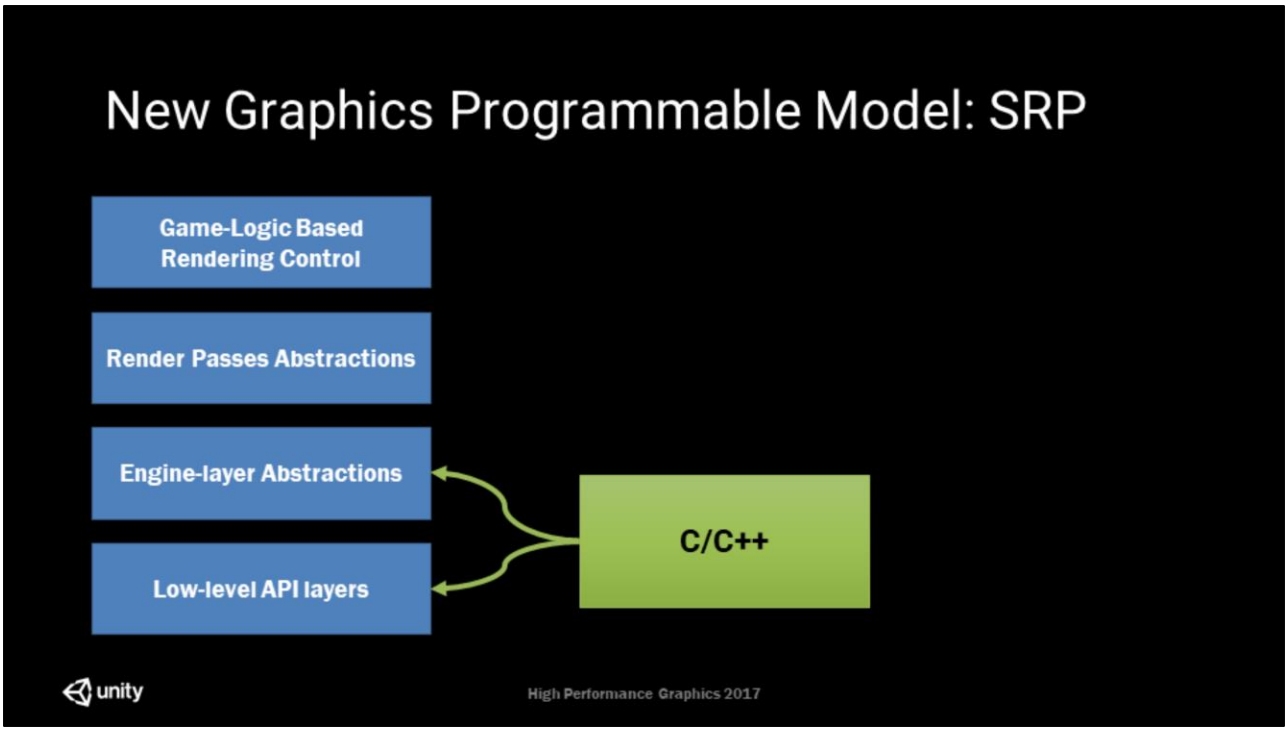
Unity’s motto is about democratization of game development
    Our core value
We are in the process of democratizing graphics development too
    Let’s speed up design and creation of new algorithms for games and interactive applications
And a little bit to help ourselves, too



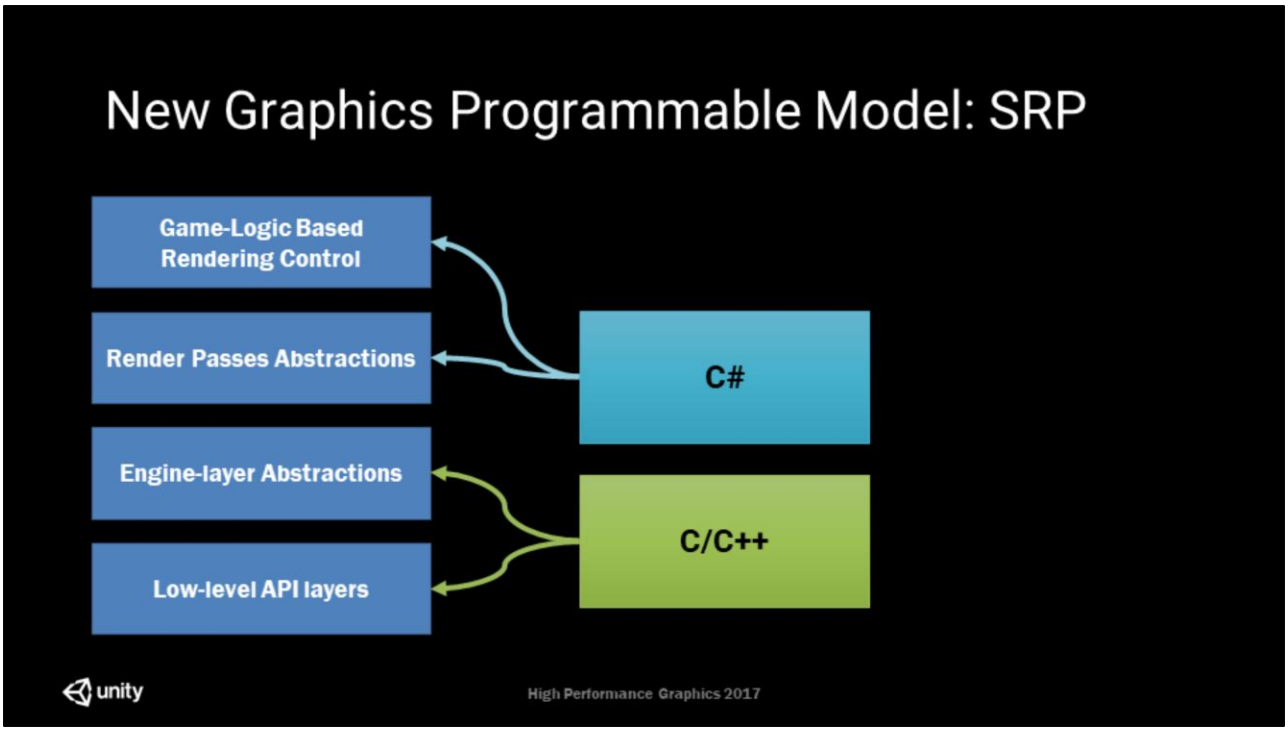
We are evolving the programmable model for graphics pipelines by going to Scriptable render pipelines architecture



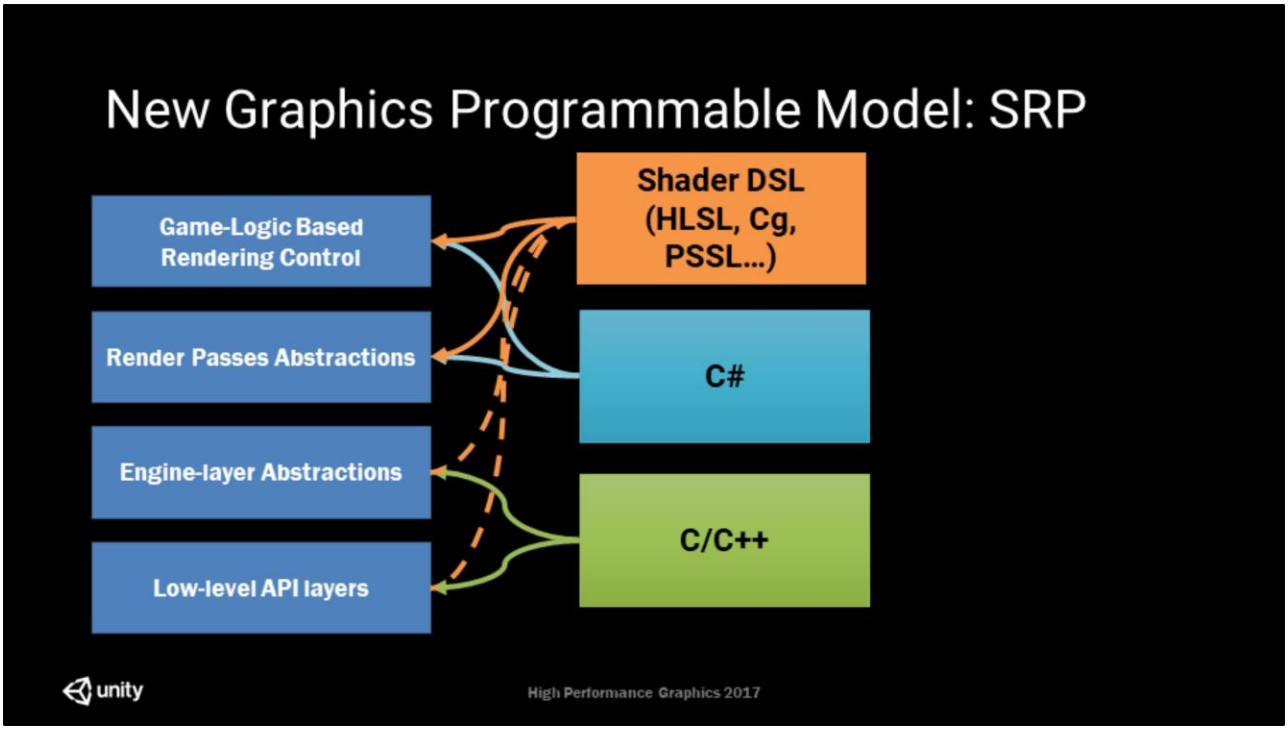
So with all of that in mind, let's look at the game engine pipeline architecture and see what shifts in the programming model



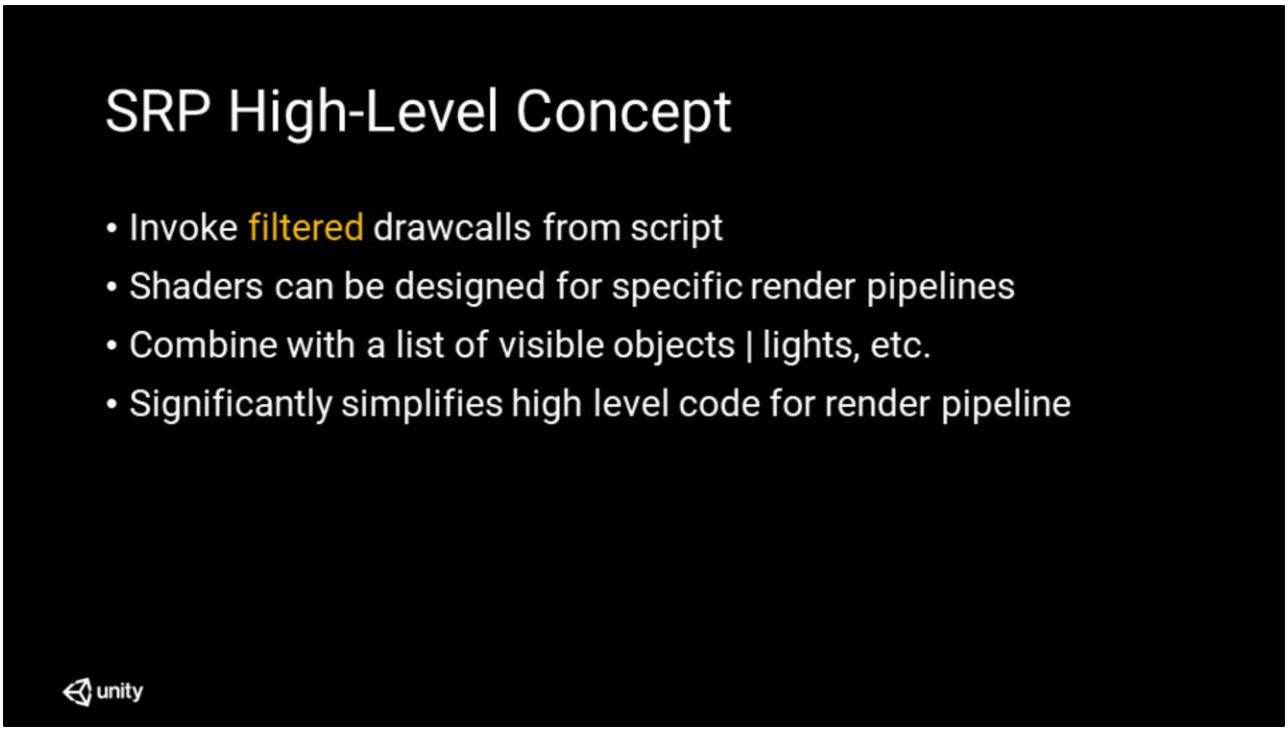
Low-level engine still stays in C++ land as before.



But we have moved some of the functionality for controlling render passes and game-based rendering into C#



Of course, the shader programming is still present, but shaders mostly need to agree just with C# layers (with the exception of hardware-specific features, which still need low-level exposure as before).



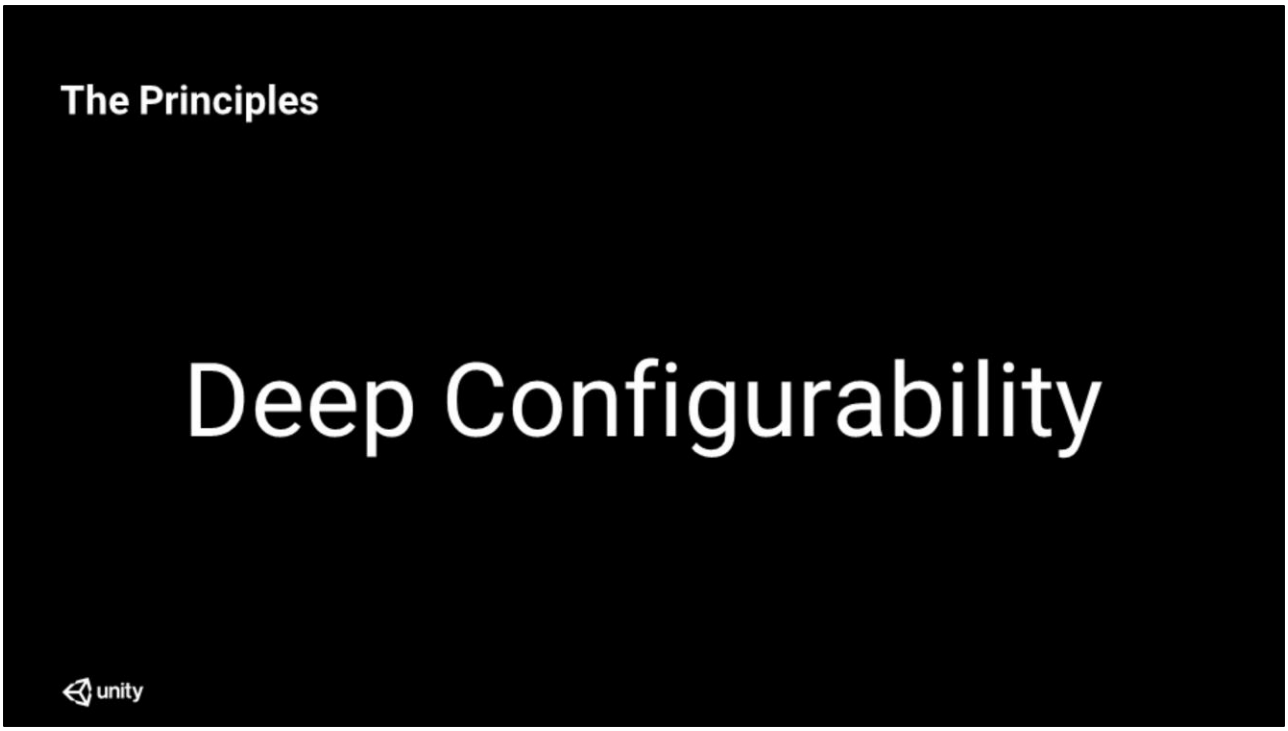
The interface is design such you invoke a draw call on the scene graph with a shader property provided as a parameter

- Similar to Destiny's render stages
- Under the hood the engine will search though the scene graph and identify anything applicable and then run it in a jobified generic fashion through the C++ low-level renderer architecture (culling / generation of drawcalls)

Shaders can be designed for a particular render pipeline in mind

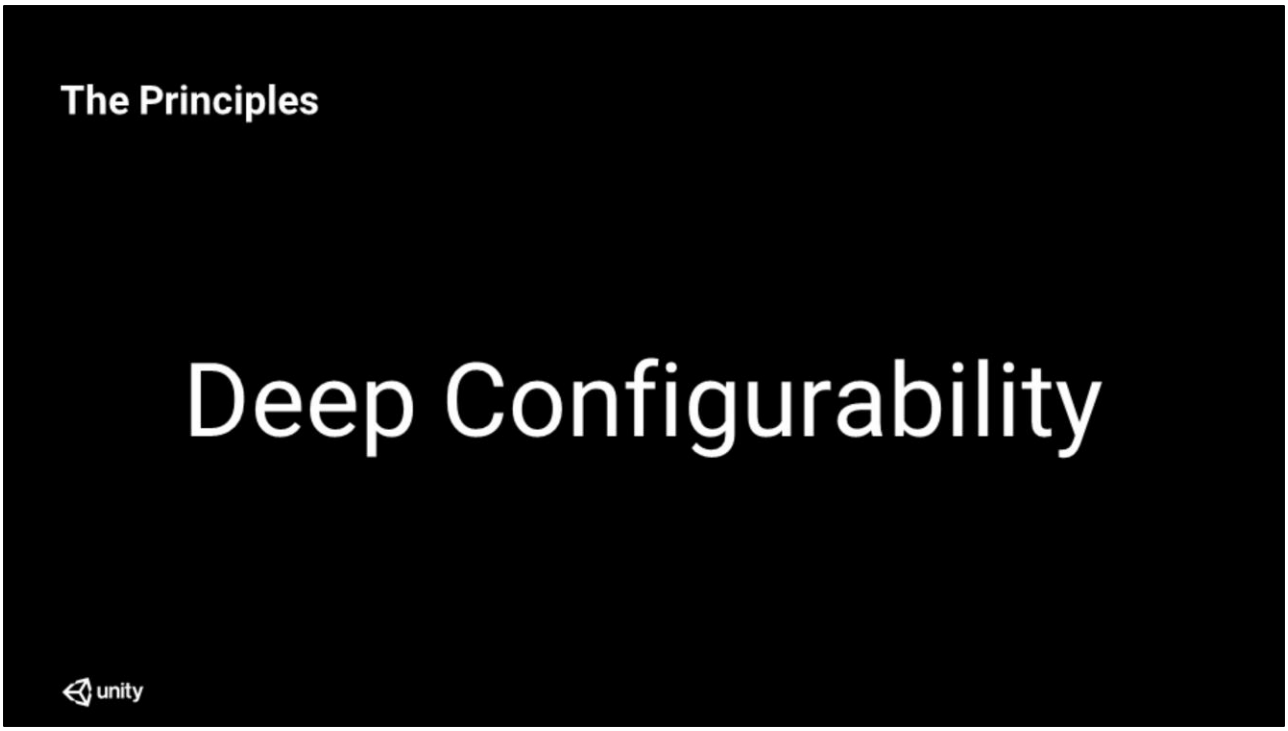
- Iterate on the same functionality in C# / shaders

Having this ability along with access to the list of visible lights and reflection probes reduces the rest of the rendering pipeline to advanced post processing which C# is very well suited for

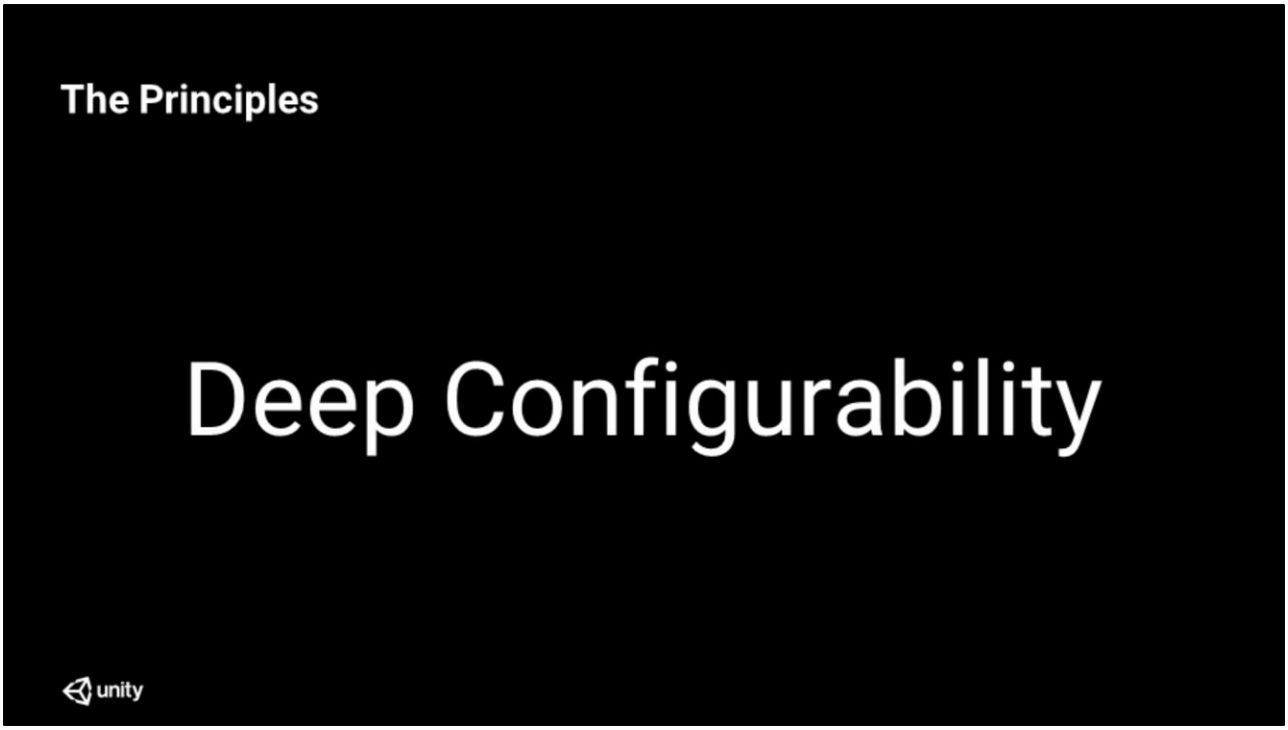


SRP makes it possible to make a completely custom render pipeline by moving this part of the renderer to C# level.

No longer tied to a one-size-fits-all low-level engine solution

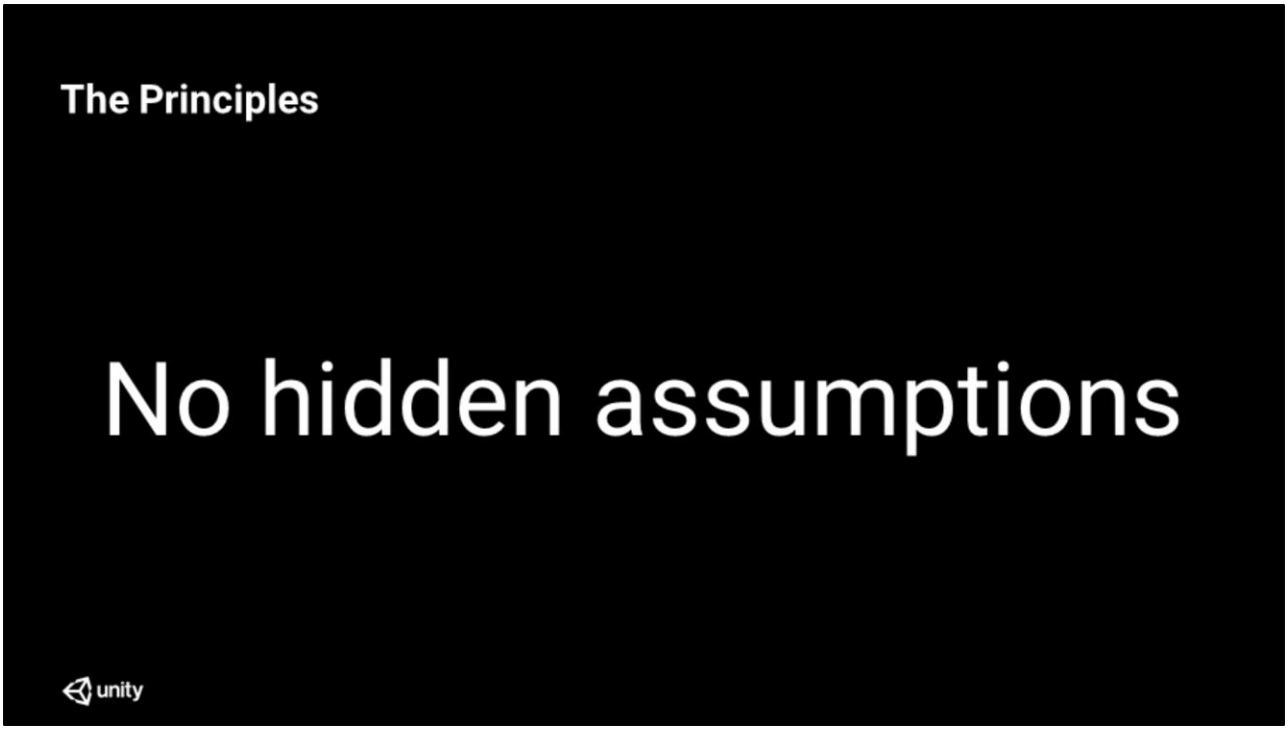


Configurability is easy based on needs
- Target platform
- Project / game specifics
  - Example: 2D / Inside / 3D
  - Why pay for what you don’t need?



Rendering pipelines need to be configured per platform

- Forward for VR
- Deferred for consoles
- FPTL deferred for high-end mobiles
- Forward for low-end mobiles
- Mixed-mode rendering for AR
- New stuff - You name it



The Game Engine (Unity) executable will make zero assumptions about the underlying rendering algorithms:

forward vs. deferred, compute based tiled deferred, compute based tiled forward, rasterized tiled deferred/forward, light data layout, g-buffer layout or even whether or not there is a g-buffer.

These are all explicit design choices at C# and shader level.

Making it easy to inspect the render pipeline algorithm work. High % of bugs are “this doesn’t work how I think it should work and it’s not documented” – make it easy to see this

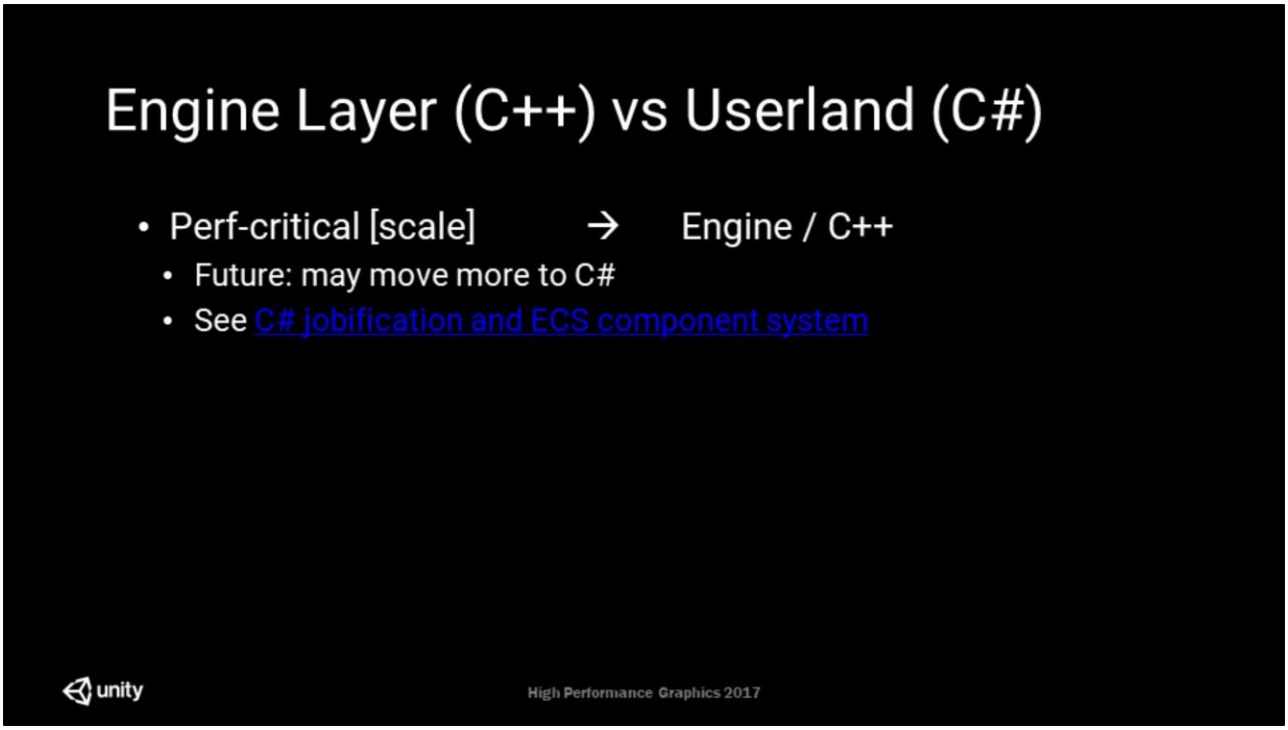
Benefits

- Anyone will be able to develop their own render pipeline whether it's one person in a garage or a large vendor.
- It is extremely helpful for cross industry collaboration since getting your work into Unity and broadcasting it doesn't require merging changes or consensus.
- A researcher or even an artist sitting at home can try an alternative render pipeline without the need for an army of engineers in the closet to merge changes and getting it to compile.

Since we have moved majority of iterative algorithm development into C# and shaders we can reap all the benefits of quick iteration with hot reload and parameter changes

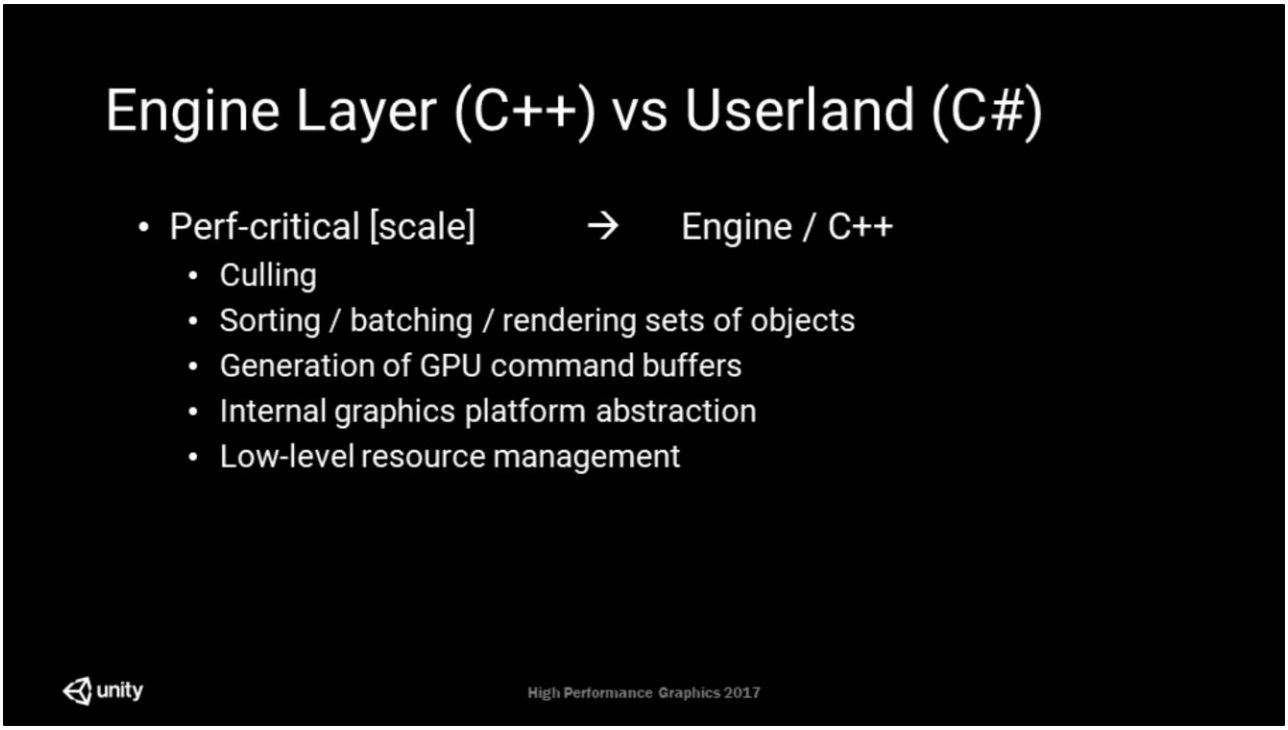
Of course, since our aim is to create amazing games, performance of this pipeline is key. No amount of flexibility and generalization will make that a non-requirement.

So how do we handle performance goals for SRP?



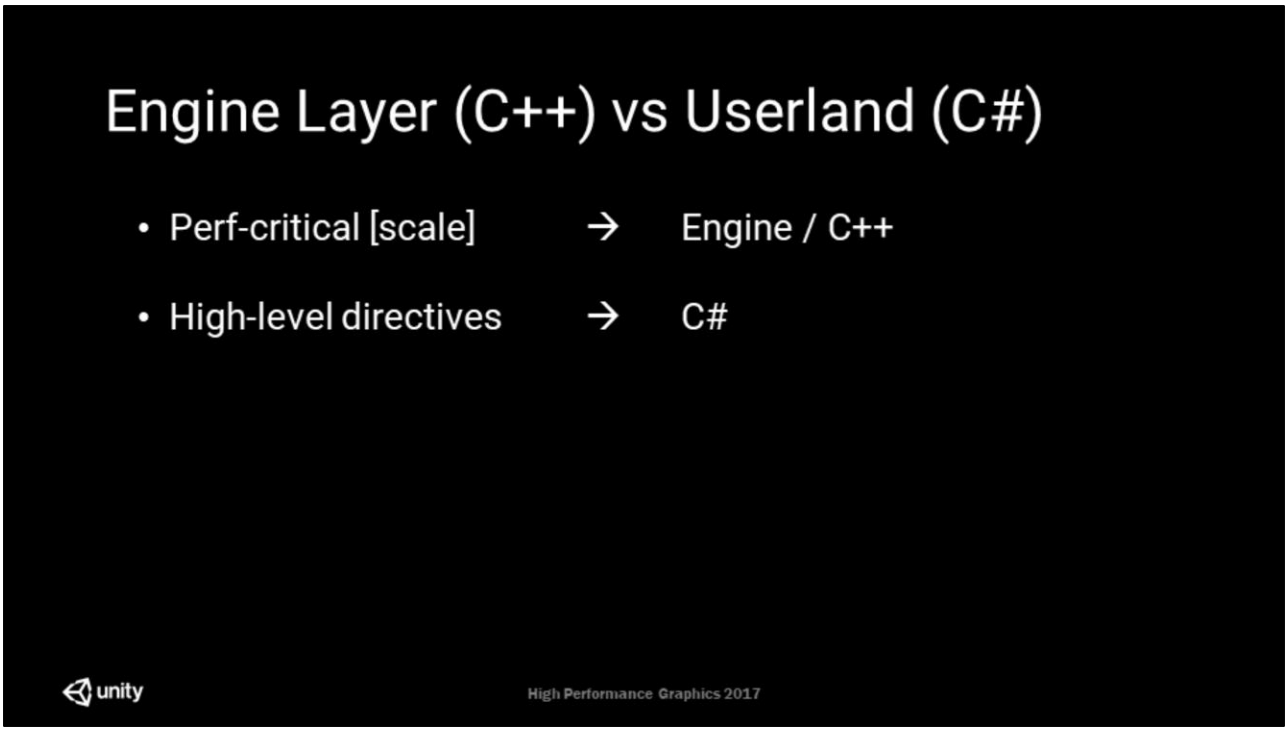
If it's perf critical, it's done in engine/C++

Future: maybe in C# if we can make it fast (ongoing research for fast C# jobified execution faster than native C++

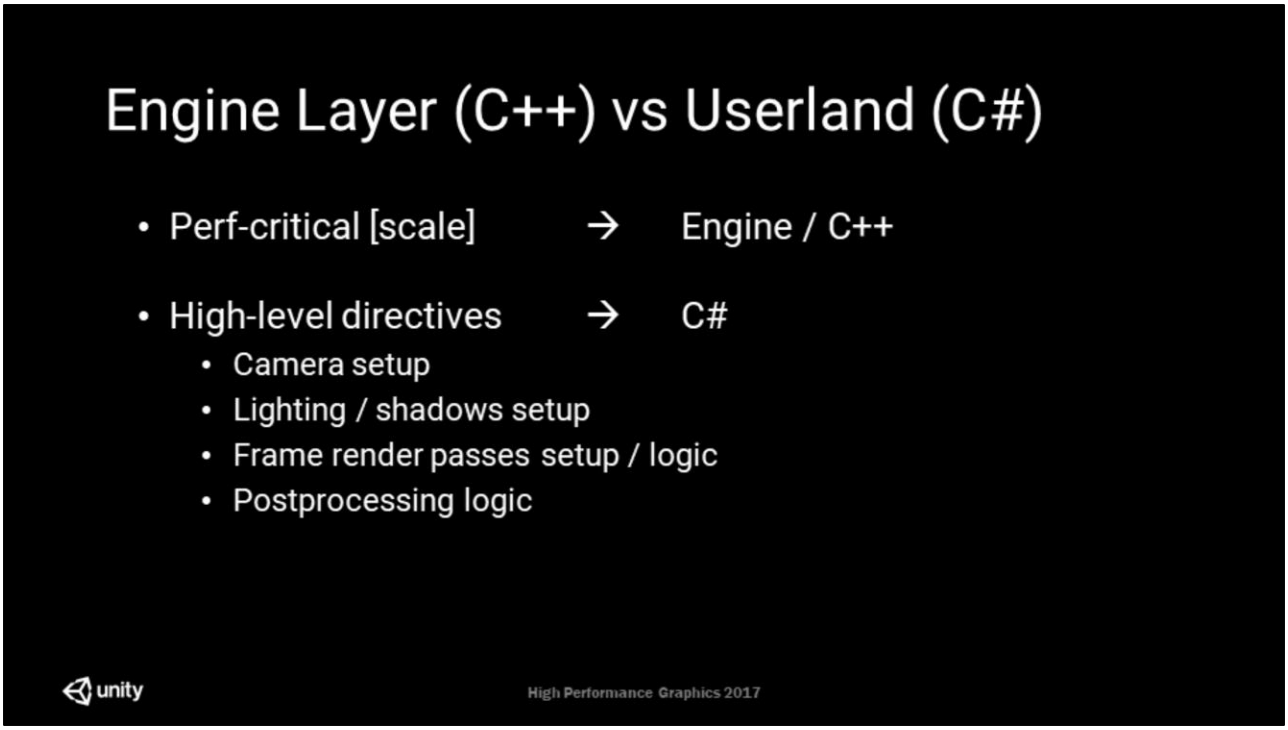


Some of the workloads that live in the C++ engine layer:

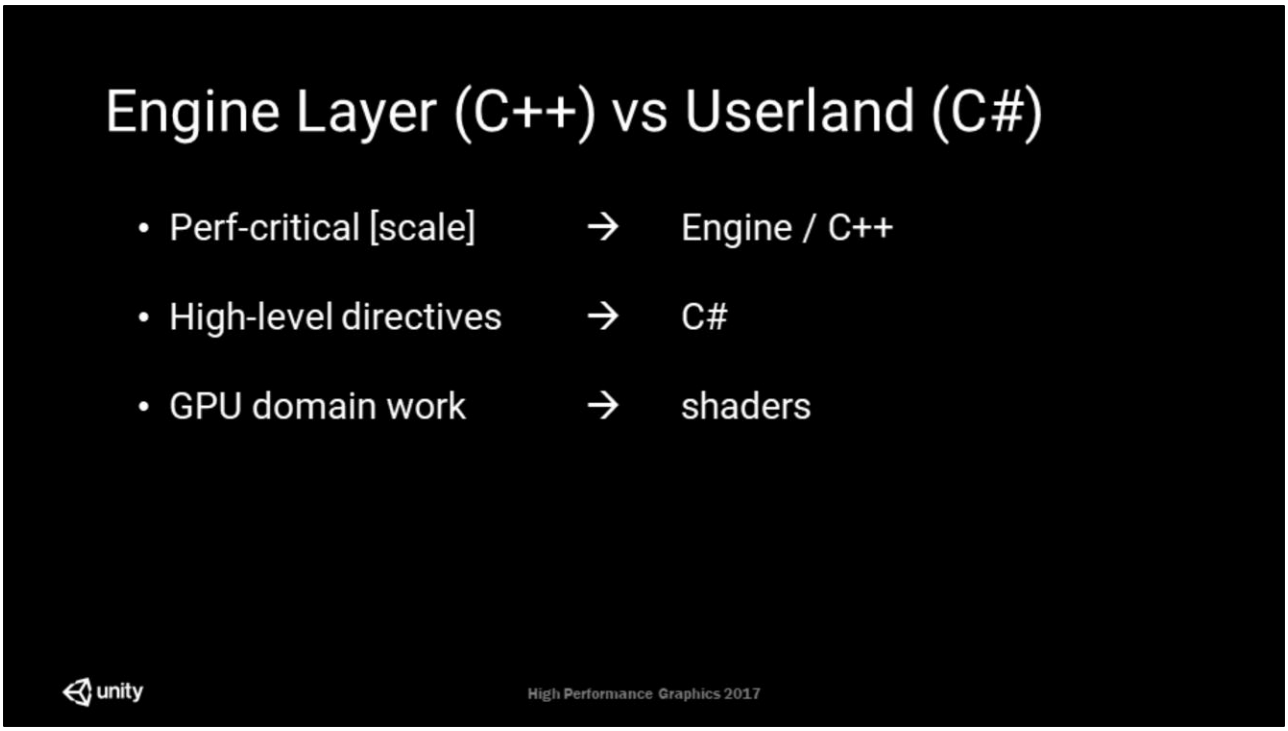
- Culling
- Sorting / batching / rendering sets of objects
- Generation of GPU command buffers
- Internal graphics platform abstraction
- Low-level resource management



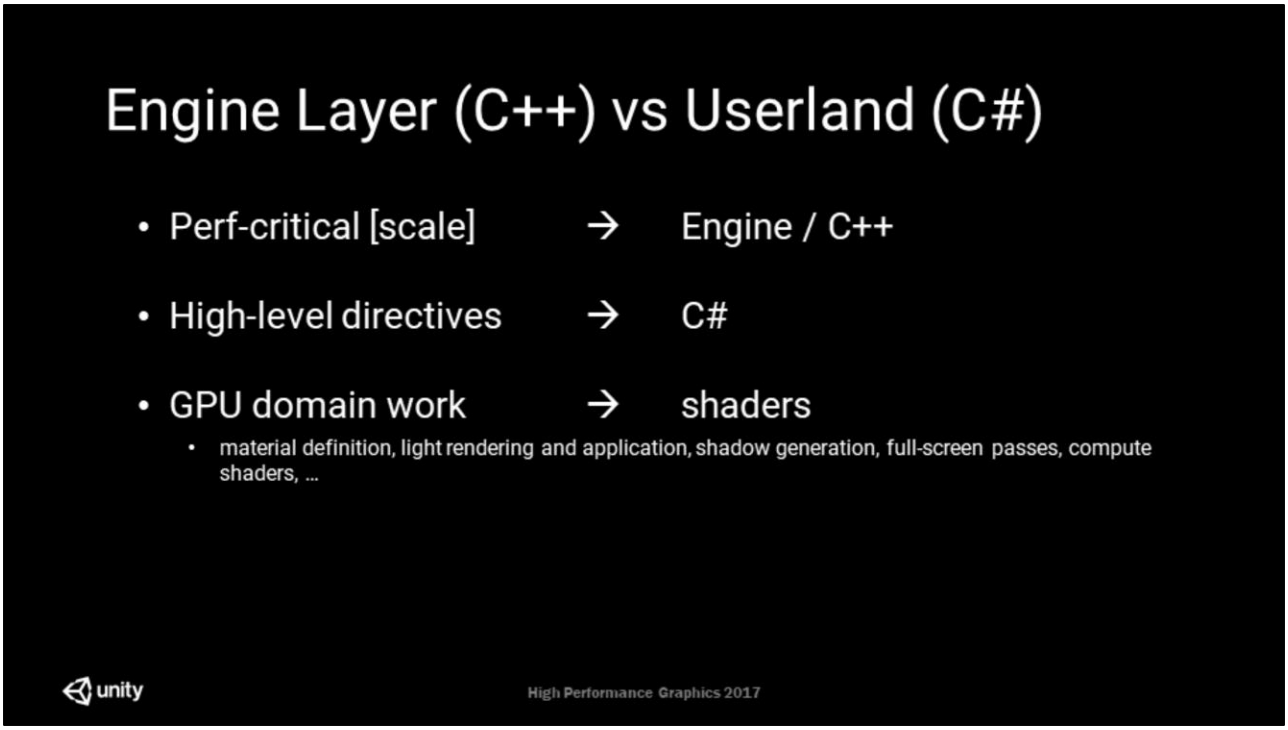
However, algorithm-level high-level directives will go into C# code – that's the heart of the render pipeline, the passes that we've looked at earlier.



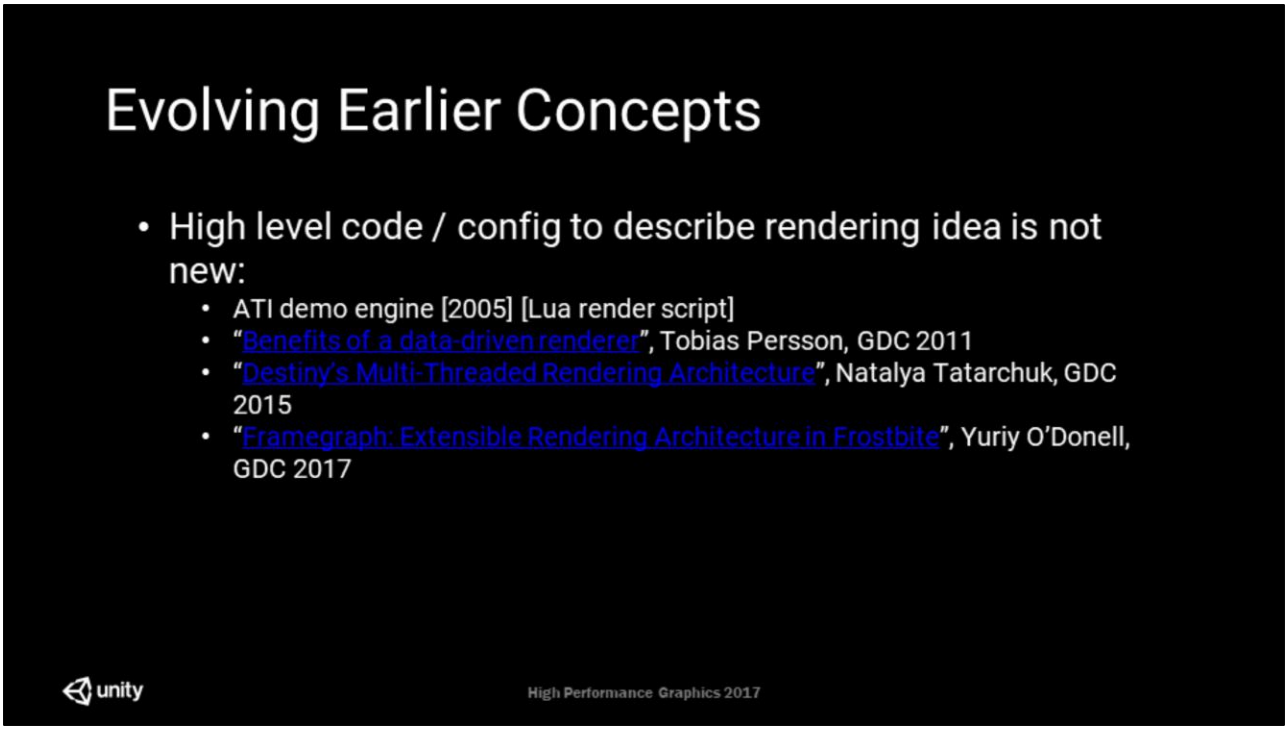
Camera setup
Lighting / shadows setup
Frame render passes setup / logic
Postprocessing logic
All go to the C# land. That’s where the fun is, truly!.



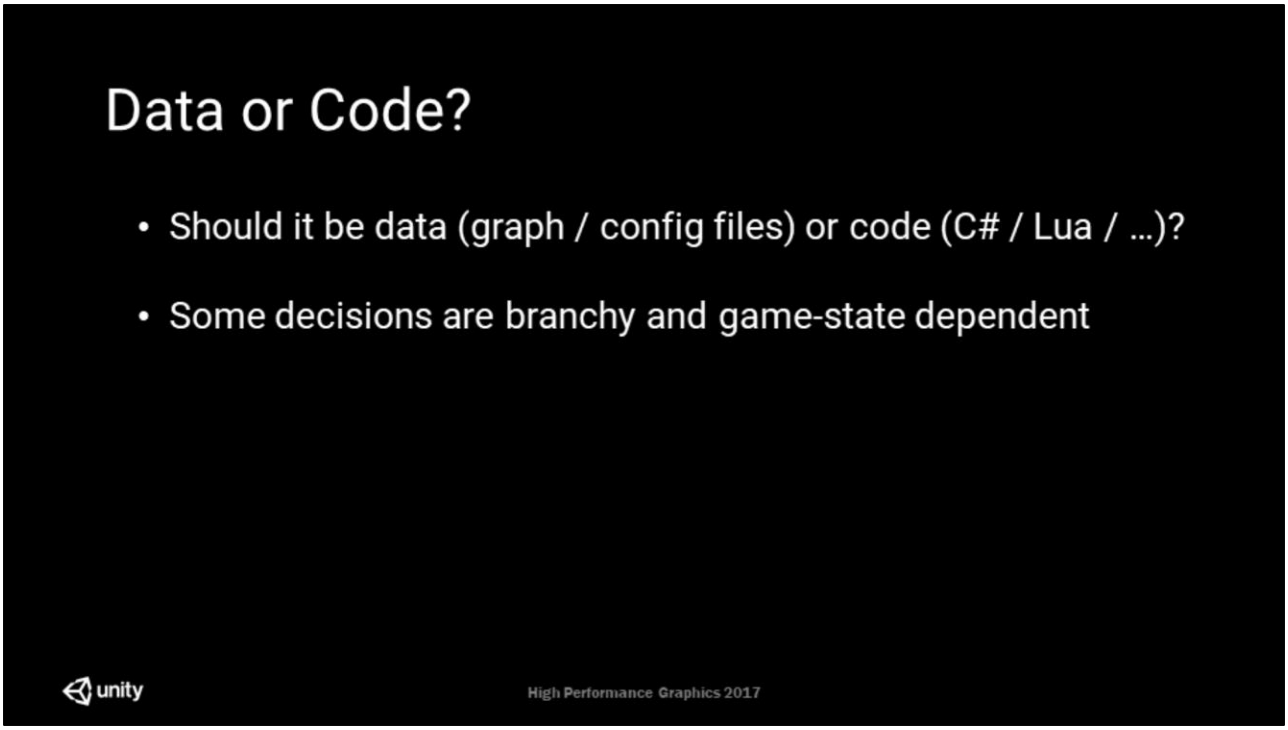
Of course, all GPU domain-specific workloads still remain in shaders – Shaders can be designed for a particular render pipeline in mind – this allows us to iterate on the same functionality in C# / shaders



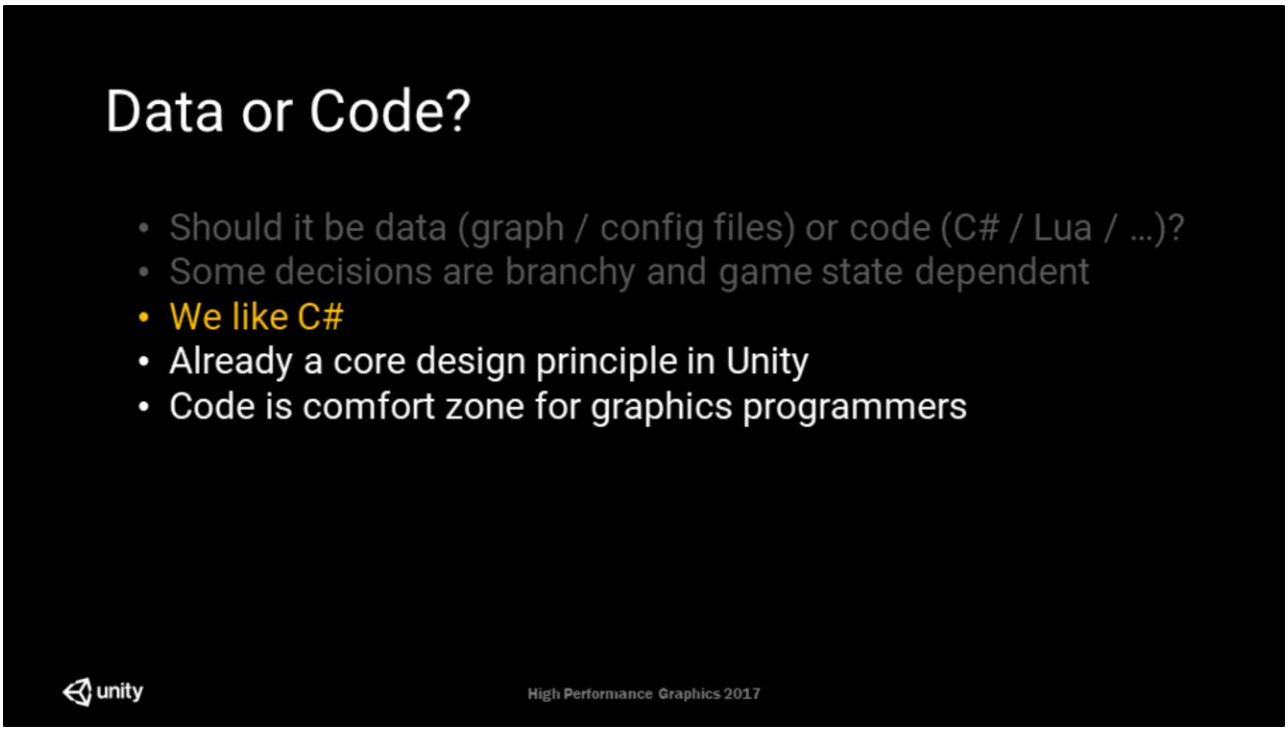
Shaders contain material definition, light rendering and application, shadow generation, full-screen passes, compute shaders, etc.



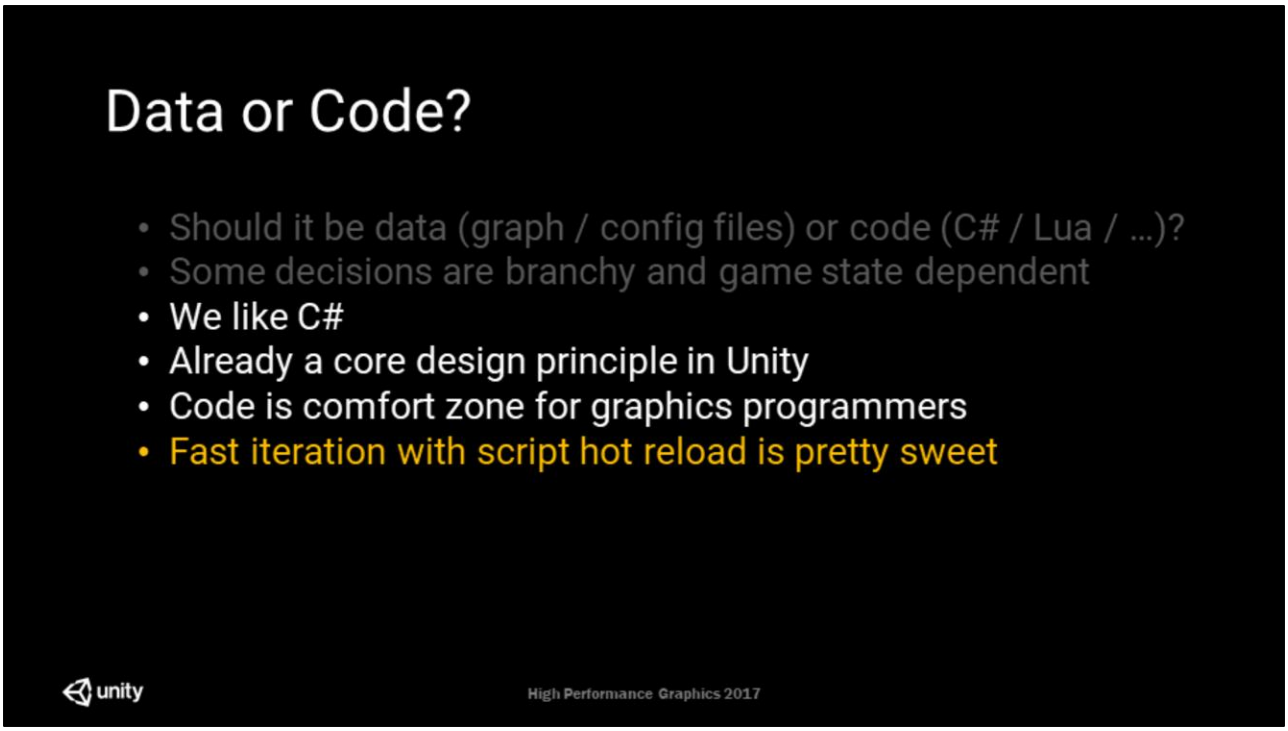
Our architecture is an evolution upon earlier concepts. High level code and configuration to describe rendering has been around the block. In fact, for my old ATI / AMD demo engine, Sushi, we used Lua to drive render pipelines. But I've listed a couple of other systems, including the one I developed at Bungie for Destiny, for various configurable pipelines.



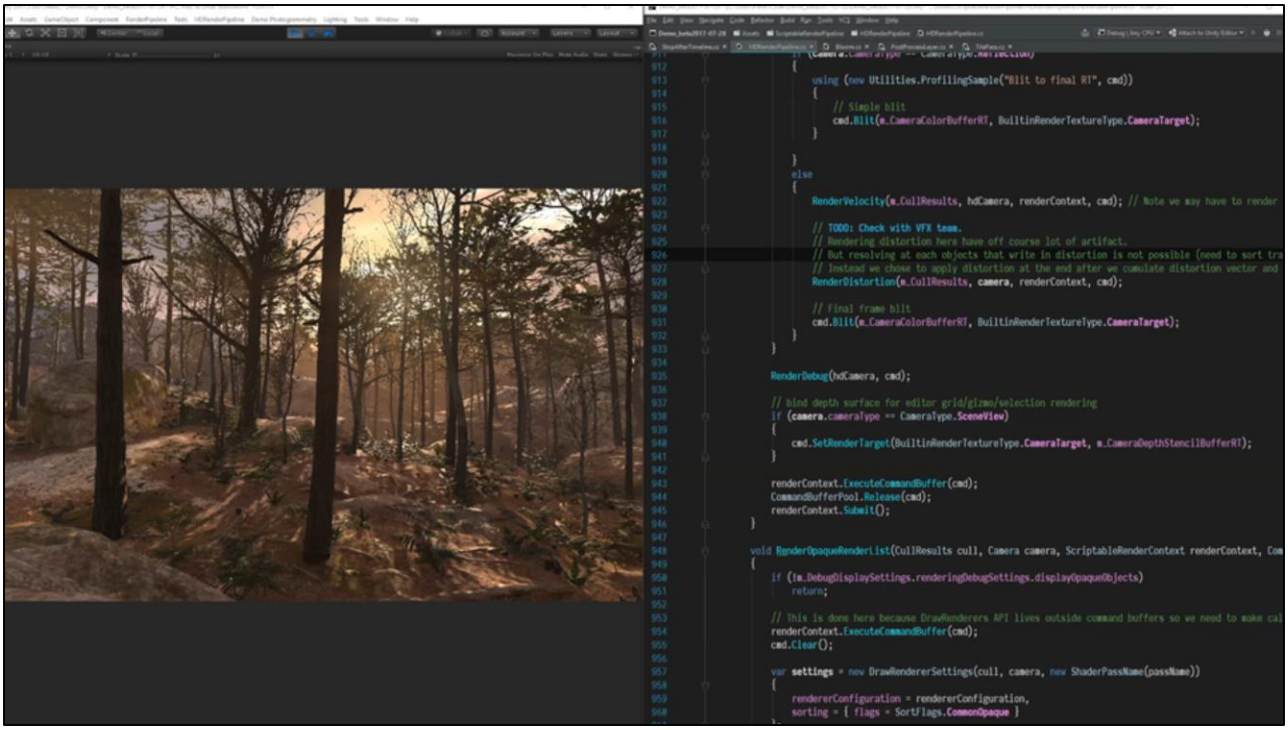
One of the decisions we had to think about is whether the configuration should be in expressed in data (via a node graph or config files that are parsed) or in code (script layer). What we've wanted to preserve from C++ land is that some decisions are branchy and dynamic (game-state dependent). That can be very tricky to express in data.



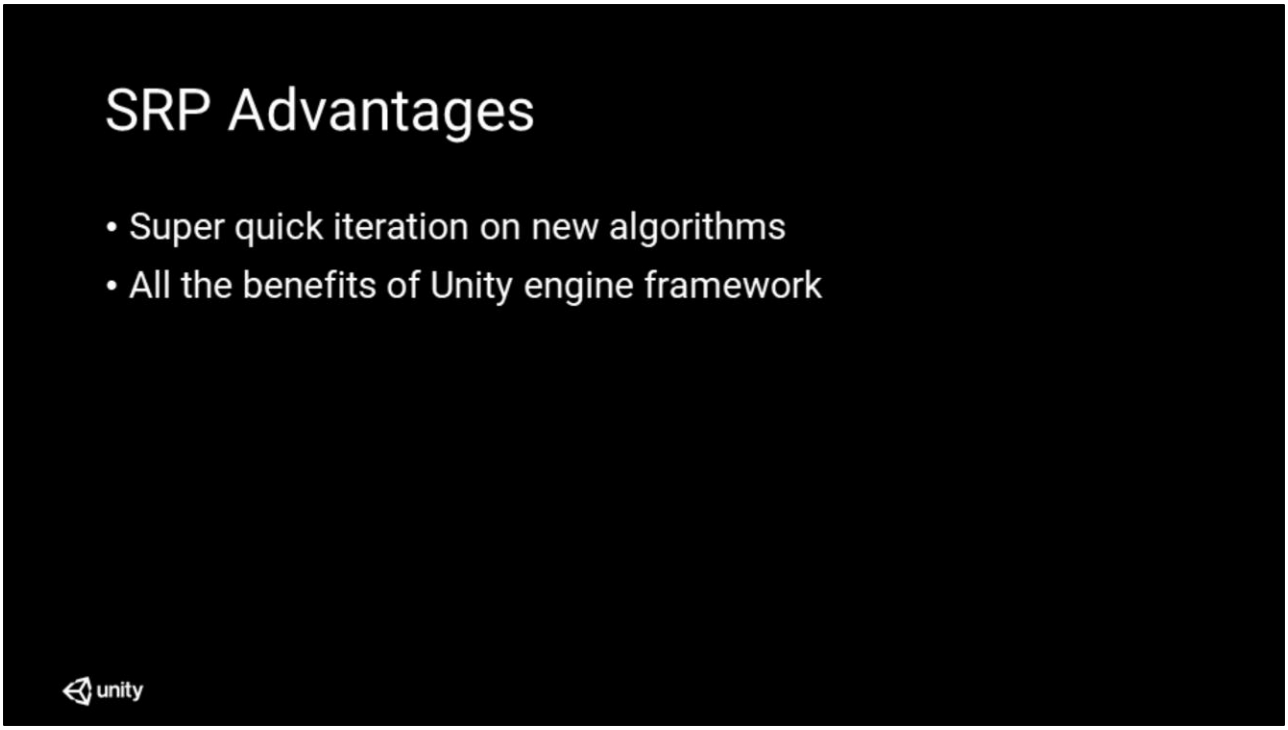
Plus we have huge affinity to C#, so in some ways, this was already a core design principle for Unity. Plus having a full debuggable IDE for coding the render pipeline brought an additional comfort zone element for programmers (and also not having to debug parsers is a nice plus!). And most



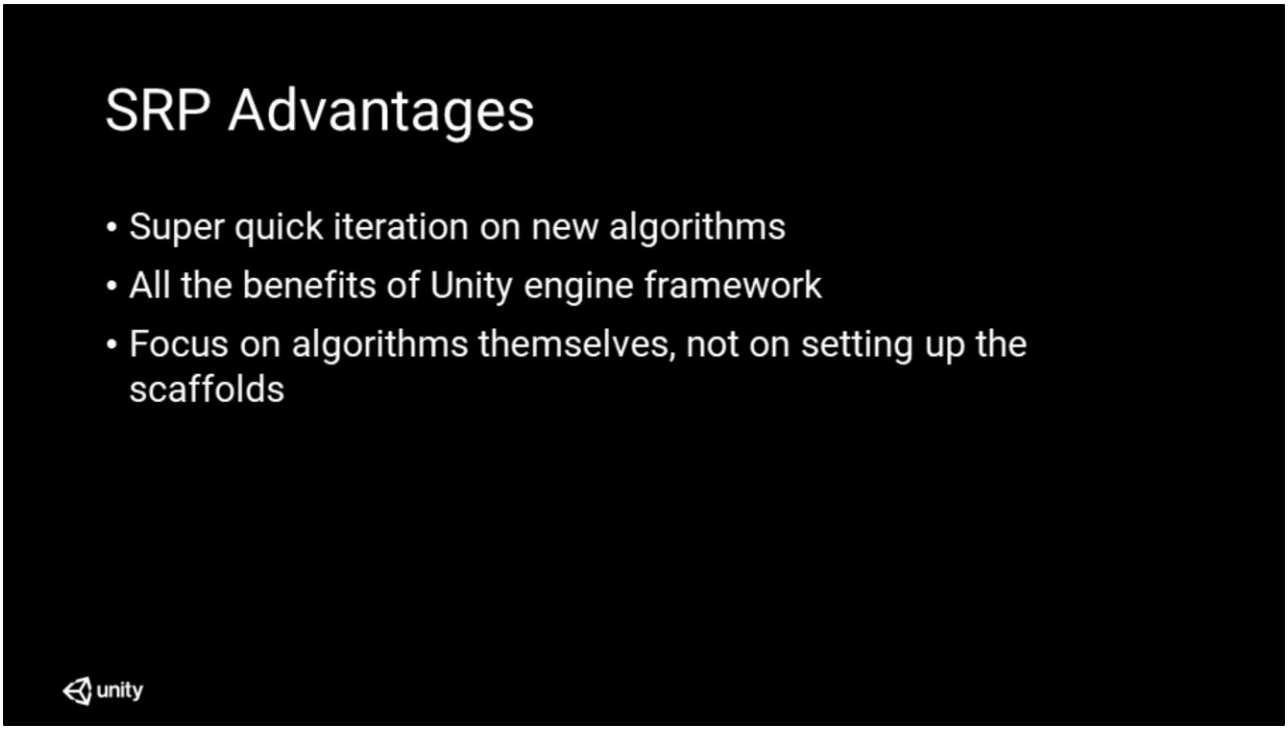
And I gotta say that fast iteration with script hot reload is a really nice thing for graphics development. It took me ten years to get that back, but I’m glad to have it back.



Here is an example - a video collected a couple of days ago (2.5 x speed up) of modifying a part of the render pipeline, hot reloading the scripts and seeing the changes in the visuals immediately. It’s ridiculous how much faster it is to iterate on graphics algorithms in this environment!.



SRP offers a number of advantages over the previous programmable models for game engines graphics:
- Super quick iteration for new graphics algorithms
- It gives you all the benefits of Unity as a mature game engine(resource management, loading / unloading, object lifecycle management, behavior scripting, animation / physics, etc., systems) that you don’t have to worry about implementing from scratch



Utilizing Unity you can take advantage of a stable and mature platform to develop your algorithms using high quality content (from the asset store, included packages, or self-made) and spend time working on the algorithms themselves rather than setting up a custom render engine.

In addition to the environment that is provided by unity, there is also in built testing tools SRP. These allow you to write unit tests to test the algorithm for regressions as well as against specification. This allows for optimisation work and forward development in a clean, safe, and low risk of regression environment.

SRP Advantages

- Super quick iteration on new algorithms
- All the benefits of Unity engine framework
- Focus on algorithms themselves, not on setting up the scaffolds
- All the performance benefits of low-level lean execution
- All the benefits of hot reloads in scripts

unity

Plus All the performance efficiency of low-level execution in native code
All the benefits of hot reload in scripts

Main C# SRP APIs

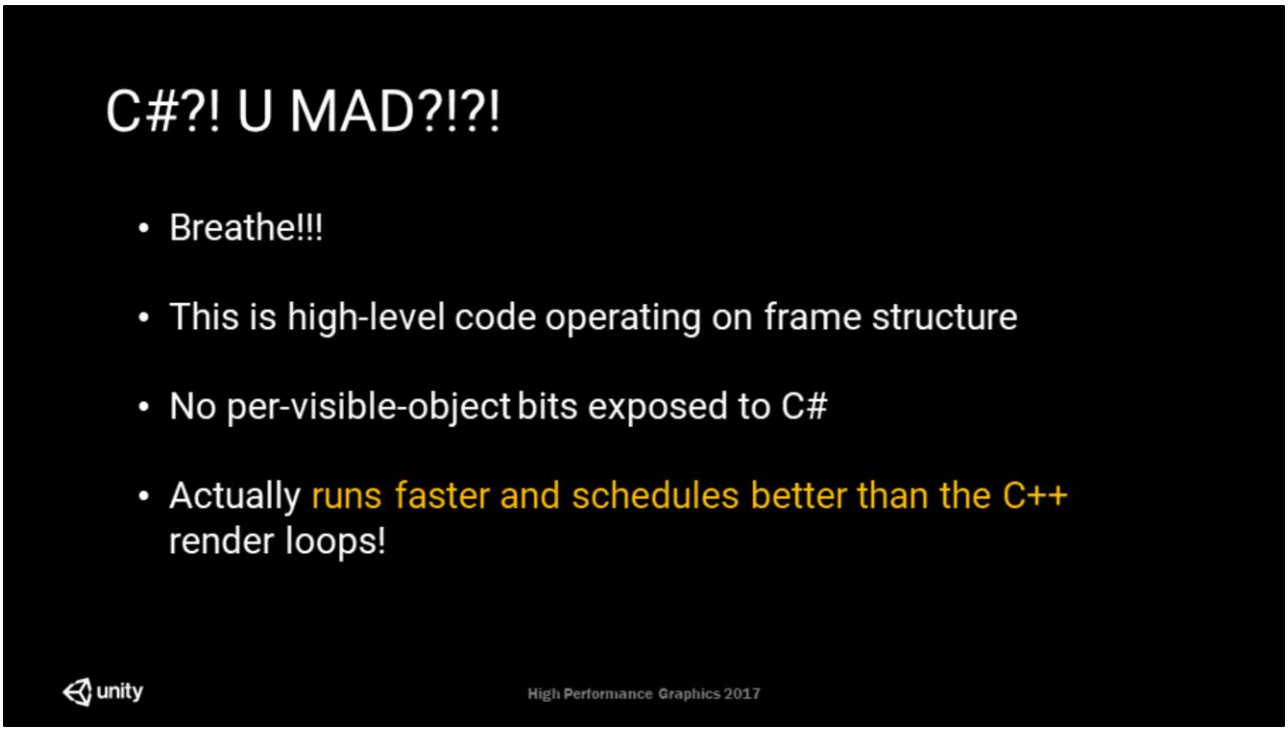
- Cull specific views
- Render subset of visible objects
  - With info on what material/shader passes to use
  - With sorting flags
  - With “what kind of per-object data to setup” (light probes, per-object light lists, etc.) to set up

Main C# SRP APIs

- Already existing APIs for:
  - Setting up render passes / render targets
  - Setting up shader constants / global resources
  - Dispatching compute shaders
  - Rendering individual meshes (for special fx / post fx)
- APIs build a “command buffer” that is later analyzed/executed

Of course people tell us we’re totally nuts for doing rendering from C#. Surely that will be terrible performance!!

Taking a short breath…

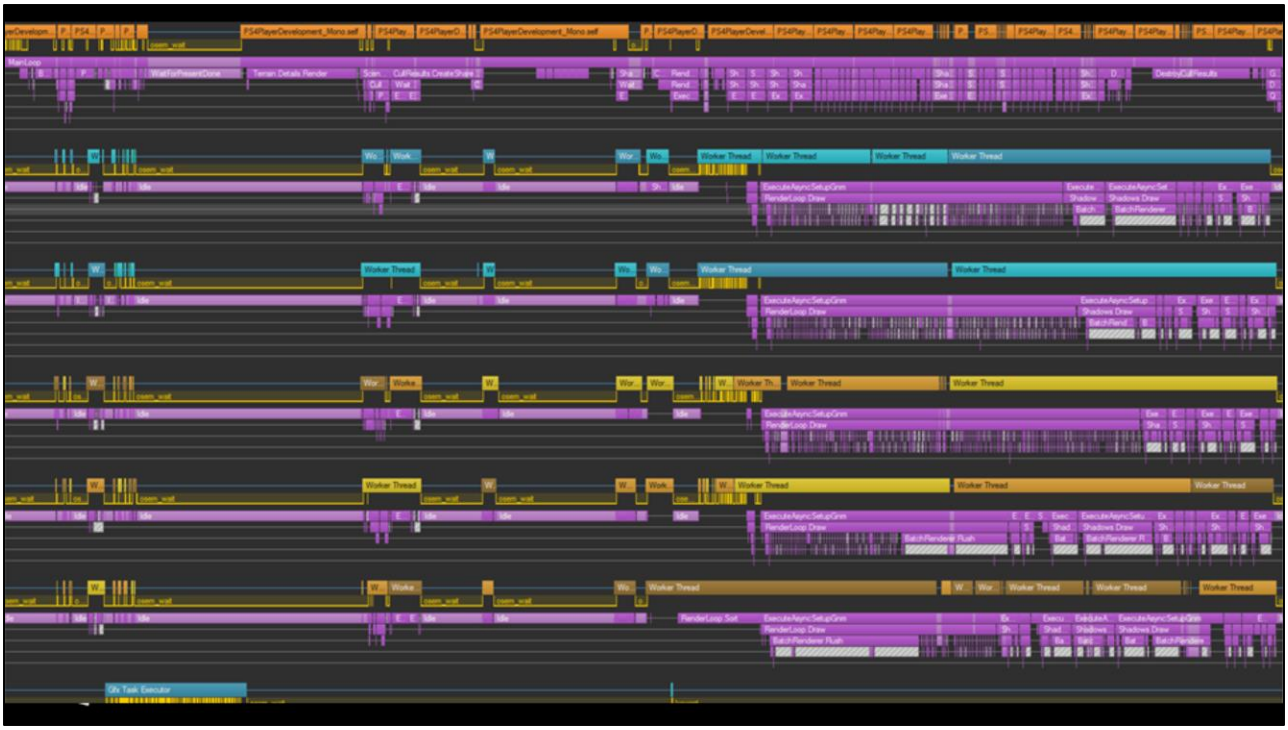


We see that our C# layers are just the high-level code operating on the frame structure. We don't expose any high-frequency per-object or per-drawcall control to C# - those stay in native code.
As the result the code actually runs faster and schedules better than our previous C++ render loops. Bonus!

Command Buffer Scheduling

- Everything is queued up
- Submit gets called
- Execution happens in order of added operation
- Each command is a job

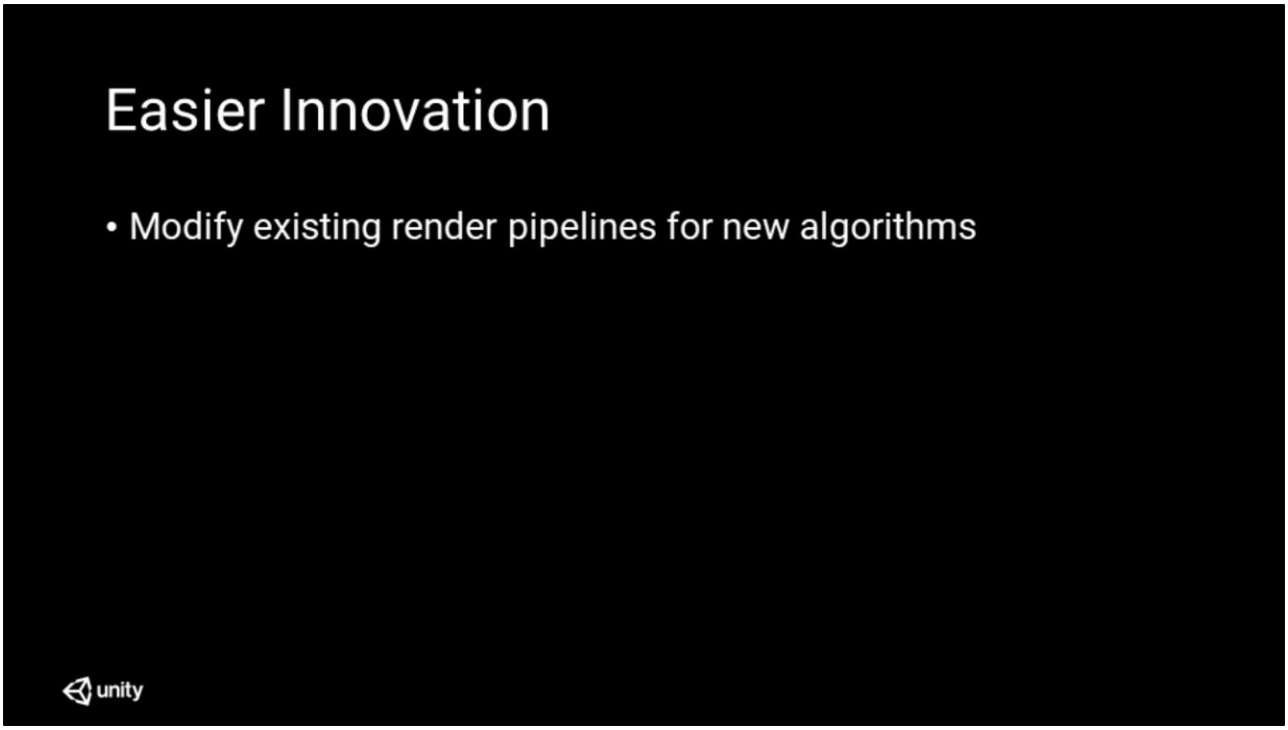
Previous execution where we had the main thread do a lot of the command logic (in this picture each row is a core.) and there were a ton of holes in rendering execution



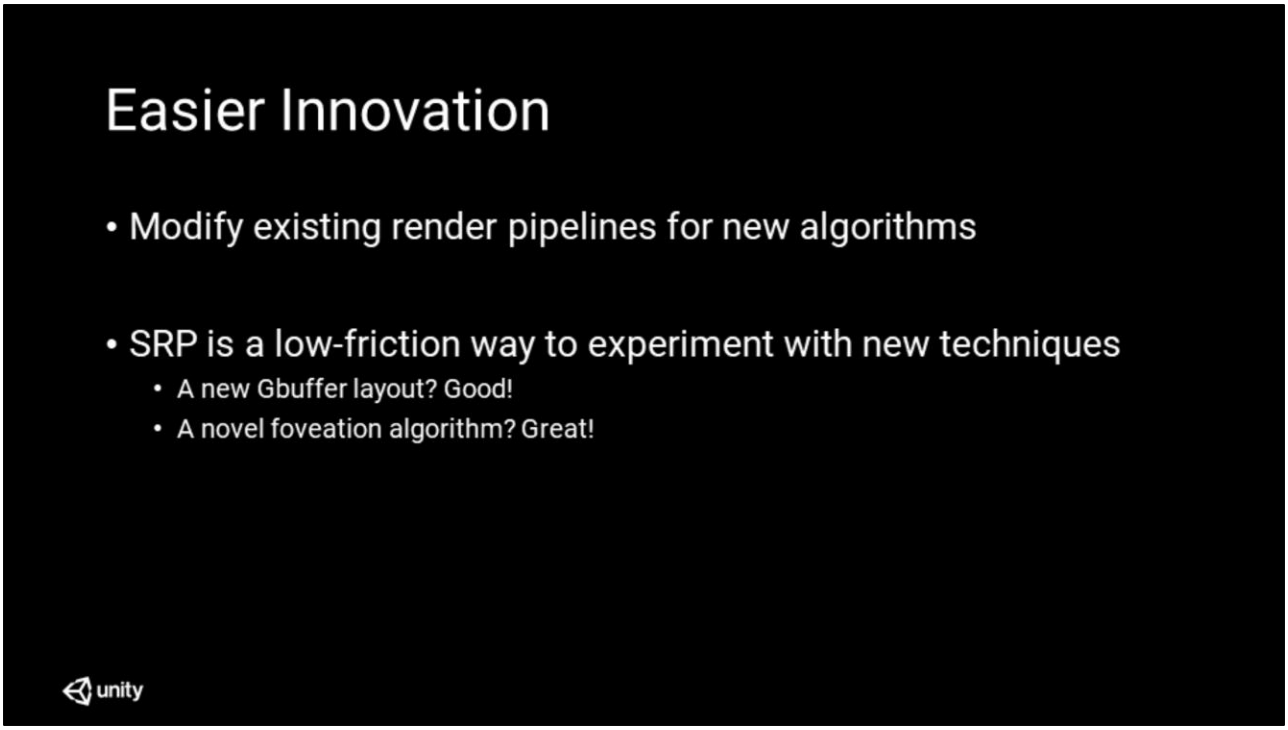
In the SRP execution, we are able to figure out the directives and quickly translate them to fast generation of command buffers, yielding far more lean performance with no stalls in the jobs. Much better latency!

Of course we are dog-fooding this architecture ourselves
We are in the process of designing several highly customized yet configurable pipelines
our own render pipelines going forward will also be using this same C# interface which our users have access to as well.

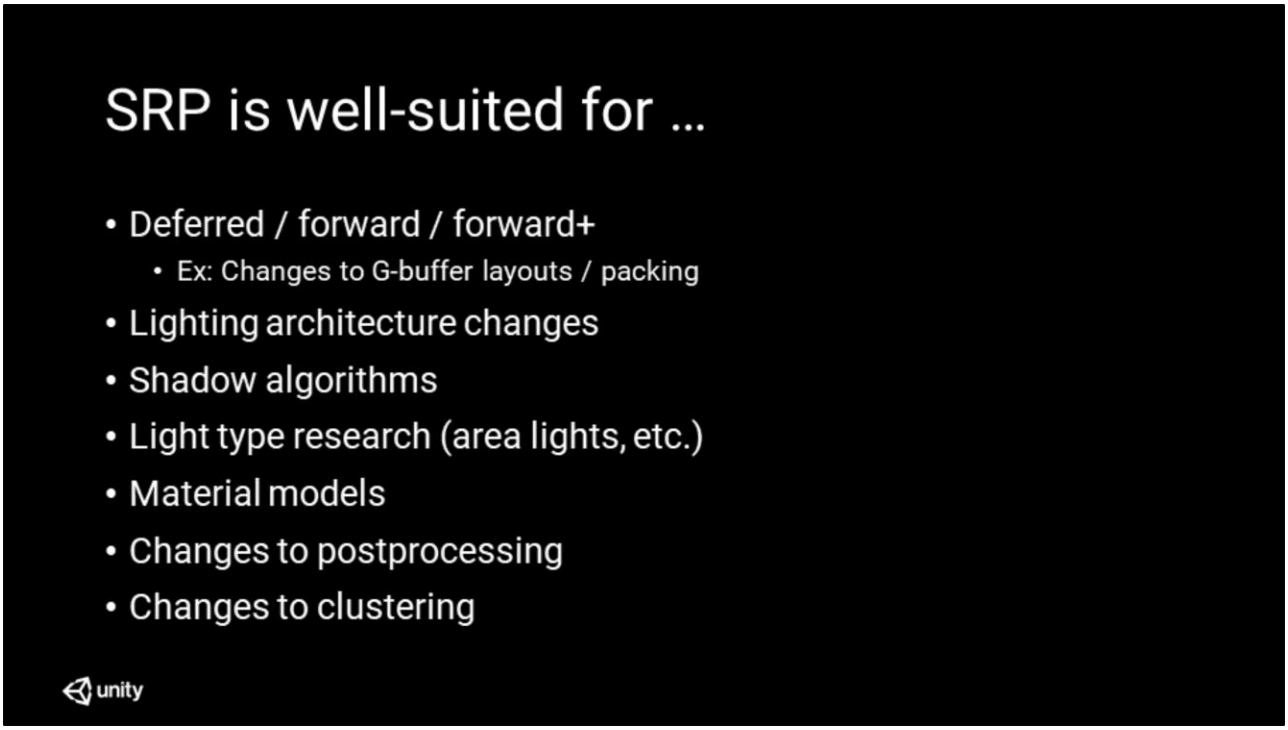
We are developing a Lightweight pipeline (highly optimized pipeline able to serve low-end mobile to consoles) – a clean, optimized rendering, not requiring full bells and whistles
And a HQ render pipeline –layers and layers of bells and whistles (full PBR, forward/deferred, FPTL clustered, etc.)
Both come with ability to do single-pass rendering for VR in the respective SRPs.



We are hoping this architecture will ease development of new graphics algorithms and pipeline by the community. You can use some of the example rendering pipelines and modify
With SRP you are generally working at the level of the graphics algorithm, that is, you are drawing meshes, setting state, executing compute shaders, and similar.



SRP provides a low friction way to work with experimental graphics algorithms in a feature rich engine that features editing tools, animation systems, custom scripting, and performance analysis tooling.
For example, If you want to make small modifications to one of the pipelines, say a different GBuffer layout for instance, you can use HD as base to do that. Or Foveated rendering is a great technique to experiment with in SRP framework



Not needing new HW or low-level API features are well-suited – won't need C++ engine layer changes

Deferred / forward / forward+

Ex: Changes to G-buffer layouts / packing

Lighting architecture changes

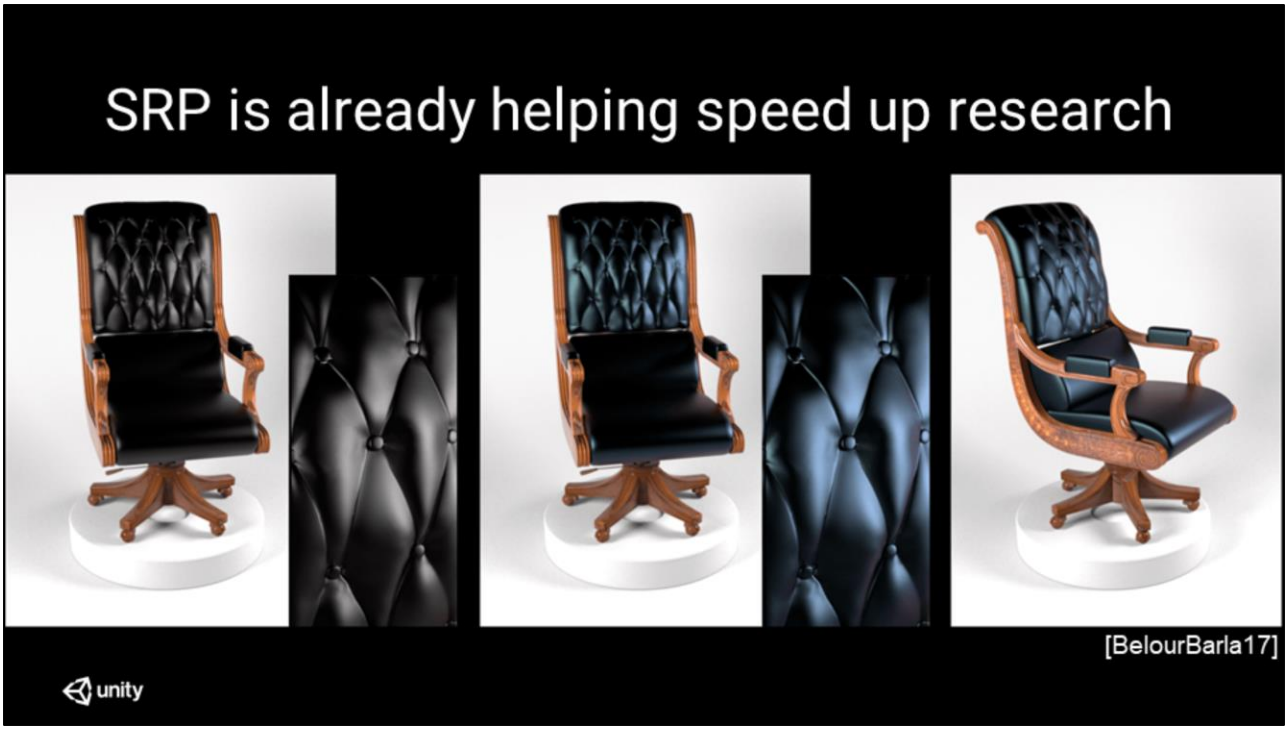
Shadow algorithms

Light type research (area lights, etc.)

Material models

Changes to postprocessing

Changes to clustering



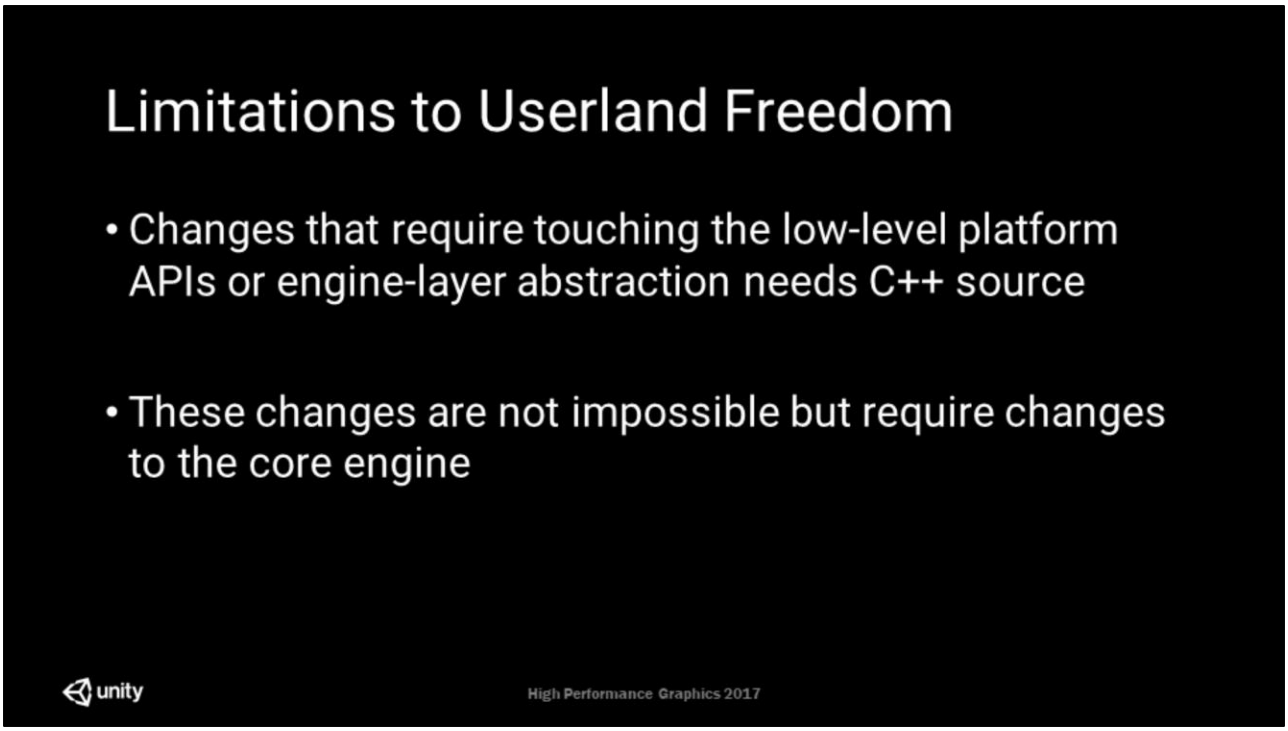
SRP already has been used in research, for example, this year's SIGGRAPH paper

"A Practical Extension to Microfacet Theory for the Modeling of Varying Iridescence" (by Laurent Belour and Pascal Barla). The researchers were able to use the light types and the shaders and just focus on the material model research using the SRP.

https://belcour.github.io/blog/research/2017/05/01/brdf-thin-film.html

“A Practical Extension to Microfacet Theory for the Modeling of Varying Iridescence” (by Laurent Belour and Pascal Barla). The researchers were able to use the light types and the shaders and just focus on the material model research using the SRP.

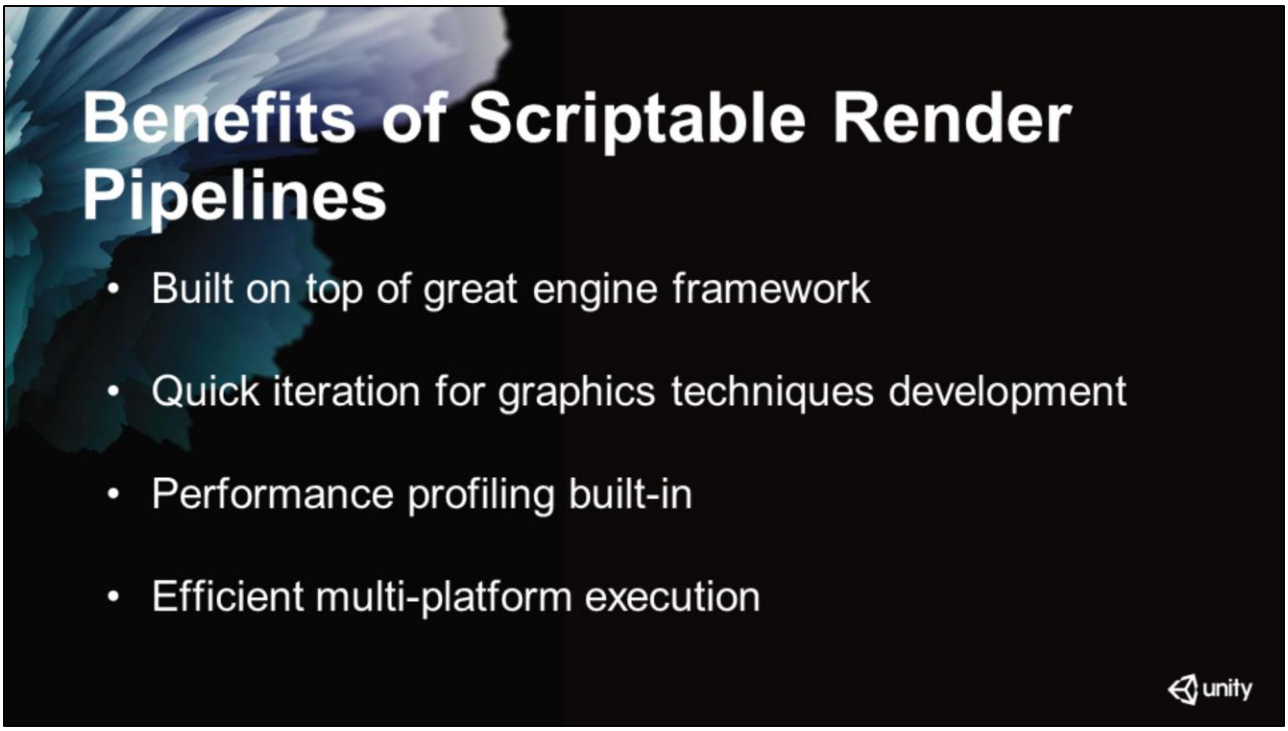
https://belcour.github.io/blog/research/2017/05/01/brdf-thin-film.html



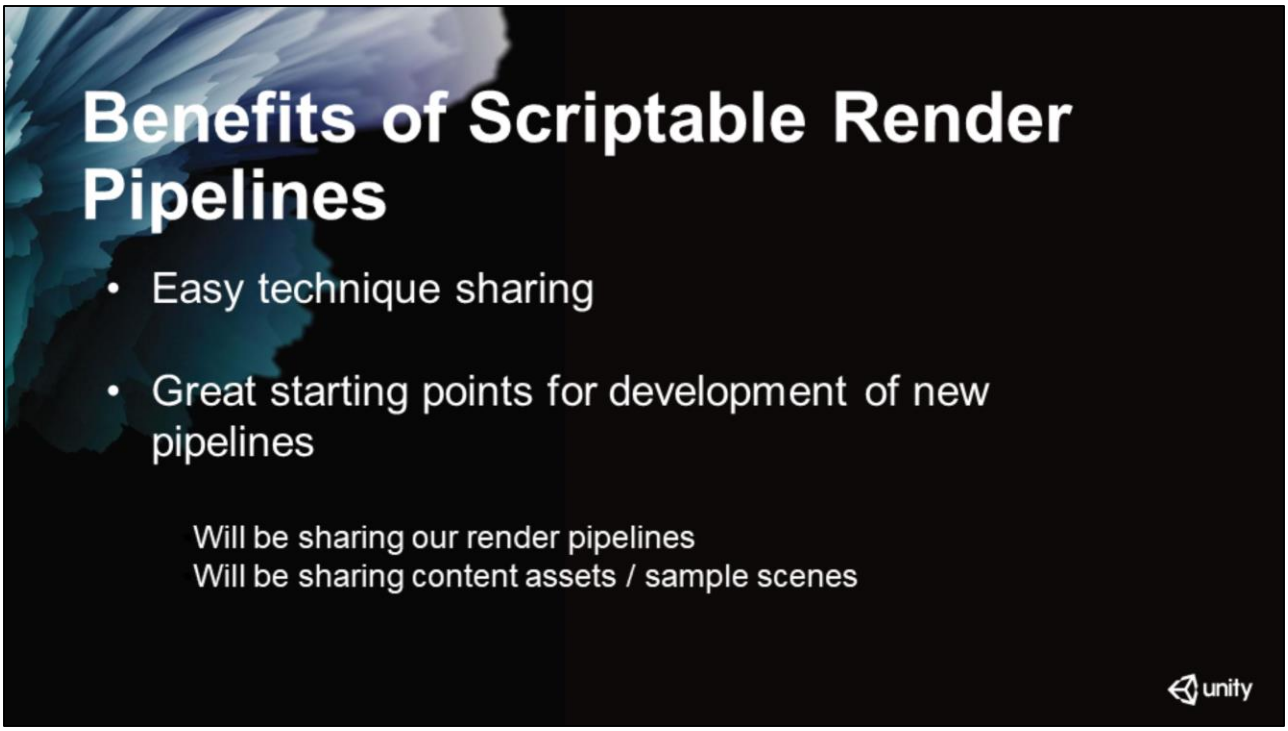
Of course, there are limitations that will need changes to the underlying engine layes themselves (i.e. C++)
Examples that needed engine modification

- Low-level features: adding asynchronous compute API, instancing API
- Example: integrating tessellation requires changes to the core low-level platform / device API layers. That won't be easier with SRP, since it'll require core engine changes
- Hardware-facing features outside of existing APIs

However, as each feature is added, we extend the SRP API to provide easy access to C# layer



So to summarise, graphics programming models can benefit greatly from faster iteration. We believe SRP provide a Great framework for all the support structure (asset management, etc.)
Scriptable render pipelines allow quick iteration for graphics algorithm development
Yield better performance through jobified command buffer generation



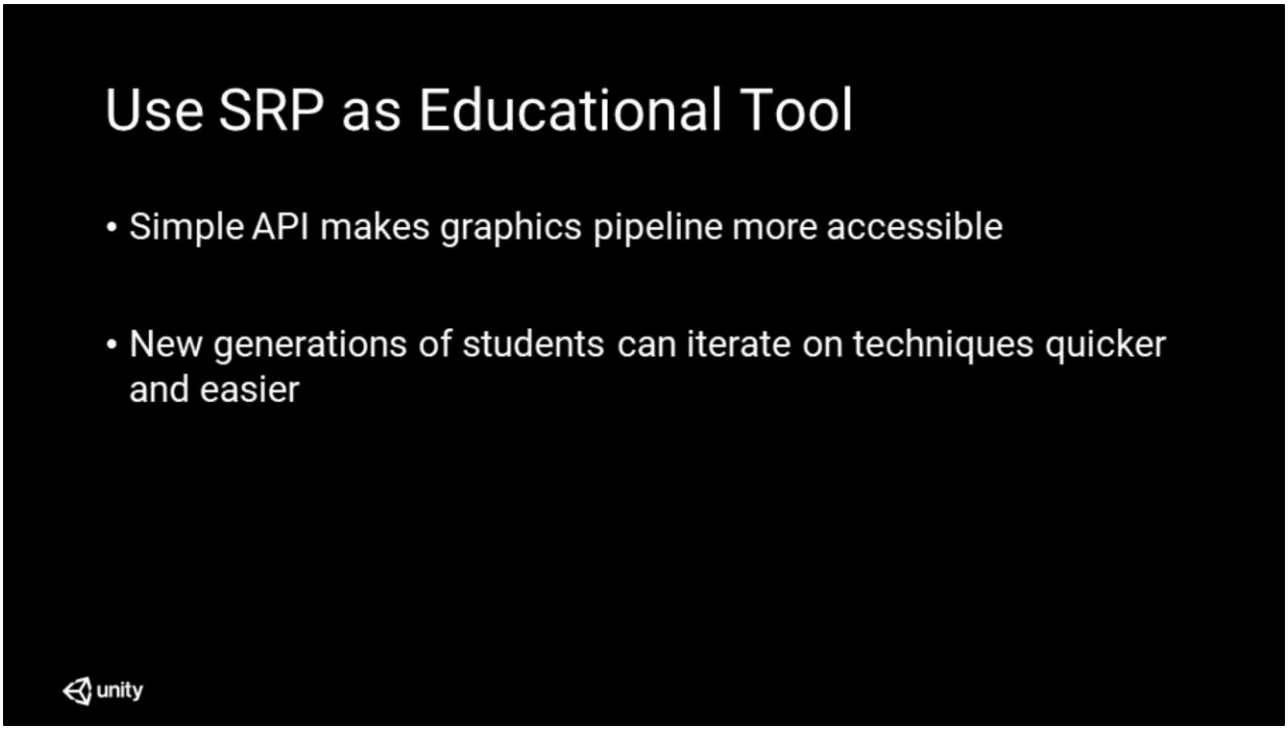
Allow Easy industry-wide sharing: Unity PE is accessible for all, scripts can be shared as content – just share your Unity projects.

Made reproducing related work a slam dunk!

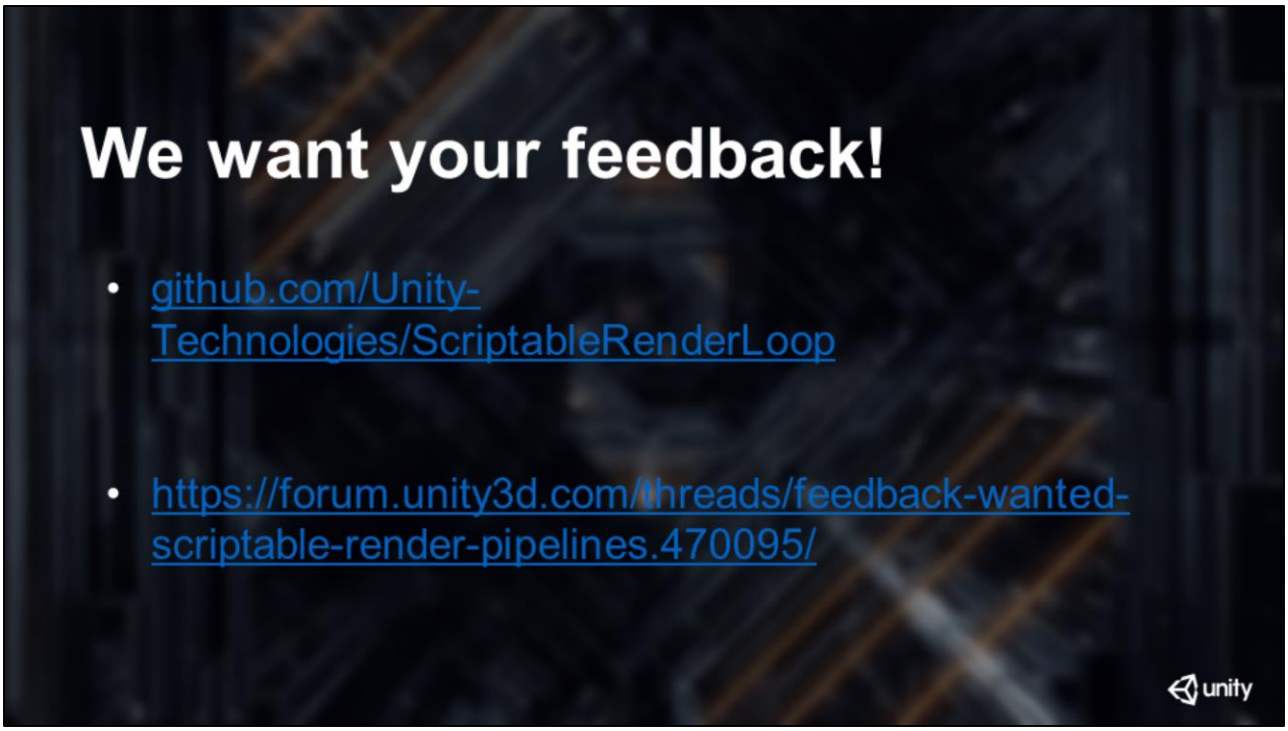
We will be sharing our SRPs code

We will be sharing content scenes

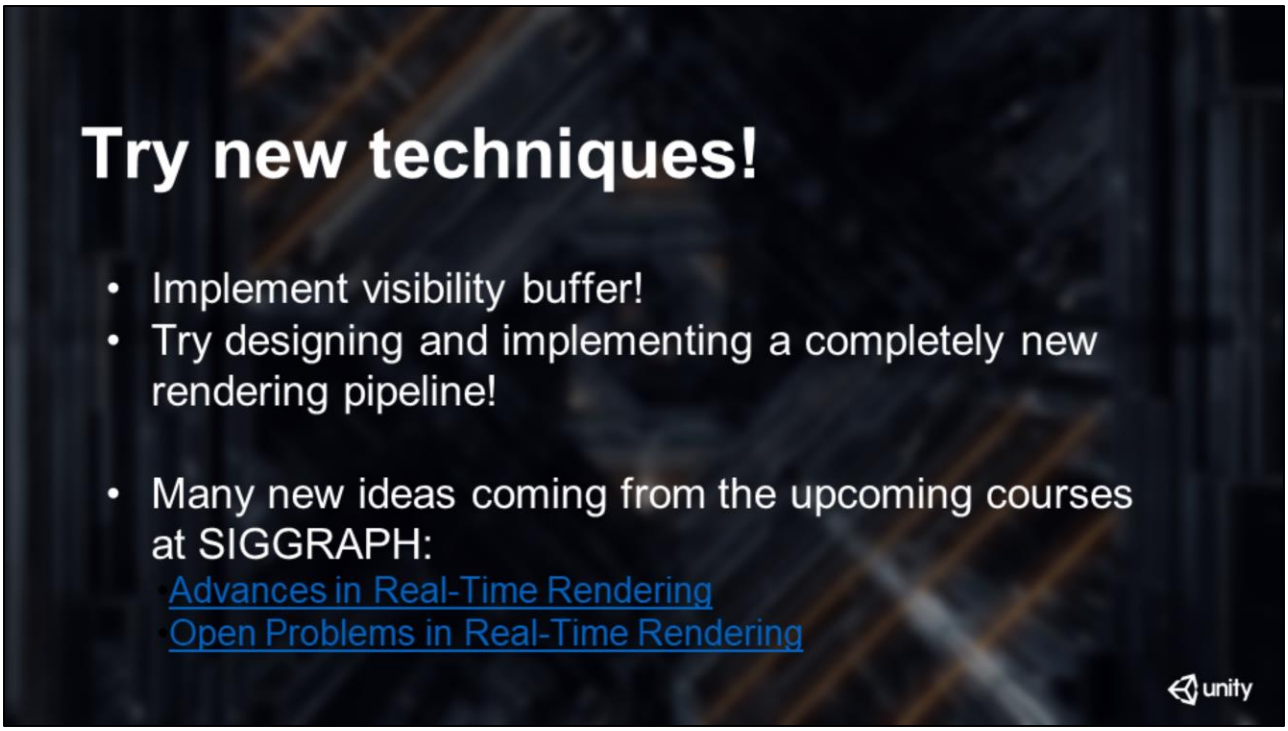
We hope this makes it a great framework for developing novel techniques!



I also really invite you to use SRP as an educational tool for graphics. The architecture makes graphics pipeline far more accessible
We can already see example of Unity devs with little to no graphics programming experience that learned to use the SRP API easily, especially with its quick script feedback look.



But most importantly, we are excited to hear your feedback!
Is this interesting to research? Do you want to use this architecture, and if yes, what excites about this API and what you wish we'd add, what's missing?
For instance, some devs in the forum targeting iPhone 5S wanted to set keywords per amount of lights (say SHADE_X_LIGHTS). We are now thinking about adding filter support to return visible renderers by the amount of lights. I know there will be a lot of clever people in the audience that will think about some cases we might have not. Please let us know!
Here is the github for the project and Here is the forum thread for providing feedback



And most importantly – try it! Develop new methods! Implement visibility buffers in SRP! Try to create a crazy new rendering pipeline! I really hope to see many of the new ideas and experimentation coming from the upcoming SIGGRAPH talks be done in this architecture easily. We will all learn from that!

Acknowledgements

- Unity graphics team!
- Tim Cooper, Sebastien Lagarde, Felipe Lira, Peter Bastian, Morten Mikkelsen
- Tim Foley
- Peter-Pike Sloan

unity

References

- [TatarchukWang14] Tatarchuk, N., Wang, S.-K. Creating Content to Drive *Destiny*'s Investment Game: One Solution to Rule Them All, SIGGRAPH 2014
- [Tatarchuk15a] Tatarchuk, N. Applied Graphics for Video Games, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, February 2015
- [Tatarchuk15] Tatarchuk, N. Destiny's Multi-threaded Renderer Architecture, Game Developer Conference, March 2015

References

- [Andersson2015] Andersson, J. **The Rendering Pipeline - Challenges & Next Steps**, *Open Problems in Real-Time Rendering* course, SIGGRAPH 2015
- [Foley16] Foley, T. **A Modern Programming Language for Real-Time Graphics. What is Needed?**, *Open Problems in Real-Time Rendering* course, SIGGRAPH 2016
- [O'Donnell2017], O'Donnell, Y. FrameGraph: Extensible Rendering Architecture in Frostbite, Game Developer Conference, February 2017
- [Ante17] Ante, J. C# Job System and Compiler, Unite Europe 2017
- [Lauritzen17] Lauritzen, A. **Future Directions for Compute-for-Graphics**, *Open Problems in Real-Time Rendering* course, SIGGRAPH 2017


Unity is hiring
careers.unity.com


Questions?


Thank You!
unity