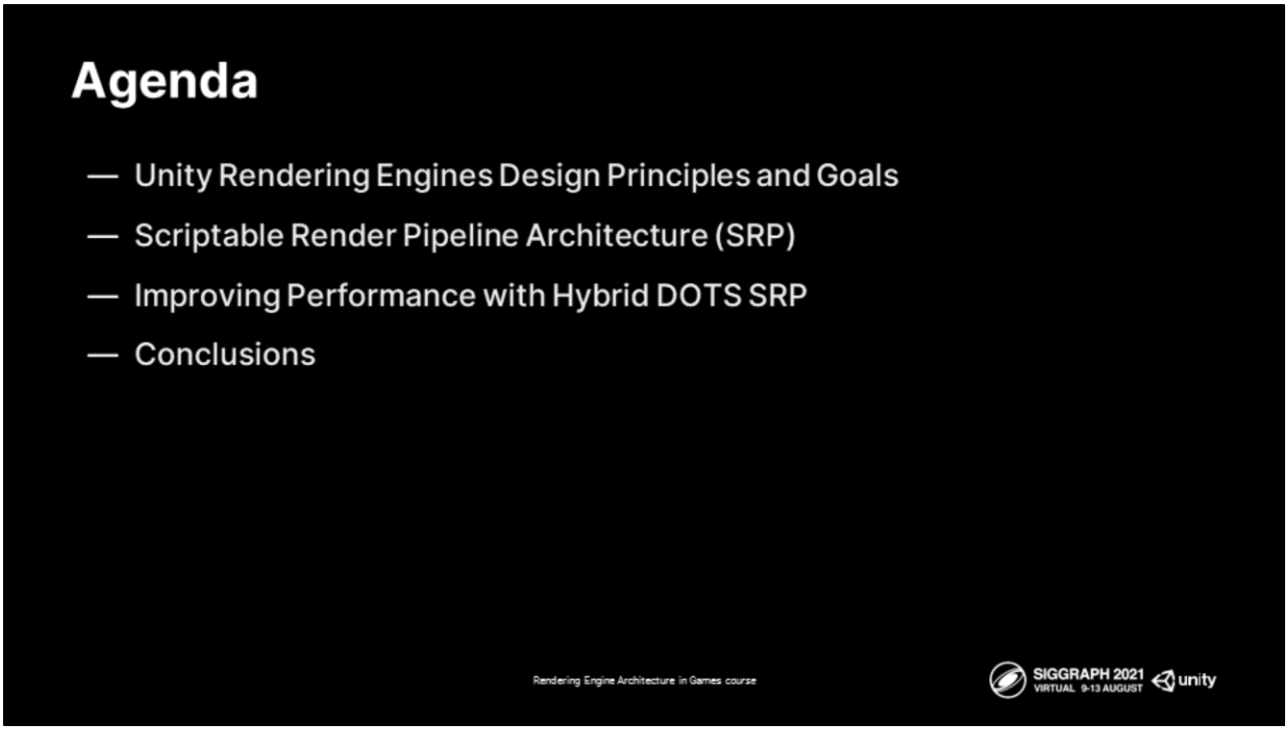
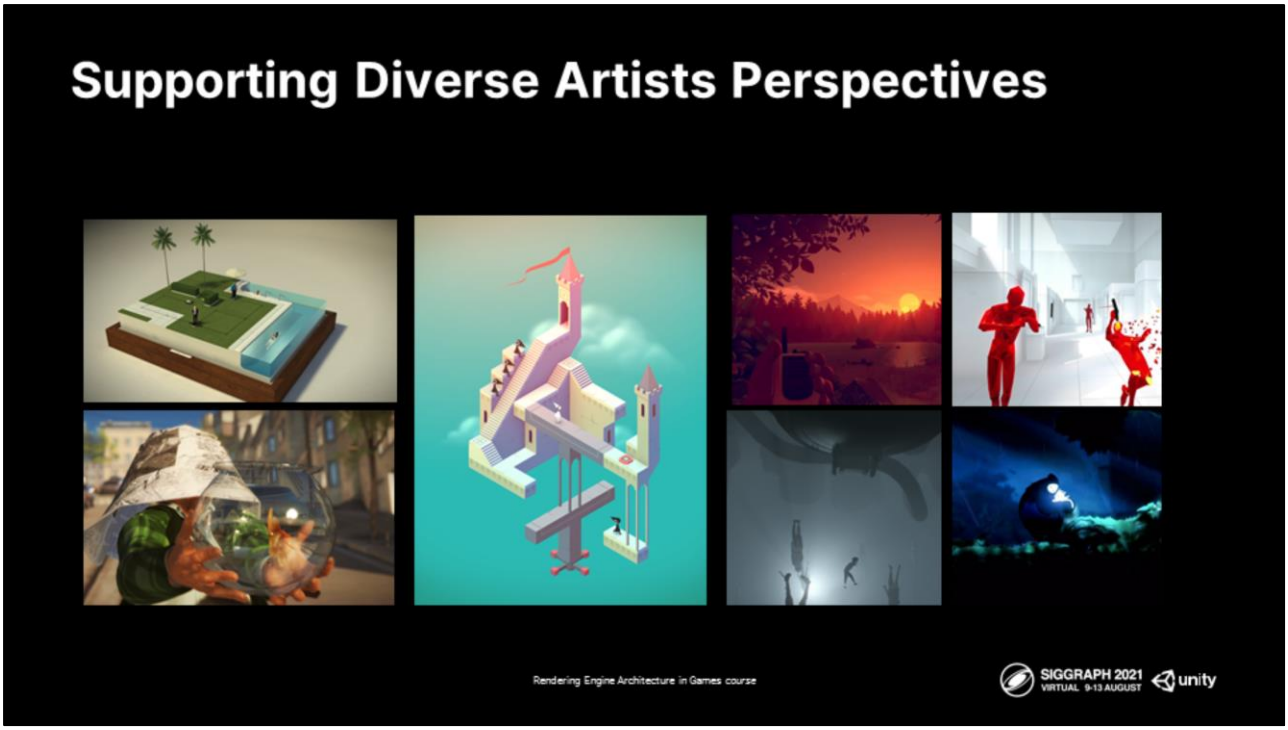
Hi! I'm Natalya Tatarchuk, and recently I have switched gears to lead our new Graphics Innovations group. Prior to that I've led Unity's graphics group. Today, we're going to talk to you about Unity's rendering architecture and its evolution.



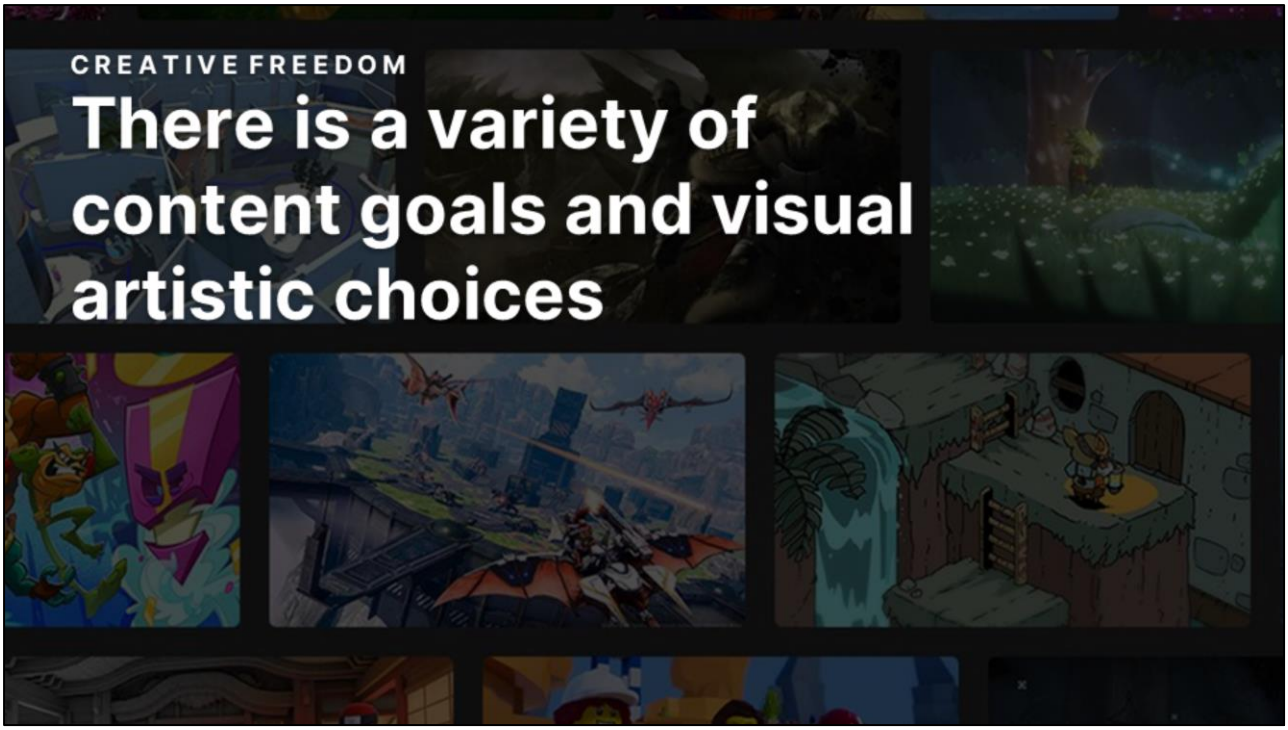
By understanding what are our engine's goals we can help ground our architecture's design principles in the context of solving the right problems, so we will go through that first. Then we'll deep-dive into the Scriptable render pipeline architecture (or SRP), and how we are evolving it toward higher throughput with the Hybrid DOTS SRP Architecture, where we extend the SRP architecture to operate on both Unity's current game object runtime as well as the new data oriented tech stack (DOTS).

Let's look at **what** our engine is trying to do and **why** first..
In order to understand the choices behind the architecture design we need to start from framing the context of what Unity is trying to enable.

At its heart, Unity is about **empowering creators**. But what kind of creators and creations are we trying to empower? What does this actually mean, for our architecture?



Artists have created visually and emotionally stunning pieces with Unity for a decade and a half, ...

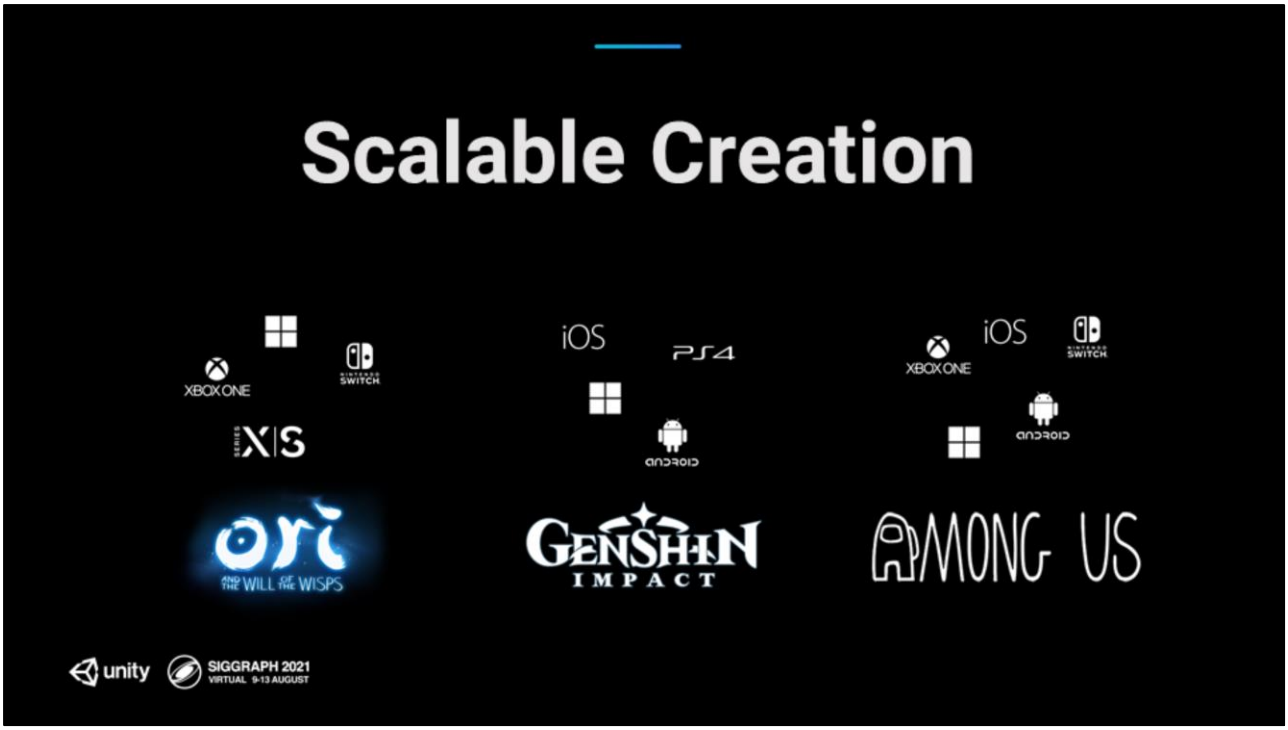


.. because the engine gives them **freedom of expression** - in terms of creativity, style, visuals. And thus, one of the major goals for our rendering architecture …

… is to give the creators maximum freedom for their creative vision. You could make something 2D, 3D, cell shaded, realistic, cartoon, it is a long list. It's important to us to ensure that developers have the tools to be flexible with the types of content they want to create. **There is a huge variety of creations being done with our engine. And we want to render it all, effectively, performantly, on all the right platforms.**

Most people create entertainment so that it can be experienced by someone, can be shared. reaching a wide audience matters, both to share the stories, but also - pragmatically - to be successful.

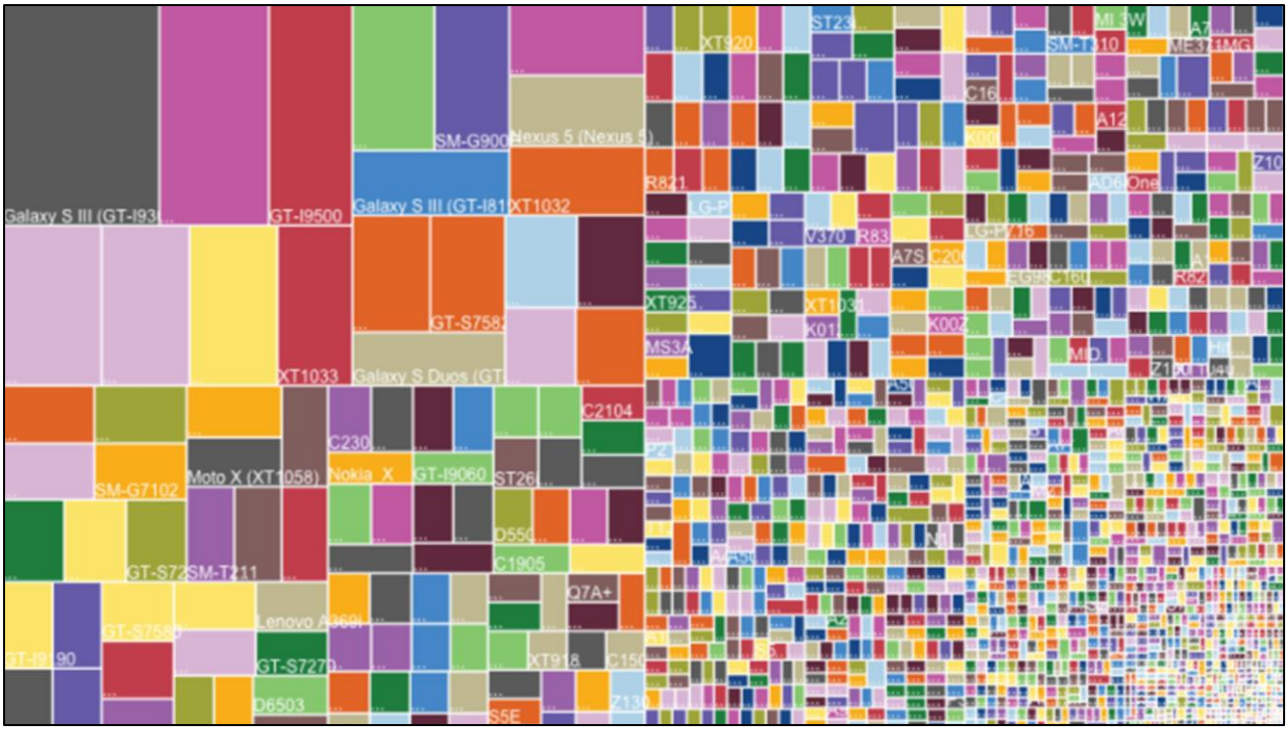
To help reach the widest audience, we support over 20 platforms, with the goal of making it as easy as possible to target multiple platforms quickly. Of course for us this means that our techniques must work on all these platforms in some way. No small feat.



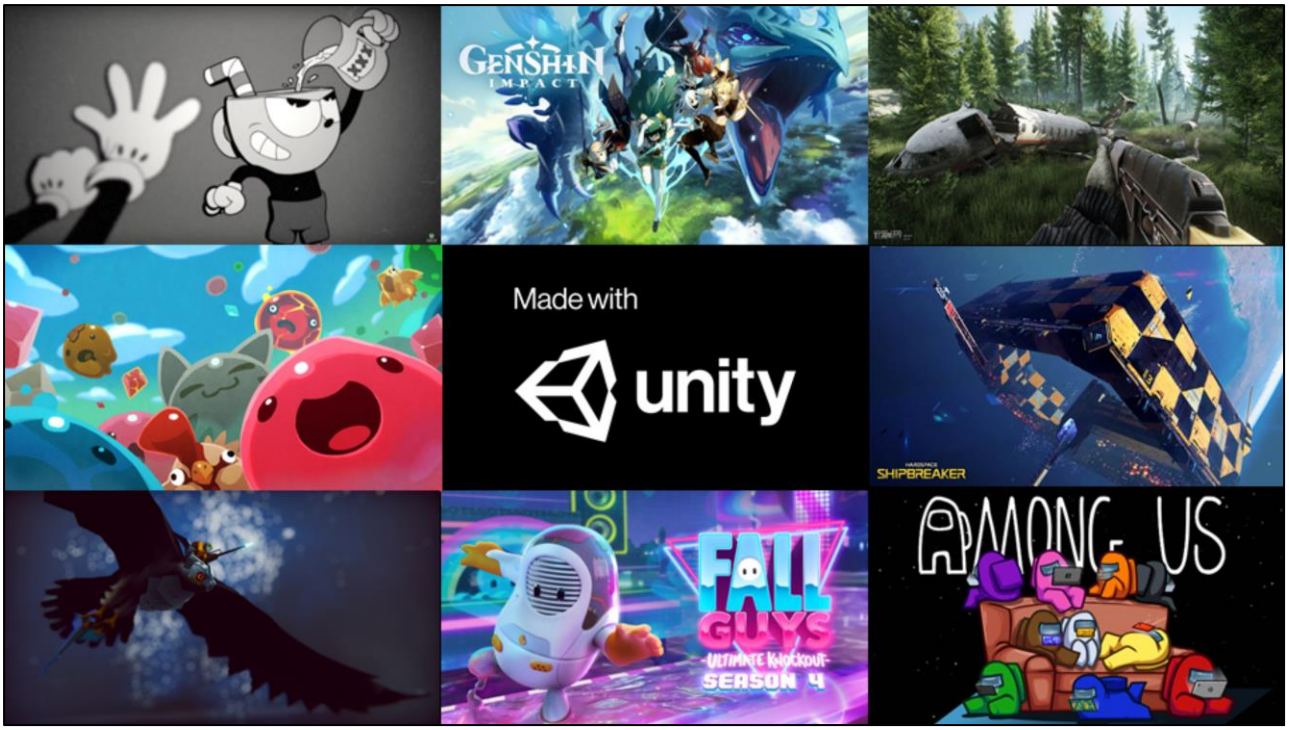
We are finding now is to be more successful, games are aiming for a wider variety of platforms at the same time.

It’s important to us that creators have the ability to make a project that can run *performantly* from low-end to high-end. This means that the tools in our toolbox need to either scale themselves or have steps that makes sense at different hardware tiers and types.

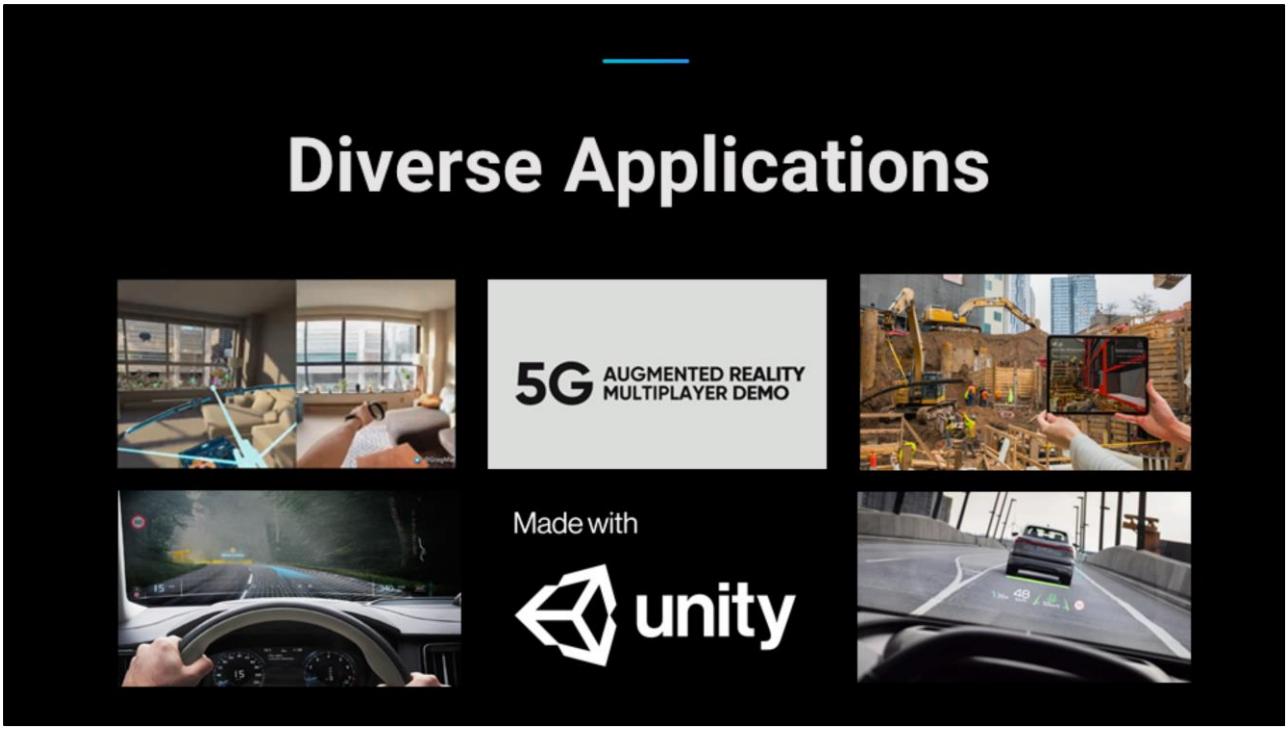
The interesting thing is that even if you slice through one tier of hardware - it’s a deeply fragmented world. Zoom in on Android and ..



… and this is the landscape - thousands of mobile devices, which vary in hardware capabilities, as well as APIs. It’s complex to test, optimize and target your game to these devices. Our goal is to make it as easy as possible (architecturally and in terms of content creation) for the creator to do, without having to burden with all of this complexity directly.



We talked about games thus far. And it's easy to keep thinking of Unity as purely a game engine. Certainly that's where we are rooted, yet we're not building a game engine *only*, but …



… a larger, generalized framework for applications. We support 10KB applications (such as instant games or embedded apps running on the latest fridges) to mobile, console and PC games to AR | VR apps and to hundreds-of-terabytes cloud-streaming industrial and automotive solutions.

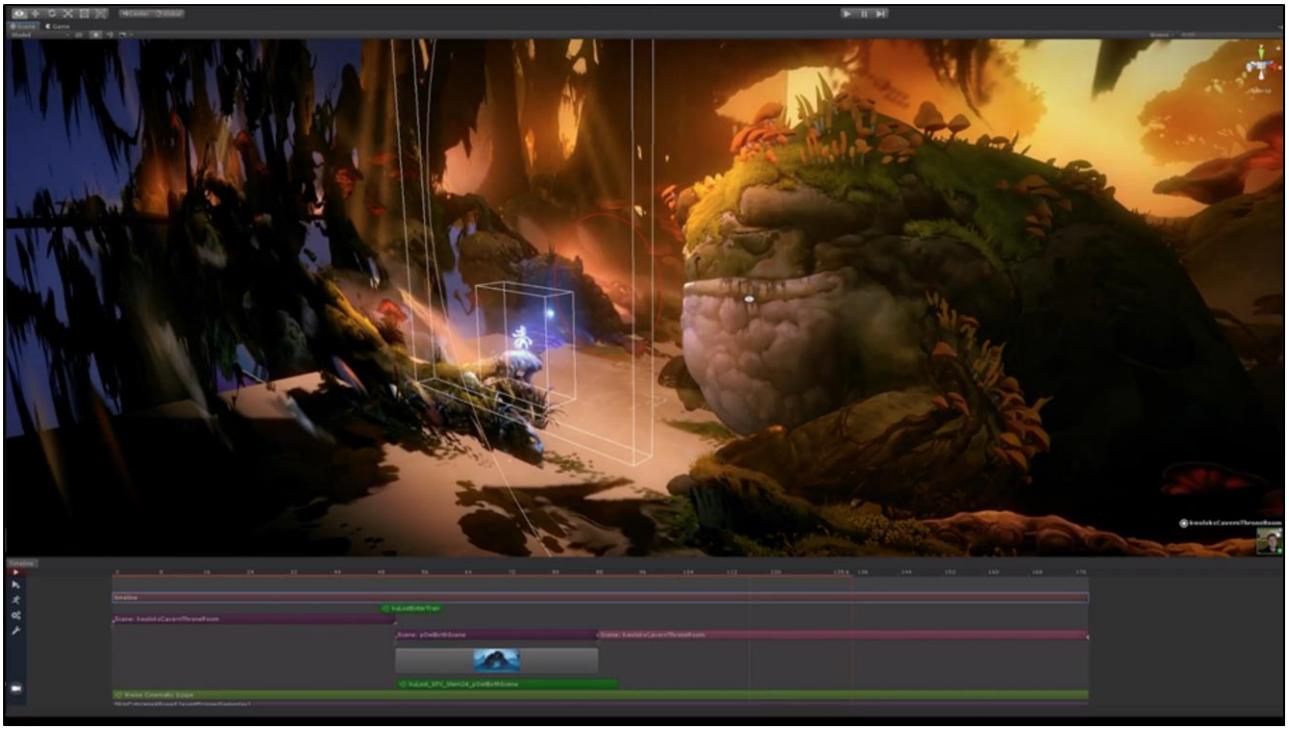
All these use cases come with their constraints and architecture demands.

For example, with *Unity Forma*, we need to have the same runtime-customizable, high-fidelity design asset run smoothly on the high-end platforms as well **<click>** on the smartphones at full quality and interactivity.

Another example is **Unity Reflect** product, which ingests large industrial BIM data directly in the player on the client. We have to performantly render this huge data set (up to hundreds of terabytes) with no pre-baking (can't build Umbra tomes for visibility culling, for example) at interactive rate on our full range of platforms.

We want very low friction from idea to implementation for developers on our architecture, with all the necessary tools provided by default but allow for quick and easy tuning as needed, whether a simple surface-level change or a deep surgery.

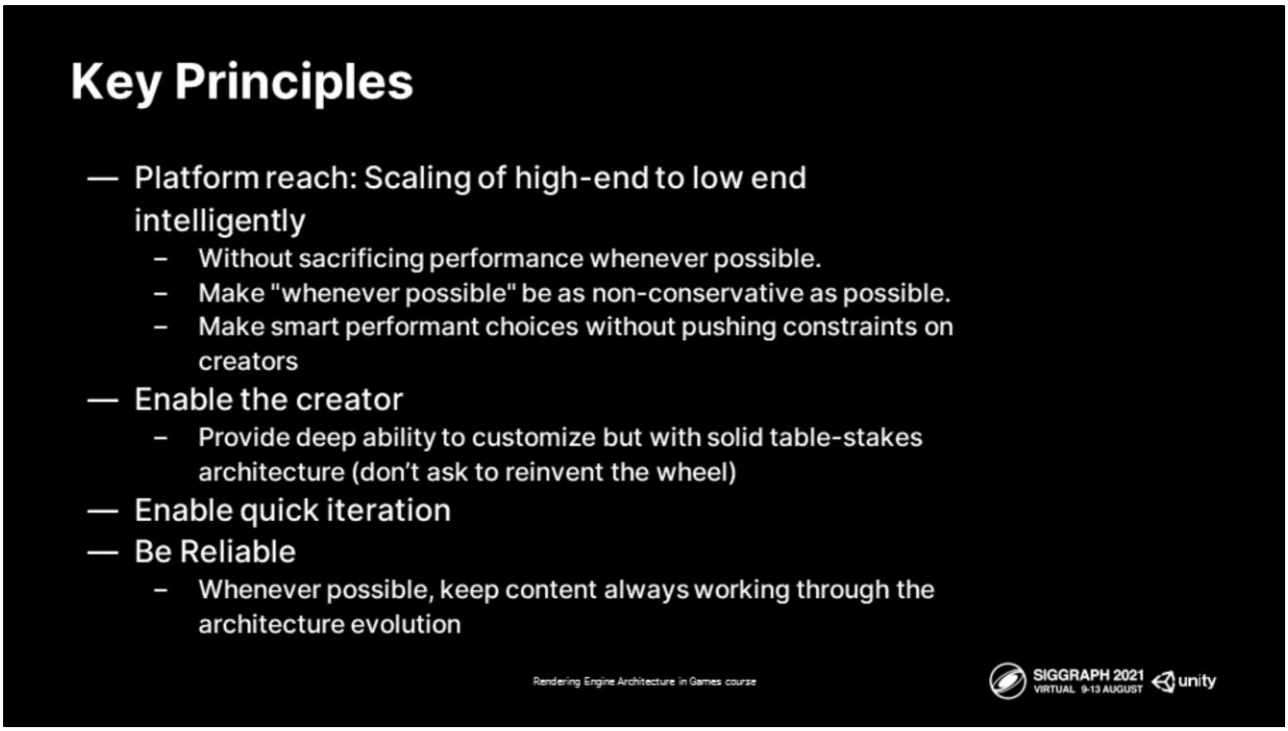
For example, the creators of Ori and the Will of the Wisps, a breathtakingly beautiful game, ...



… had such different needs for the look of the game, that they created their custom rendering pipeline in SRP. This would be very challenging without the flexibility of SRP's architecture, though we are still in the process of making this easier going forward.

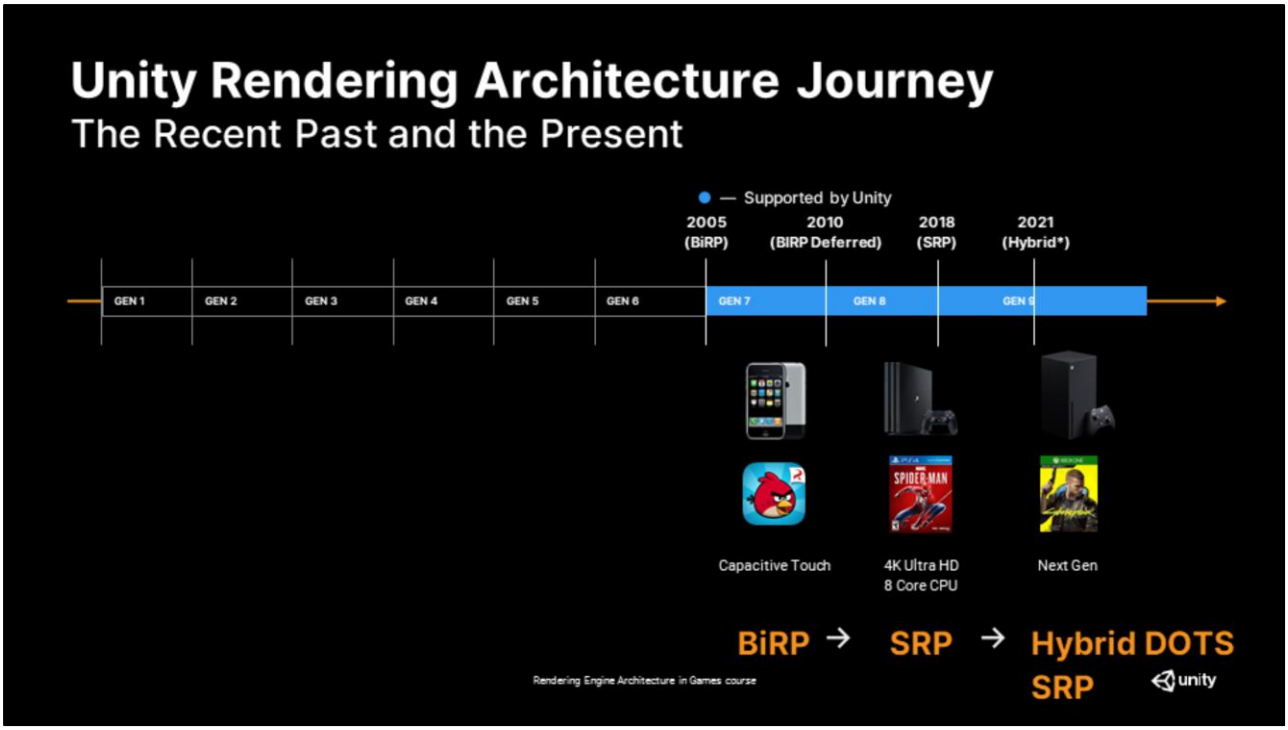
Customizability can often mean “easy to change”. But we need to think about extensibility as “build on top, while providing a stable base”. And for us this especially comes into play when thinking about …

… our asset store ecosystem, with many thousands of assets and plugins that creators develop on top of our engine. Sustaining this ecosystem is an important requirement for our architecture.



Extracting from these engine goals, then the key principles for our architecture then are: **Platform reach**, **Deep flexibility and customization**, **Performance across the gamut of platforms, quick iteration** to unshackle the creativity (both for developer and creator), and **continuity:** keep existing content working or upgrade it smoothly.

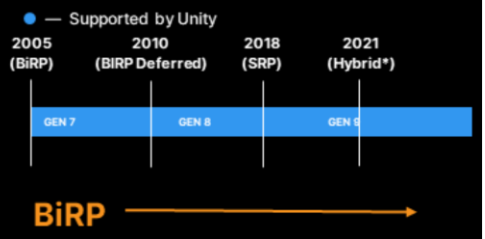
next let’s look at how our architecture evolved to answer these goals.



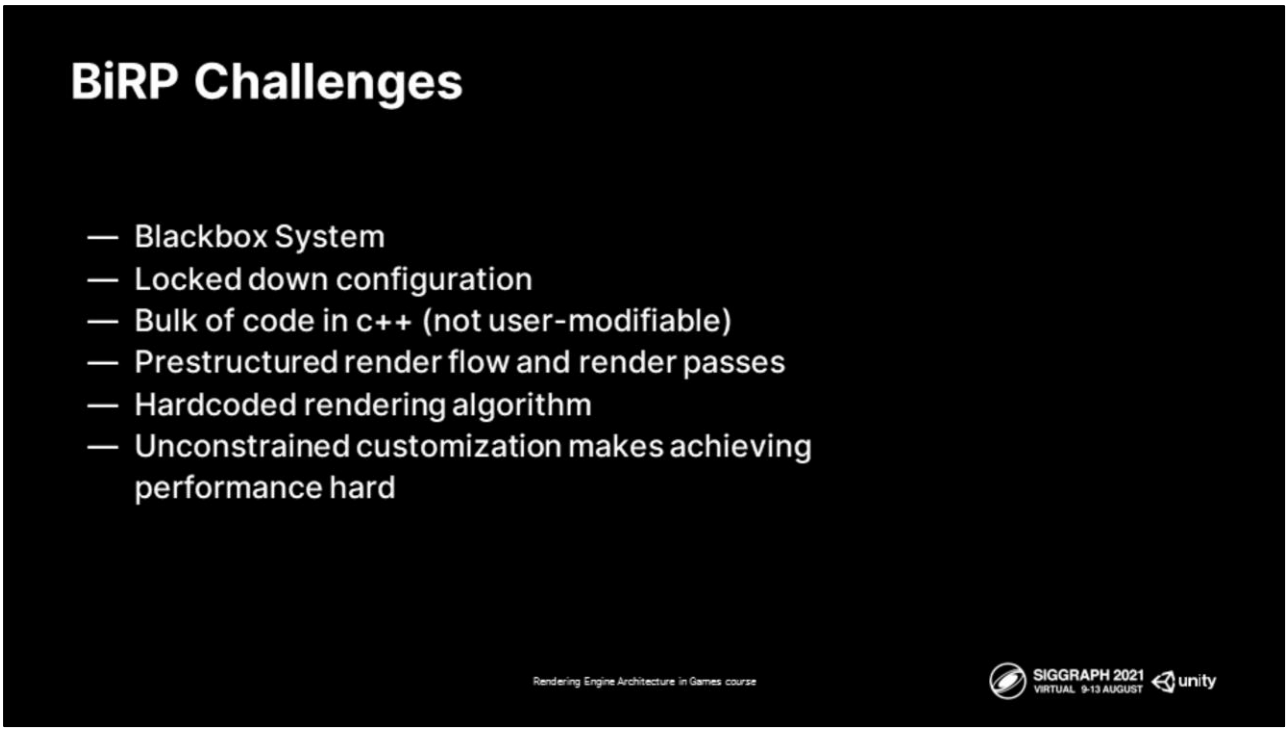
We started with the **Built-in Render Pipeline (BiRP)**, moving to the **scriptable render pipeline (SRP)** architecture, and, our latest evolution, the **hybrid DOTS SRP (**note: the hybrid architecture does not support built-in, as built-in only supports the game object runtime).

Built-In Render Pipeline (BiRP)

- Unity's single rendering pipeline prior to SRP
- Targets mobile to console | PC | AR | VR
- Supports forward | deferred rendering and decent variety of features
- Convenience: One-stop shop for rendering
- Turnkey solution for all platforms

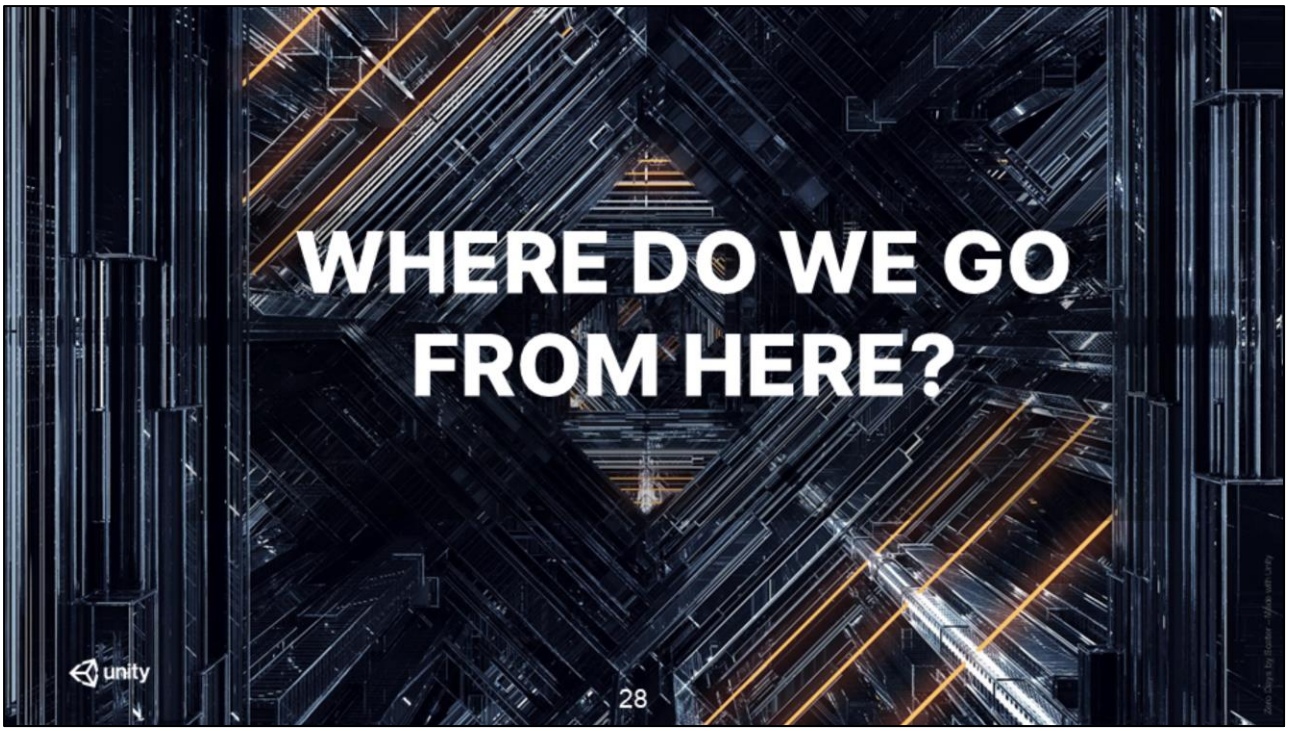


Built-in render pipeline was our one-stop-shop turnkey solution for all supported platforms, with forward & deferred, and a solid mix of graphics features. Users found it very convenient and easy to use. Yet it was simple to run into major challenges with this architecture.

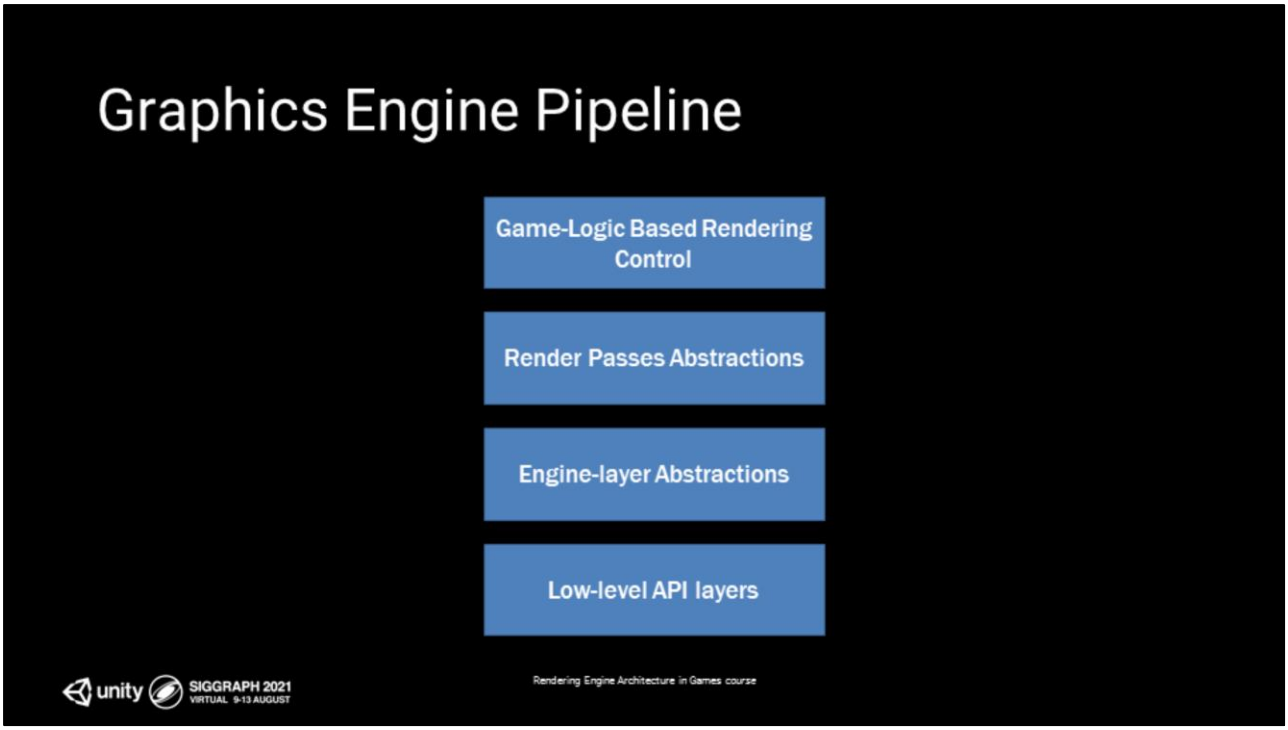


With its blackbox system, the configurations were hard-coded for the hardware in the C++ land, and not easily modifiable.
And while BiRP offered a turn-key solution, it also locked the rendering algorithm in a way that often didn't provide best performance for a given platform, it yielded sort of lowest common denominator, to keep up with the cross-compatibility needs. And with each addition of new platforms or hardware stages or APIs, the cost of modifying this architecture continued to increase exponentially, making it fragile and hard to maintain.
It's flexibility was also a challenge. It exposed a large amount of user-land callbacks allowing changes or injection of state state at any point in the frame dynamically, by calling to C# (do you want to switch your object from rigid to skinned? Go for it. Do you want to change decal blending mode late in the frame? Go ahead), which made it very challenging to cache data effectively and to manage persistence state intelligently. This was at the heart of many performance issues for this architecture.

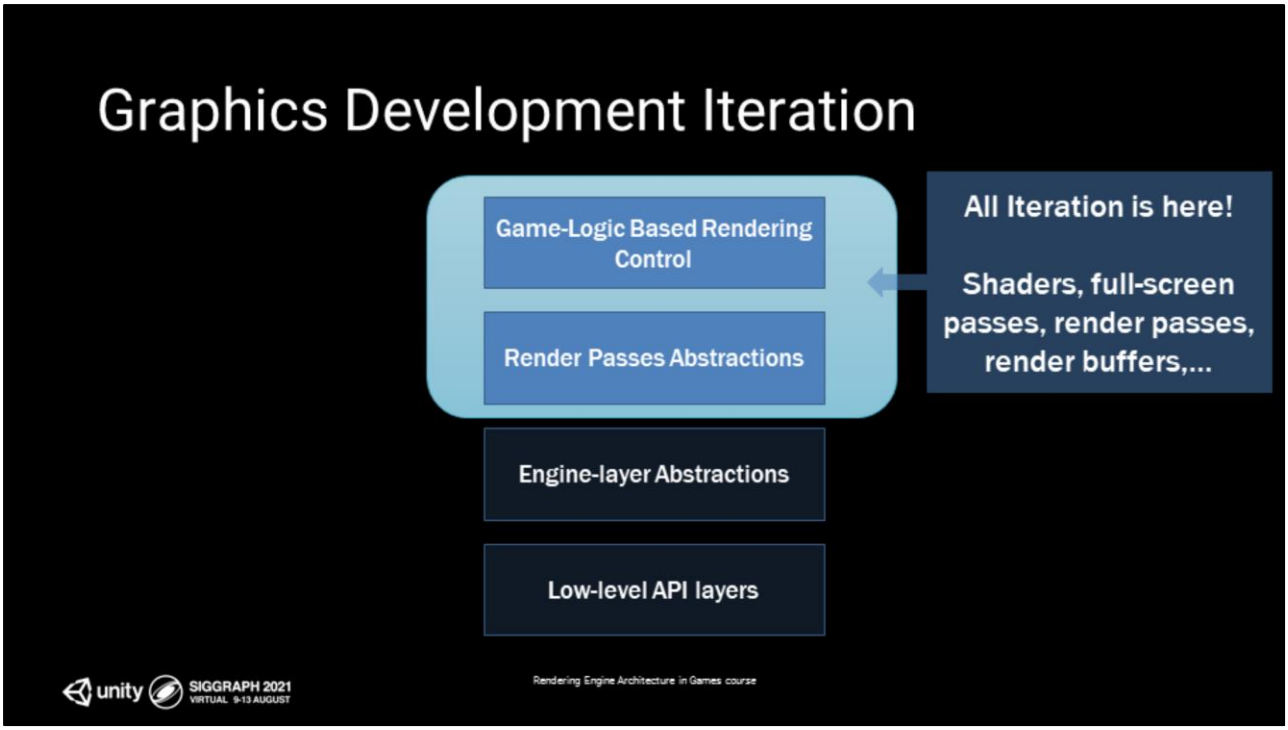
Yet - while it's tempting to scream "Do away with this with fire!!!", it's also incredibly important to realize how much this flexibility offered the very advantage of Unity - the lack of shackles, which really enabled powerful prototyping and creativity. These hard-to-optimize-for choices might seem very bad, but they also enabled Unity to be an amazing place of creativity - and our goal for the architecture evolution is to learn from this and preserve as much of the power and the freedom as possible while also finding new ways to better understand creator's intent and smartly convert it to performant runtime. That's the really hard problem to solve, but also the most interesting one.



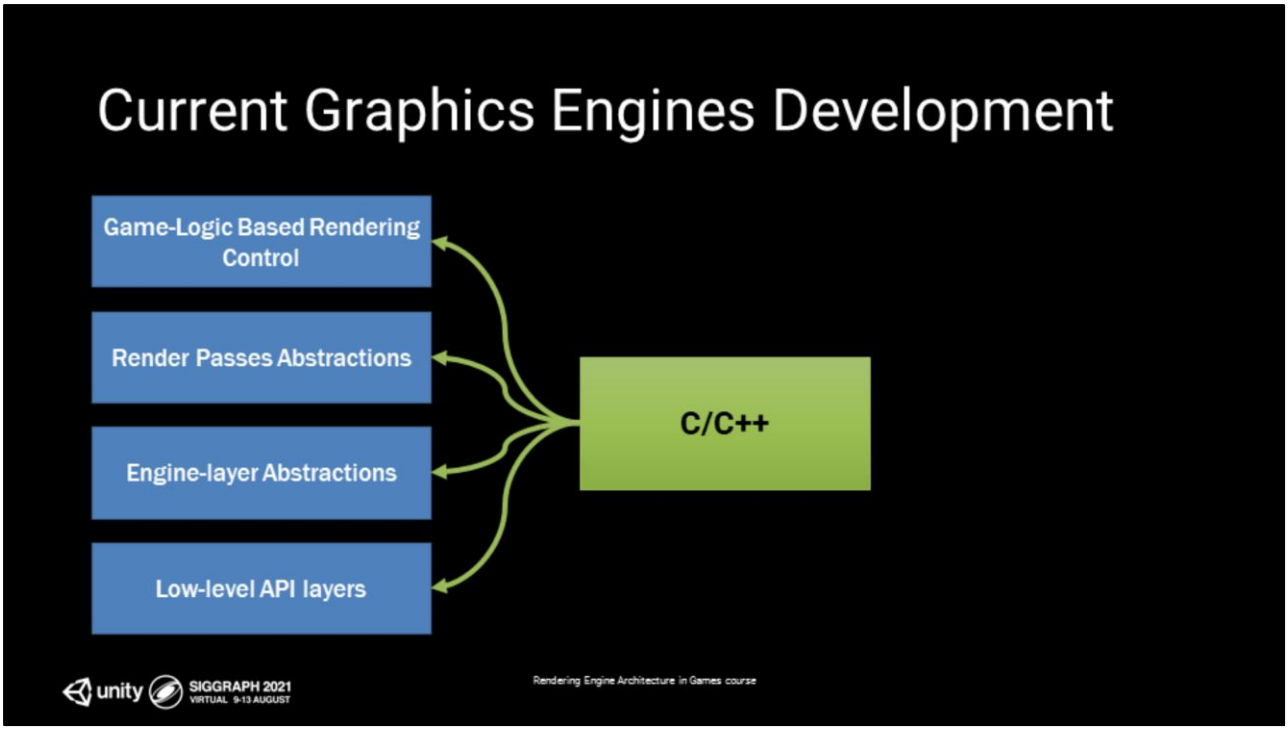
So it’s clear that BiRP wasn’t going to be fit our users’ needs. How did we evolve forward?



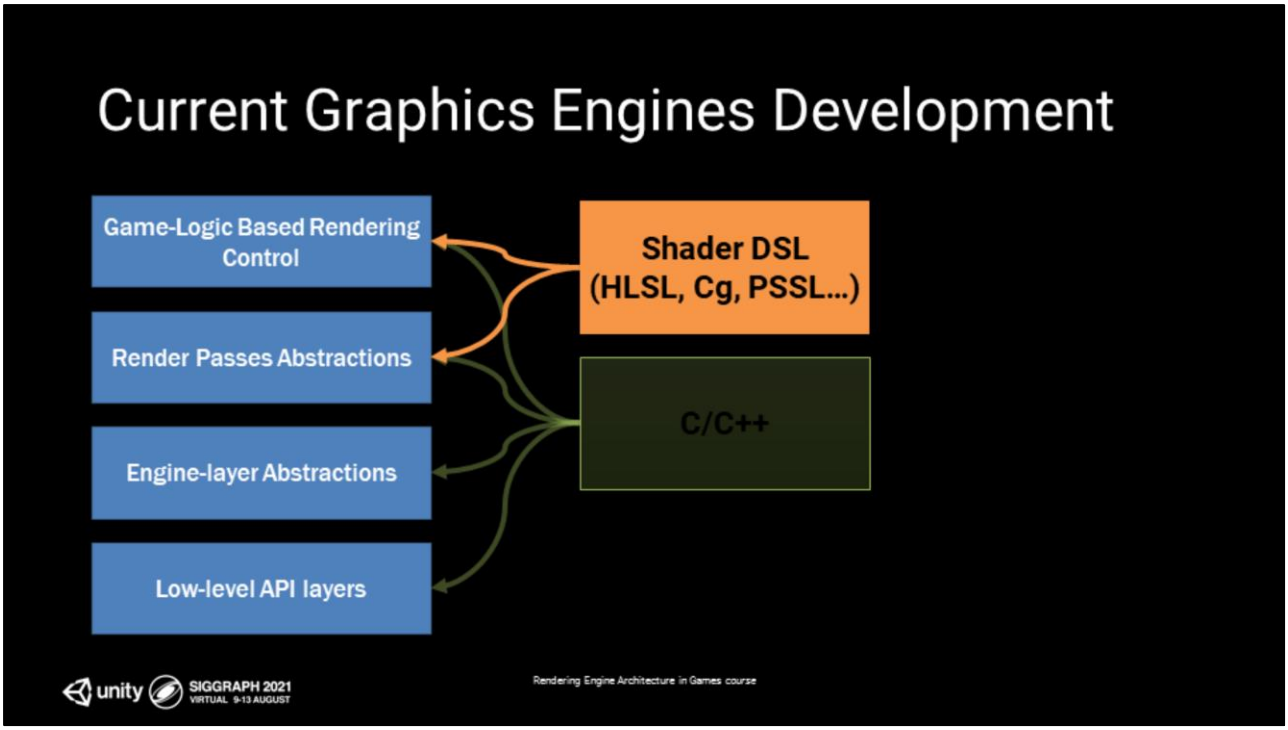
Let’s say we wish to write new physically-based material model with changes to the material shaders and G-buffer layout plus lighting in our engine. How does that flow through the engine layers?



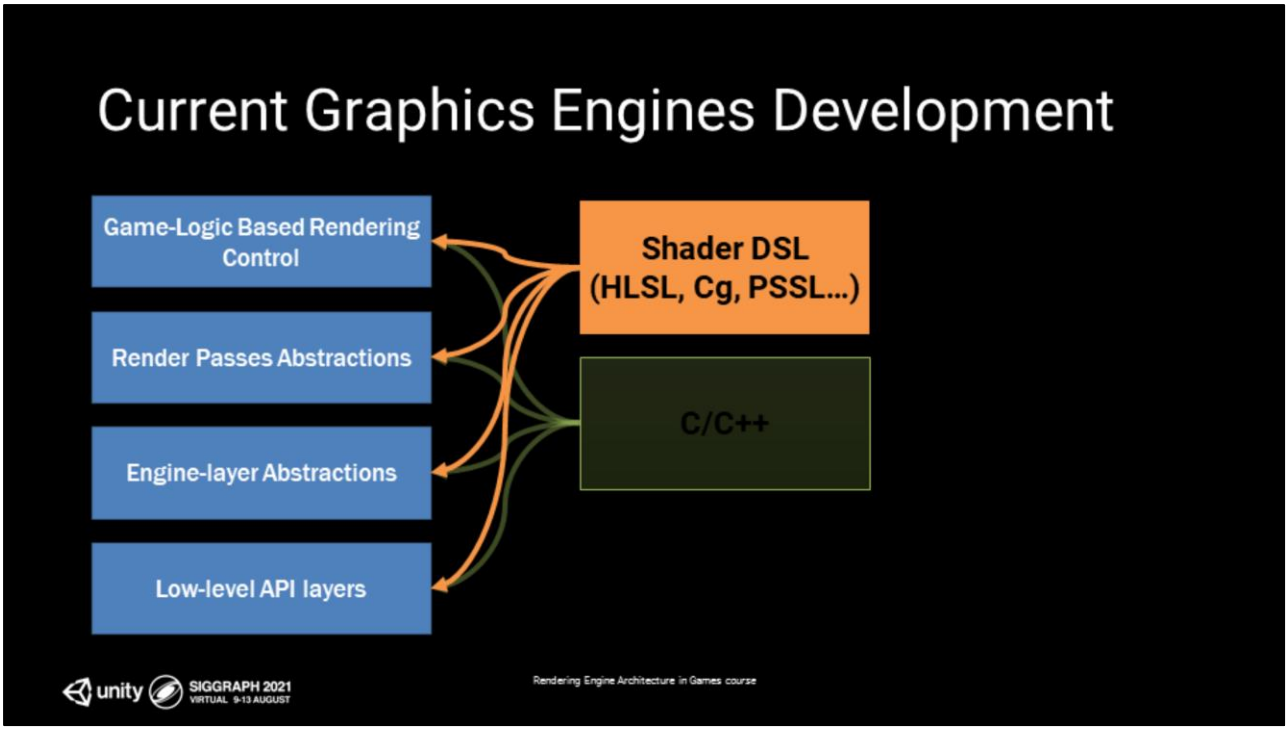
We probably need to iterate on the shader passes, render textures layout, etc. The heart and soul of the algorithm – and that's where we want to spend our time, not the boiler-plate code in the layers below.



Typically, when we're working the game engine systems, we're operating in c++



For graphics, we frequently do work using shader domain-specific languages (whether it be a platform-specific shader language like HLSL, Cg, PSSL, or a custom DSL), which mostly deal with the higher elements of this stack.



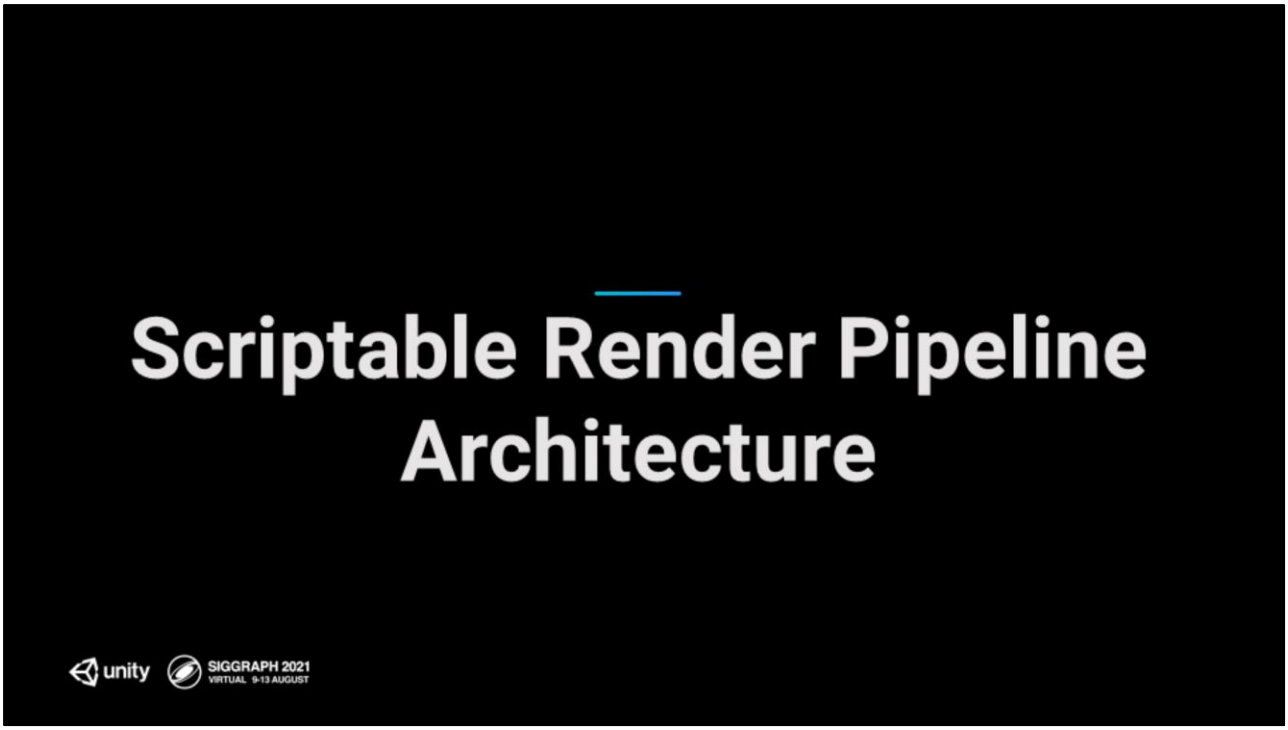
Though occasionally, when shaders have very specific hardware dependency (VRS, instancing, tessellation, async compute), they need to reach to the engine or low-level API layers and may require deeper changes, in line with the corresponding shader changes. This is an infrequent operation, mostly when bringing up new hardware or API.

And in C++, we iterate via build+link | continue, a familiar experience.

While distributed build helps with the iteration time, we are still limited by the serialized link step pain, and as the engine size increases the cost of that step grows proportionally. And this workflow of rebuilding the executable also means the inconvenience of having to reload my game level, restarting the game play, all in an effort to get to the same place in the frame I was trying to debug in the first play. That's tedious.

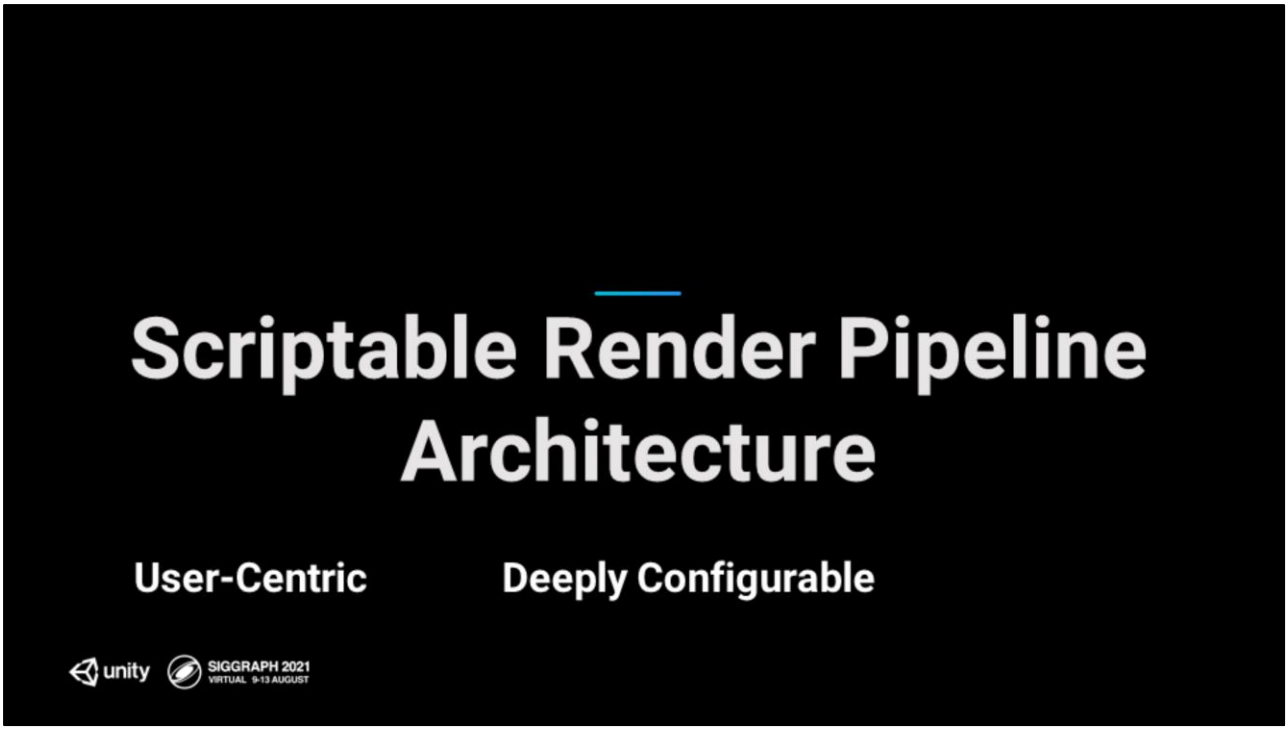
What if there is a different way to iterate for a graphics developer?

We can see that to keep up with the platforms, use cases variety and product needs, a modern *third-party* game engine needs to be highly configurable, react dynamically to varied throughput, and make intelligent choices about platform constraints. But we need to rethink how we design it. How can the previous observations help us find a philosophical shift for our design?

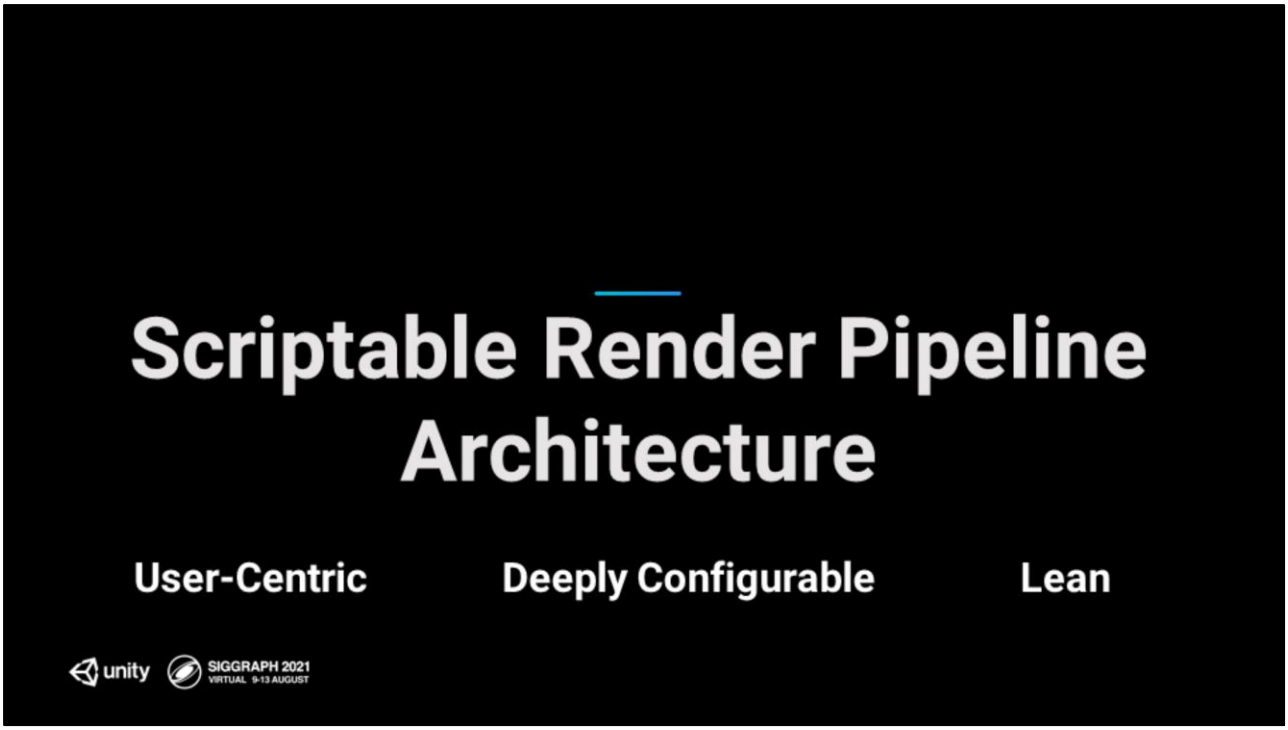


Learning from the challenges of BiRP and wishing to have a better iteration experience than the C++ land, we designed the SRP architecture to be ...

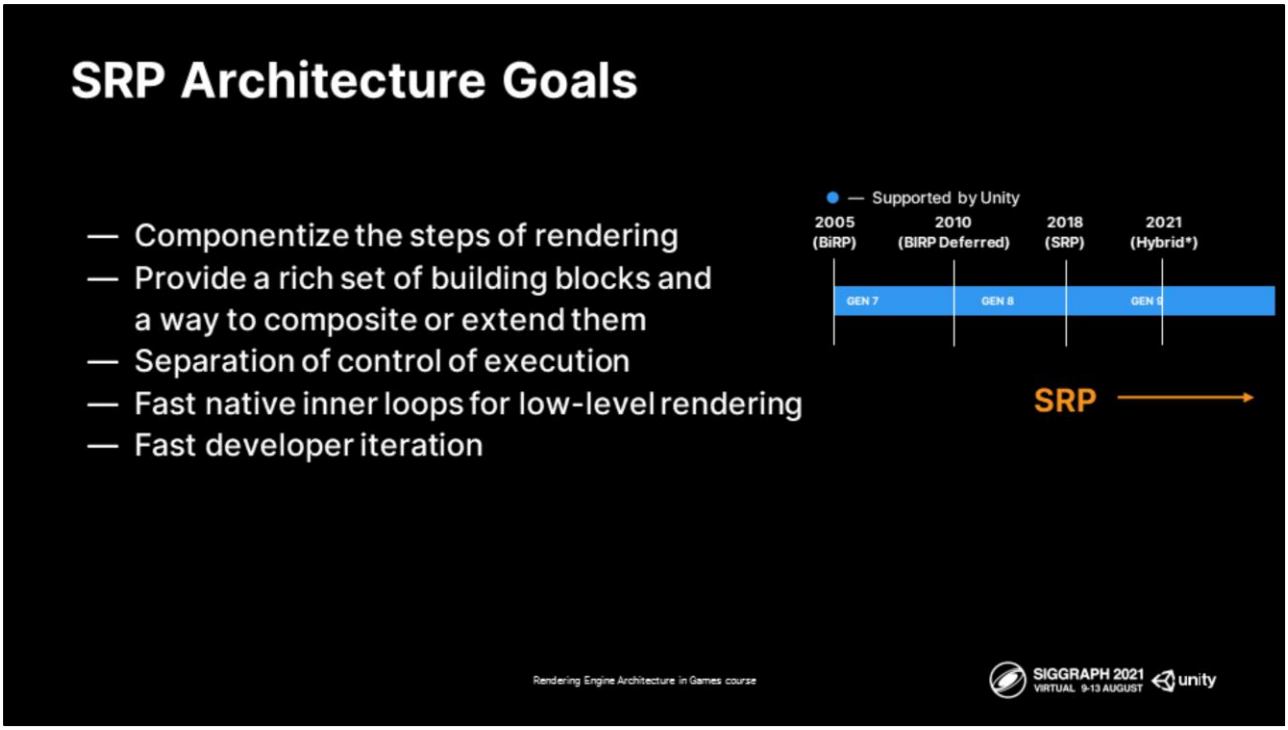
.. **User centric**: It controls rendering from C#, easy to modify and debug. The engine executable makes no assumptions about the underlying rendering pipelines: forward vs. deferred, tiled or not tiled, ray tracing or rasterized, g-buffer layout or even whether or not there is a g-buffer. **These are all explicit pipeline design choices at C# and shader level.**



It is highly **Configurable,** with an **inspectable**, open **API** so you can write your own renderer if you want to do that or extend as needed.



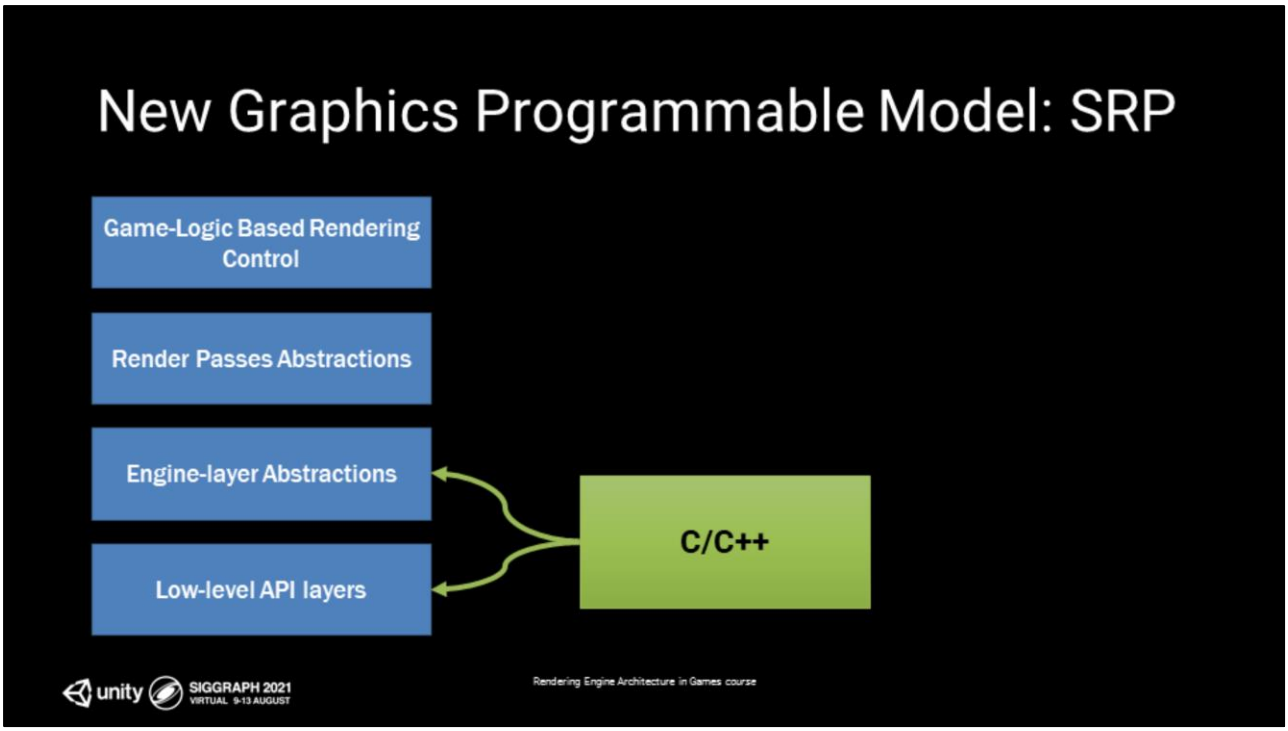
It is **Lean**: Take only what you want / need for your platform or project in mind. Of course, since our aim is to create amazing games or real-time experiences, **performance of our architecture is critical**. No amount of flexibility and generalization will make that a non-requirement for us.



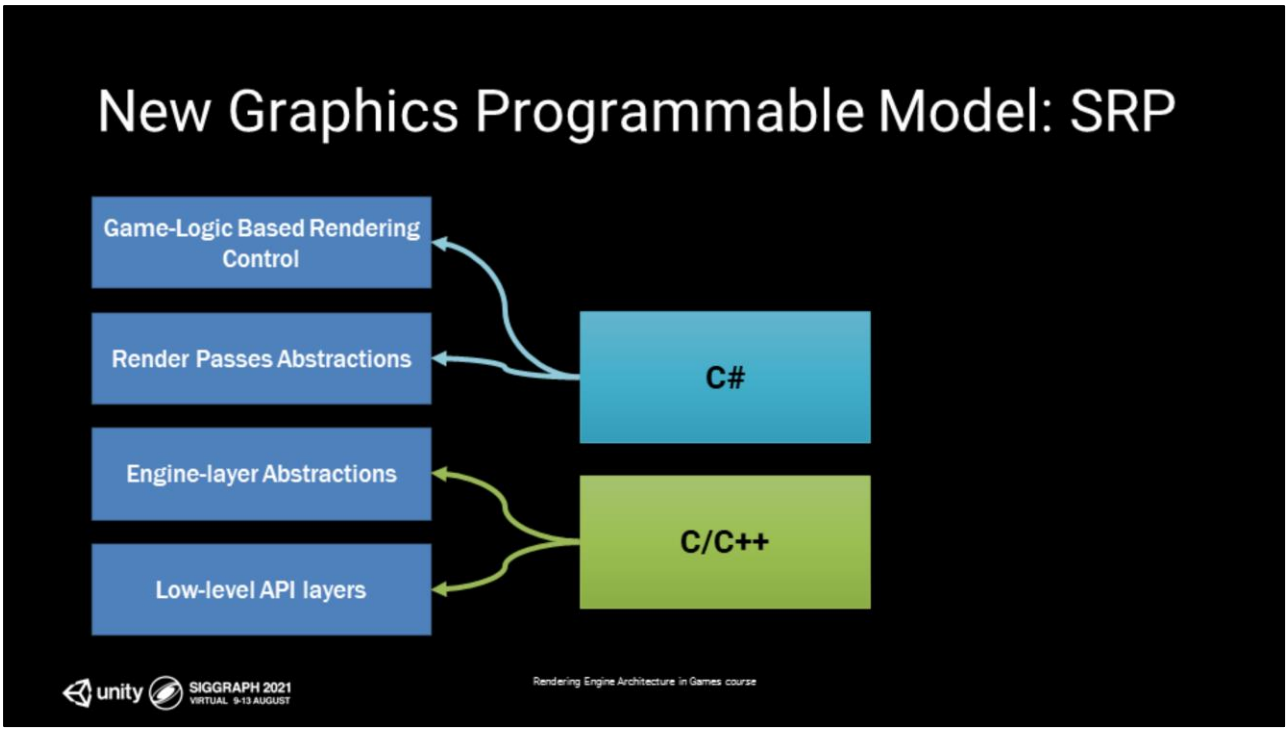
The technical goals for the architecture are:

1. Componentize the steps of rendering via a concepts of building blocks you can put together, customize or extend
2. Separate control of rendering order from the high throughput inner loops execution

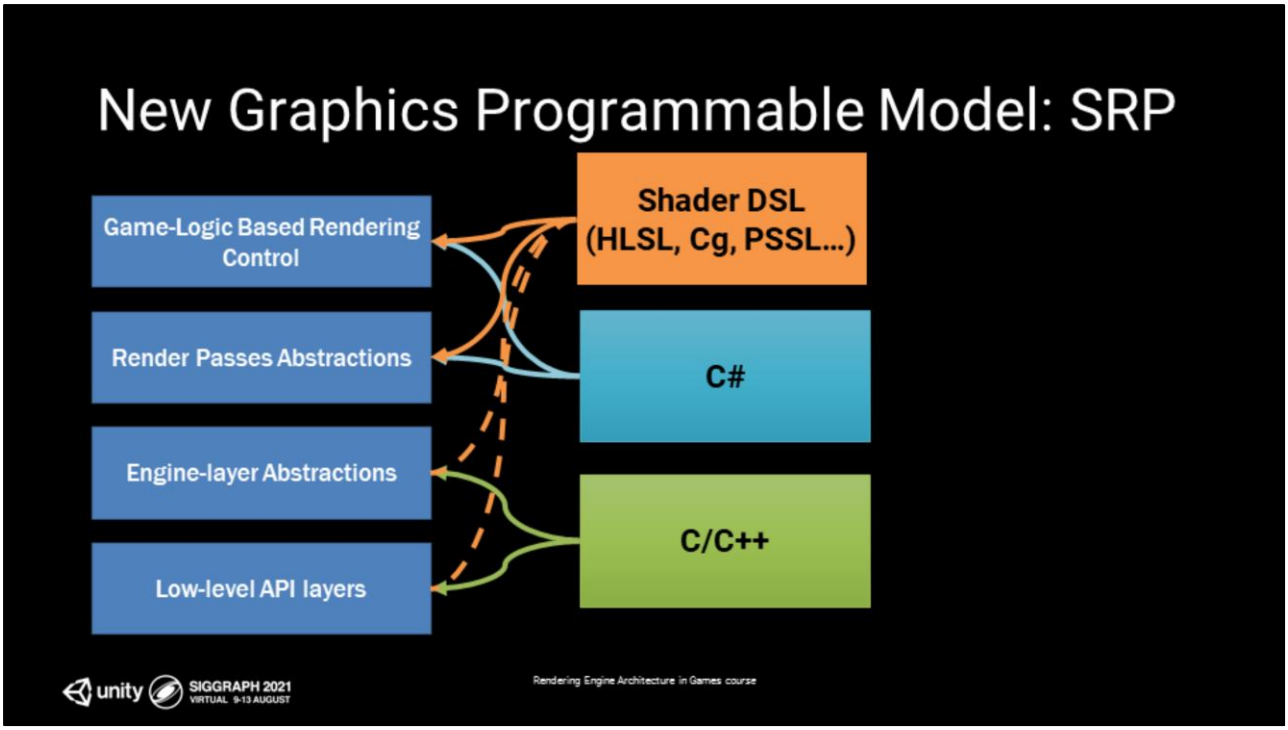
And, last, but not least, since we have moved majority of iterative algorithm development into C# and shaders we can reap all the benefits of quick iteration with hot reload and parameter changes. That's one of my favorite parts, personally.



With SRP, low-level engine still stays in C++ land as before. This is our high-throughput, performance-critical layers.

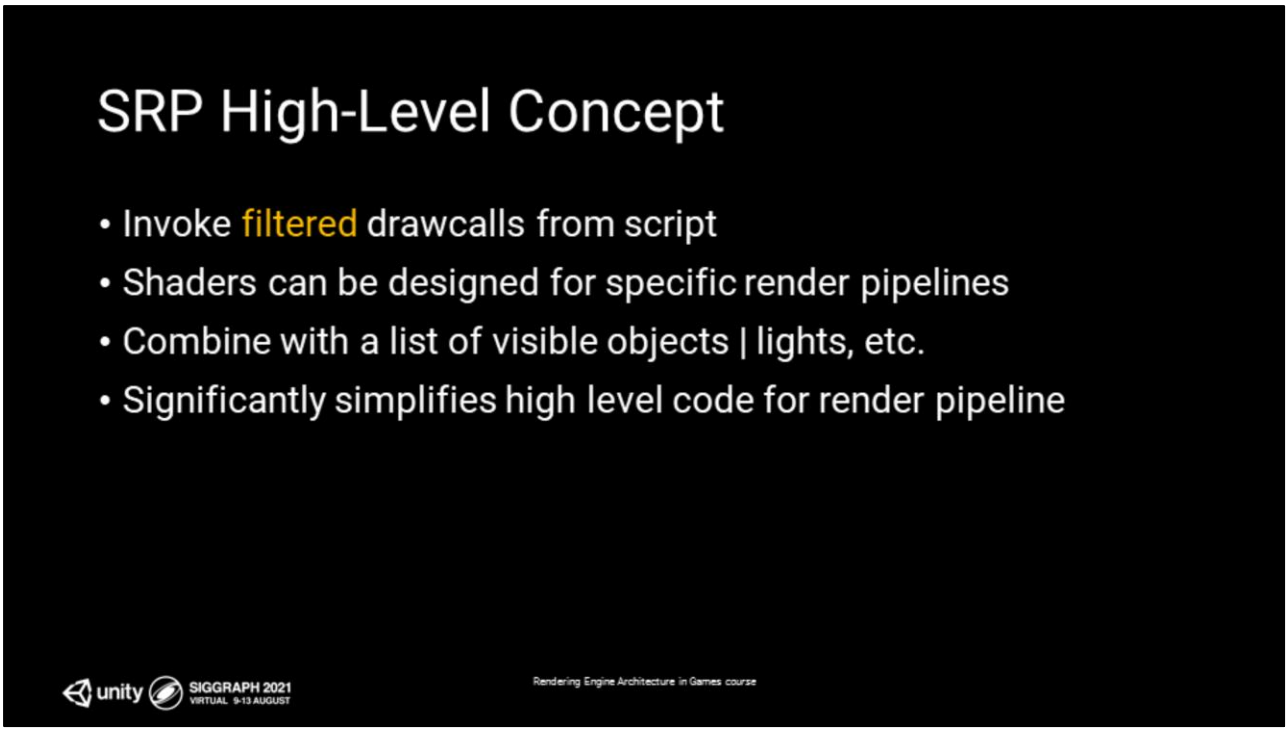


But we have moved some of the functionality for controlling render passes and game-based rendering into C#



Of course, the shader programming is still present, but shaders mostly need to agree just with C# layers (with the exception of high throughput instance data, which is still provided by C++ for performance reasons)
which still need low-level exposure as before, for example, all instance data to shaders still comes from the C++ side: Transform matrices, light indices, reflection probe settings, light settings, layer settings, lightmap indices and UV rects, interpolated probe SHs, for faster setup).

Table for all hardcoded C++ shader instance data:
https://blogs.unity3d.com/wp-content/uploads/2019/02/Screen-Shot-2019-02-27-at-3.50.52-PM.png



The interface is designed such that you invoke a culling operation on the scene graph followed by a draw on the resultant list of scene nodes - specifying specific shader passes to draw with. Under the hood the engine will cull the scene graph in a jobified way and identify nodes that pass the culling and drawing parameters that are passed into the C++ low-level renderer architecture. Shaders can be designed for a particular render pipeline in mind.

And now that we understand all the components on the high level, it may seem that we just talked about creating a bunch of legos for devs to use - so this is a good time to think about Legos for a second.

If we look at The Lego Group, over the years, they have reached a large number of audiences with many of their customers having different personas, age, needs. But there are two categories that are clear:

Lego classic: this is their main offering, for all audiences - kids can understand and play with it, even adults enjoy playing with it. It provides the flexibility and freedom in creativity - you can build anything you want. But you don’t have the “perfect” pieces that aren’t part of the main building blocks to craft something very fine-tuned.

On the other hand there is **Lego Technic**. This one targets an experienced LEGO builder. These are the the state-of-the-art of Lego products, for the perfectionists. That Bugatti has the perfect curves and shapes from the custom lego pieces designed for that vehicle. Yet, you get what you get - your customizability is limited to what's in the box. You can't build a great helicopter with the Bugatti pieces.

The lego analogy fits how we can think about the starting places for Unity's render pipelines.

Scriptable Render Pipeline

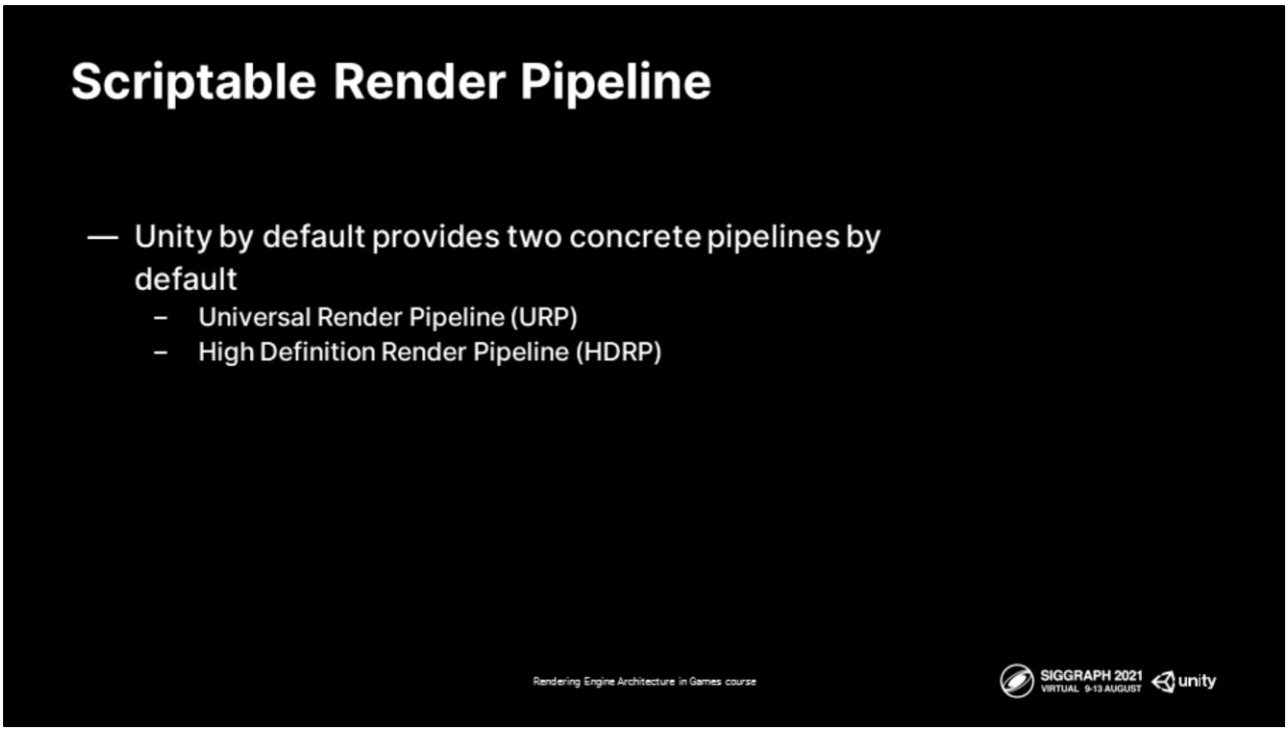
— Unity by default provides two concrete pipelines by default

We ship with two render pipelines “out of the box”

Scriptable Render Pipeline

- Unity by default provides two concrete pipelines by default
  - Universal Render Pipeline (URP)

One that provides maximum platform reach and customizability - the Universal Render Pipeline (URP)



And one that strives for photo realism and high-fidelity graphics, but that reaches a more constrained set of platforms - the High Definition Render Pipeline (HDRP)

Scriptable Render Pipeline

- Users can build their own pipeline
  - Either by modifying existing URP and HDRP pipelines as the base
  - Or completely make your own

Of course, as both of these are user-land customizable, anyone can reuse existing functionality and change and adapt parts of the pipelines that they need to modify. For the hardcore, there is also the path of building a completely custom pipeline from scratch.

LEGO CLASSIC

Universal Render Pipeline (URP)

Beautiful scalable graphics

Maximum platform reach with best performance out-of-the-box

Customizability and flexibility

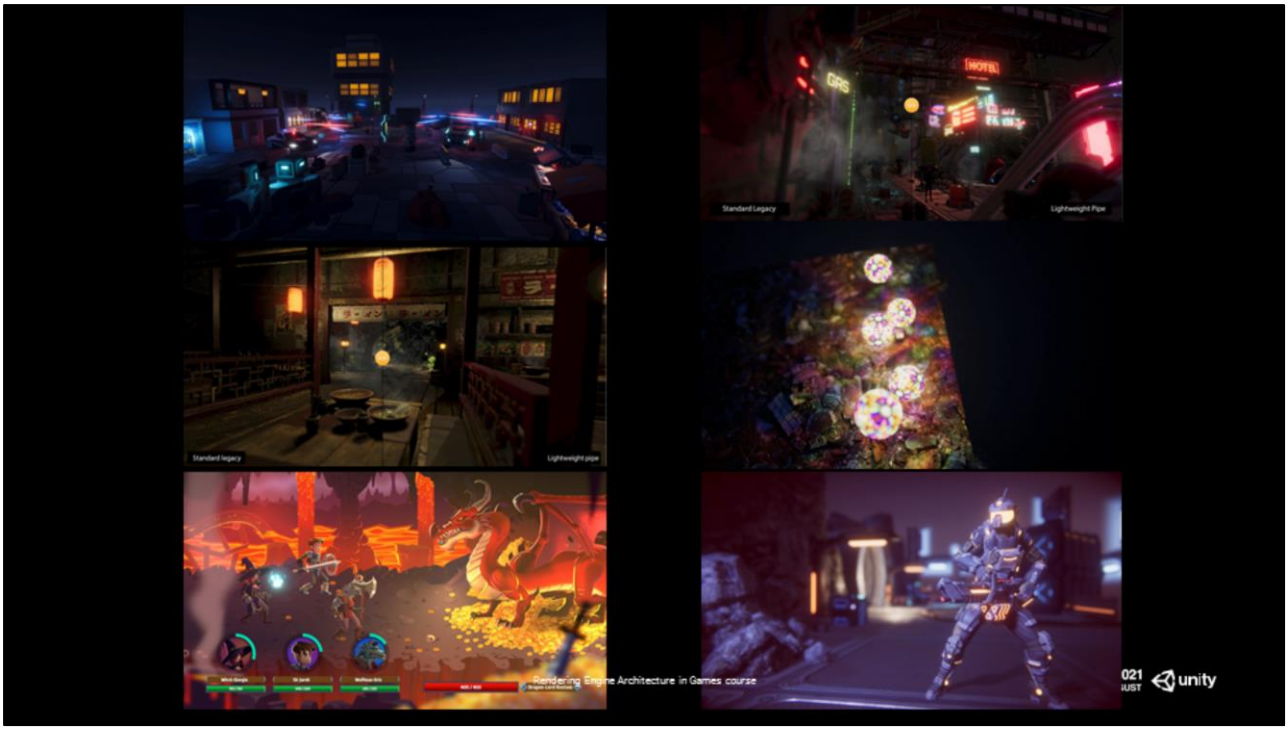
So in the Lego terminology, URP is lego classic. It's accessible, the barrier to entry for developers is lower. You have the maximum flexibility to build the type of visuals you are after and reach your audience on as many as platforms possible and build beautiful creations with great performance. Yet if you are seeking the bleeding edge, you may have some limitations toward perfection.

We think of URP as the powerful successor to the built-in pipeline, designed to be the default rendering for Unity for authoring beautiful 2D and 3D graphics and deploying everywhere.

URP supports the vast majority of Unity’s platforms

And can be used for a variety of examples like some here - 3D,

2D

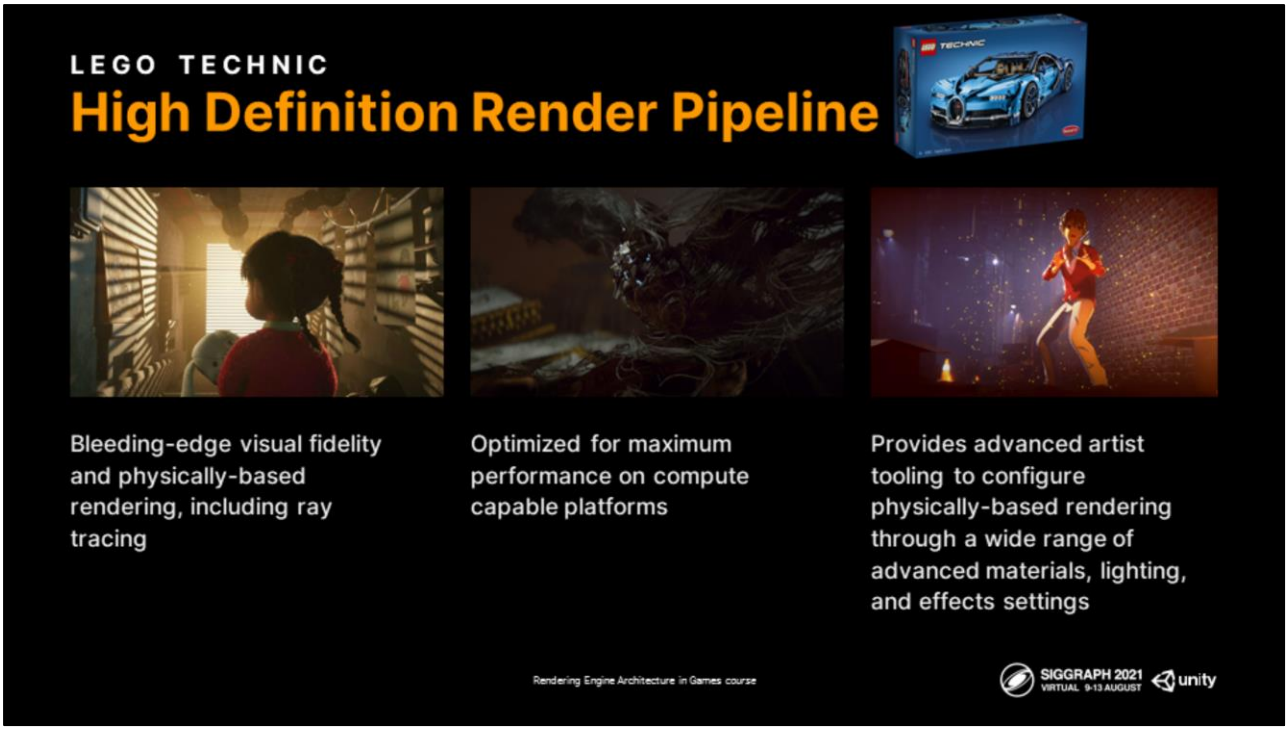


Forward, deferred,


Boat Attack


ROAD
96


AVIERA



HDRP is lego technic. It's designed for the perfectionist, to achieve that state-of-the-art high-end look (photo-realistic or stylized, but physically-based) that you would see in the AAA or offline rendering, with bleeding-edge graphics. It ships with a rich feature set (material models, lighting, shadows, volumetrics, scalability settings, raytracing) but a constrained set of customization options, which is constrained due to console GPU performance reasons.

Our goal with HDRP is to provide the bleeding edge rendering - the high-fidelity 3D graphics for high-end platforms, with a rich feature set (material models, lighting, shadows, volumetrics, scalability settings, raytracing). While HDRP offers a set of customization options, the set is constrained for performance reasons.

HDRP has been optimized for maximum performance on GPU-compute capable platforms such as these here.

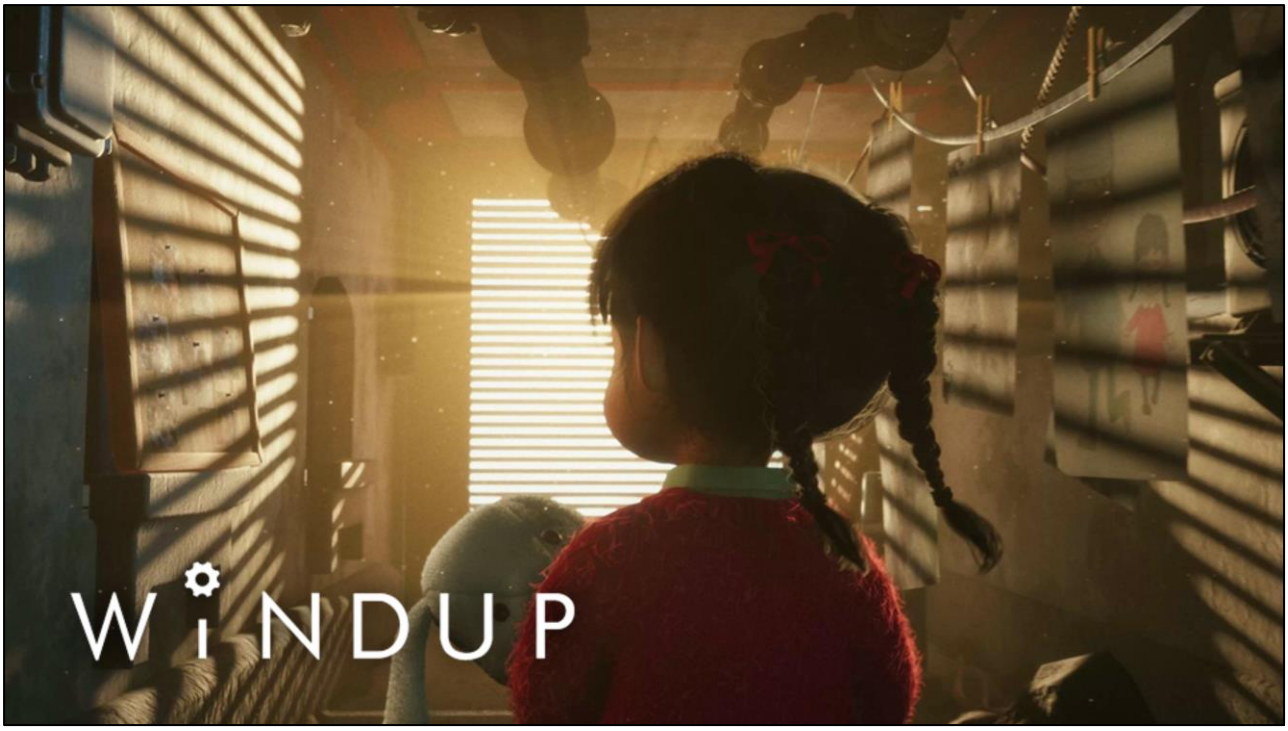
Here are some examples of what HDRP is capable of...


AVIERA
Rendering Engine Architecture in Games' course
SIGGRAPH 2021
VIRTUAL 9-13 AUGUST
unity

https://forum.unity.com/threads/unity-experimental-hdrp-dxr.656092/page-21#post-6455248


THE
HERETIC


WINDUP

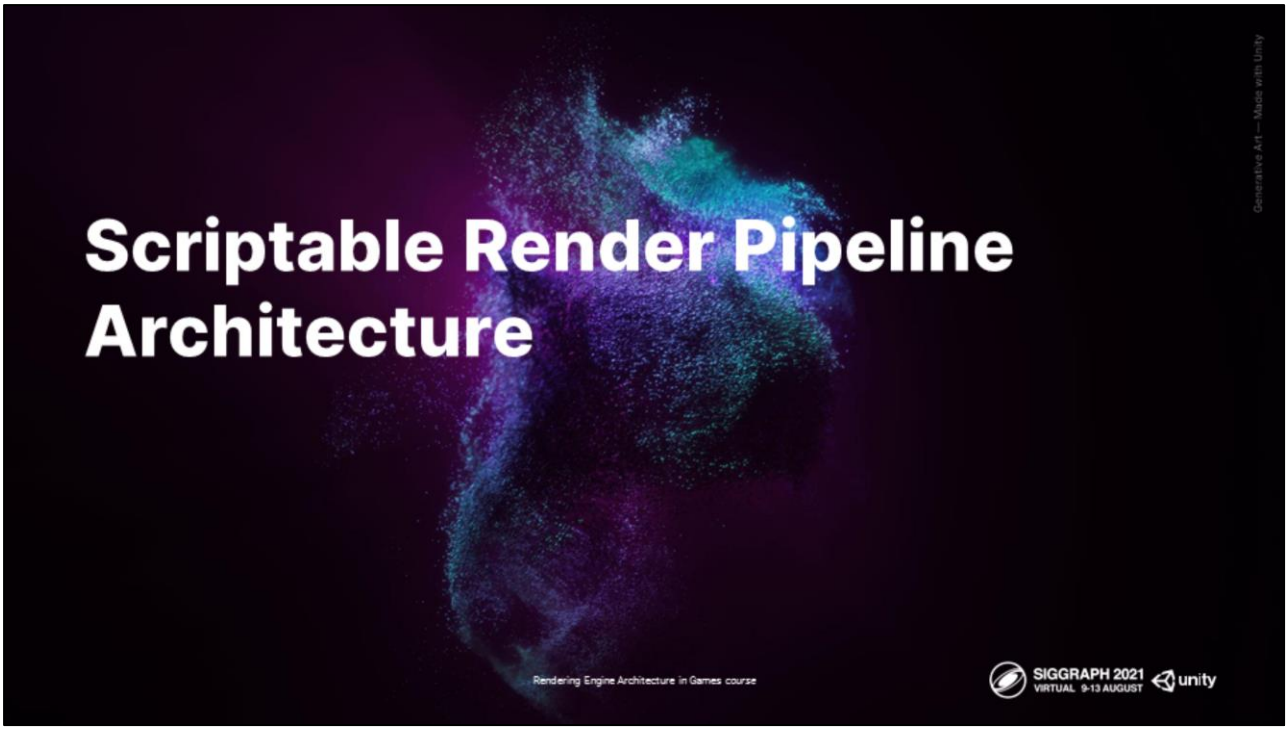

WINDUP


WiNDUP

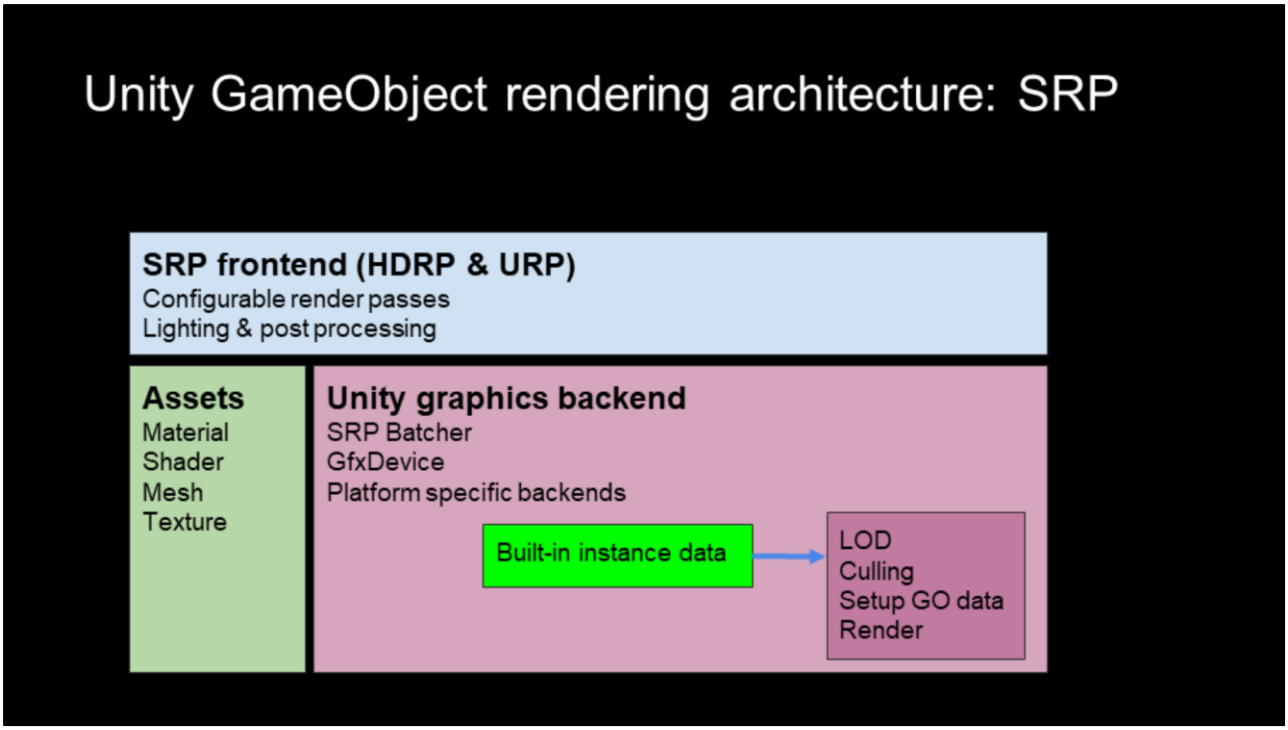

WiNDUP

So with this, I'd like to hand this over to Tim, who will tell you more about how we designed the scriptable render pipeline architecture in more detail.

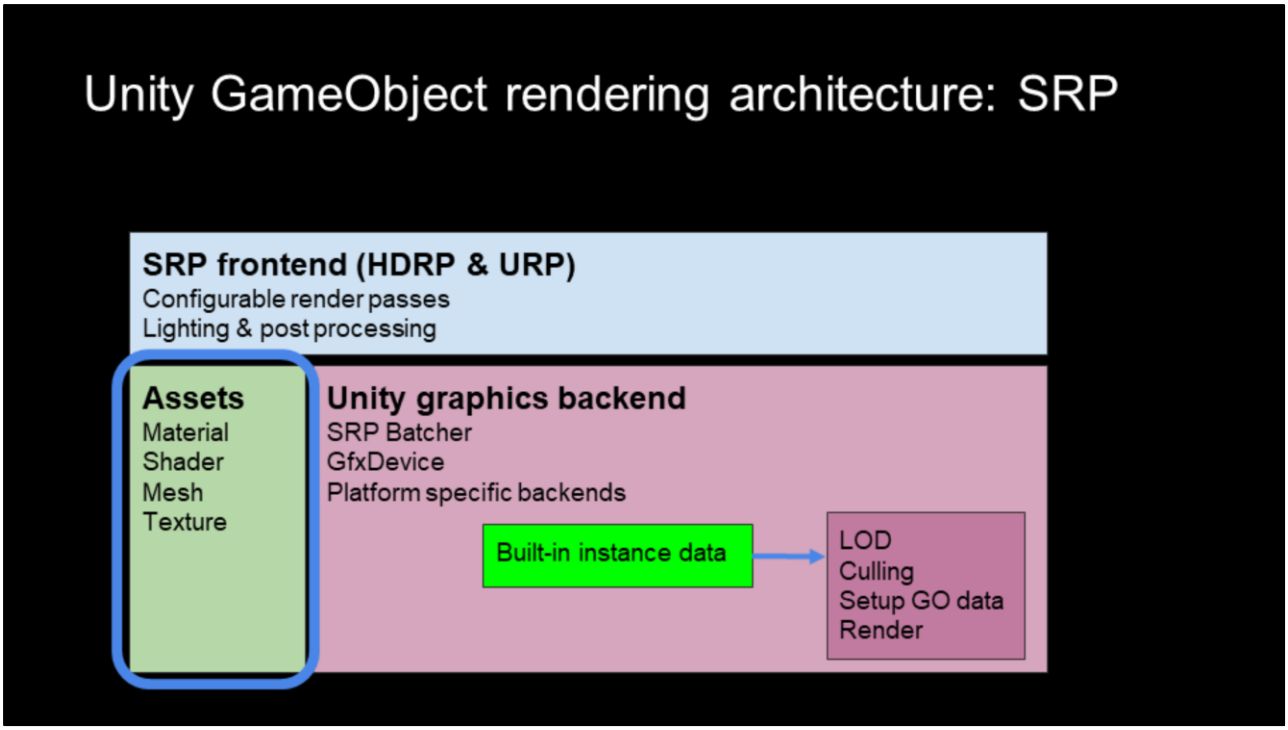
https://twitter.com/Digixart
https://www.polygon.com/2020/12/10/22166257/road-96-release-date-price-the-game-awards



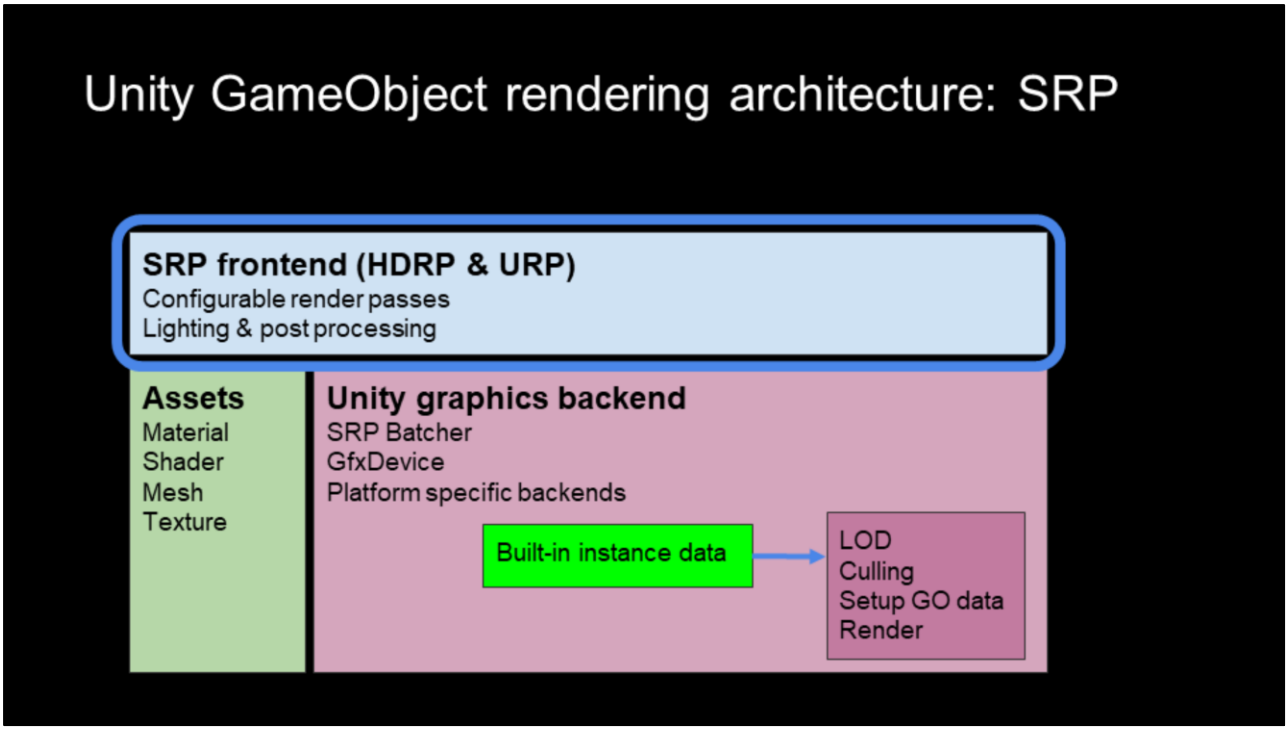
Hi I'm Tim, director of the graphics foundation team at Unity and I'm going to talk today about our journey from a fixed, blackbox style rendering technology to a much more flexible and customizable architecture called the 'Scriptable Render Pipeline'.



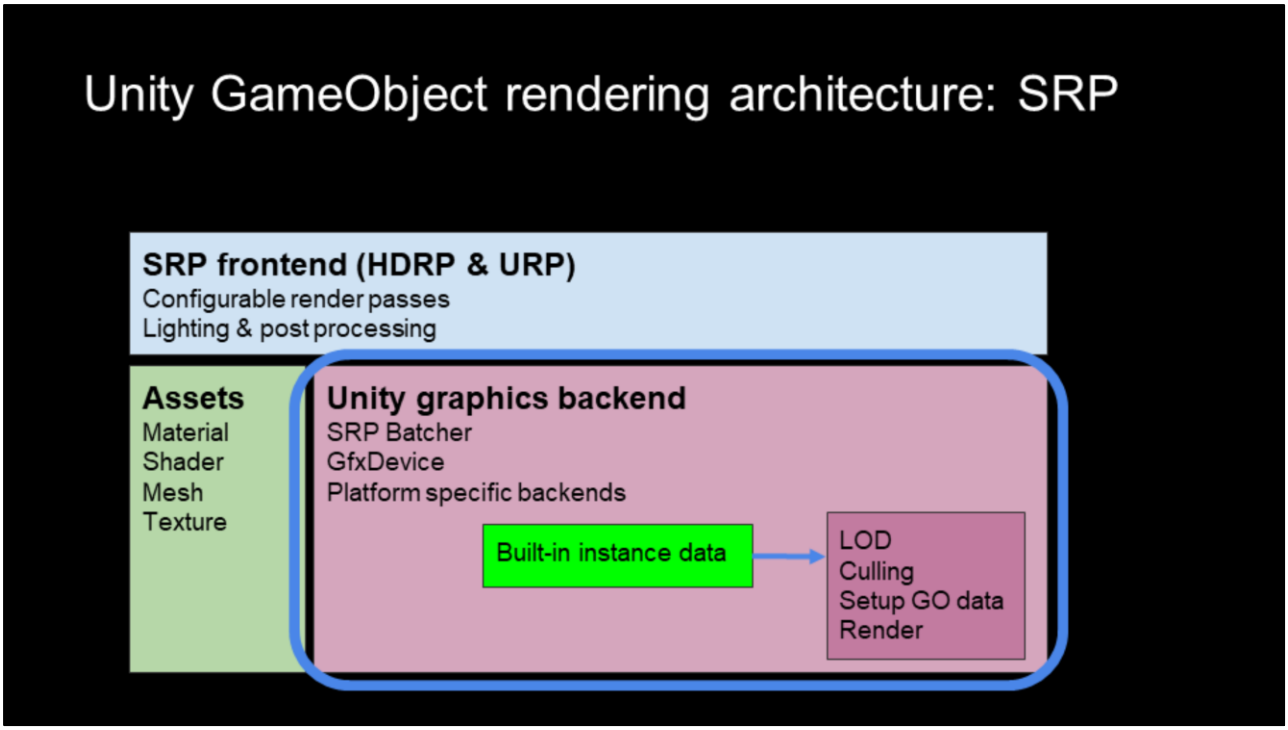
We will be taking a look at different parts of the Unity rendering architecture today, covering parts of the asset interface, our graphics backends, and the SRP api that both HDRP and URP utilize for their rendering.



The asset layer is the front end user data interface into rendering - This is the space where content created by artists and similar is injected into the renderer.

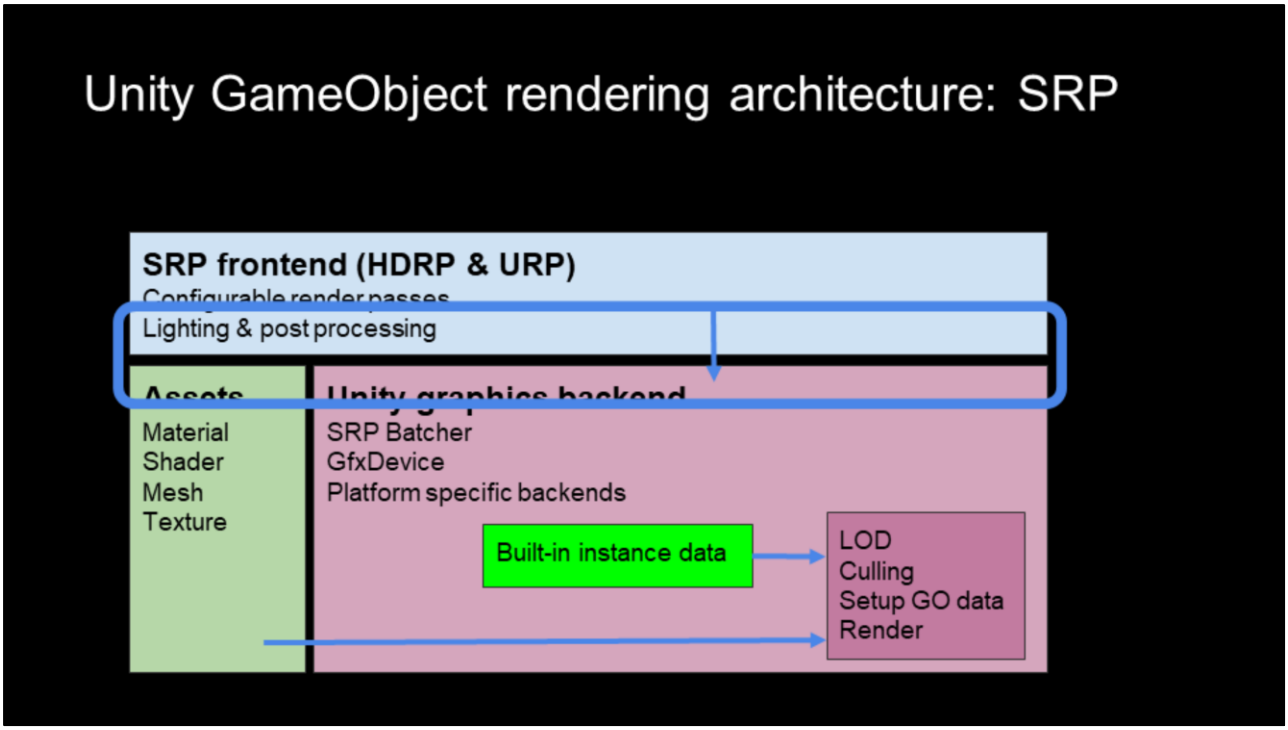


The SRP frontend is a scripting layer that lives in userland where you can customize your rendering. This is where you would write your rendering flow or processing and is the layer where URP and HDRP are implemented.



The graphics backend is where all of our core rendering lives - batching, graphics device interaction, and other parts of our performance sensitive code.

This layer consumes the authored asset data by processing instructions injected from the SRP frontend.



We will cover all these layers but the SRP API is what allows the SRP technology to be what it is - flexible, fast and configurable.

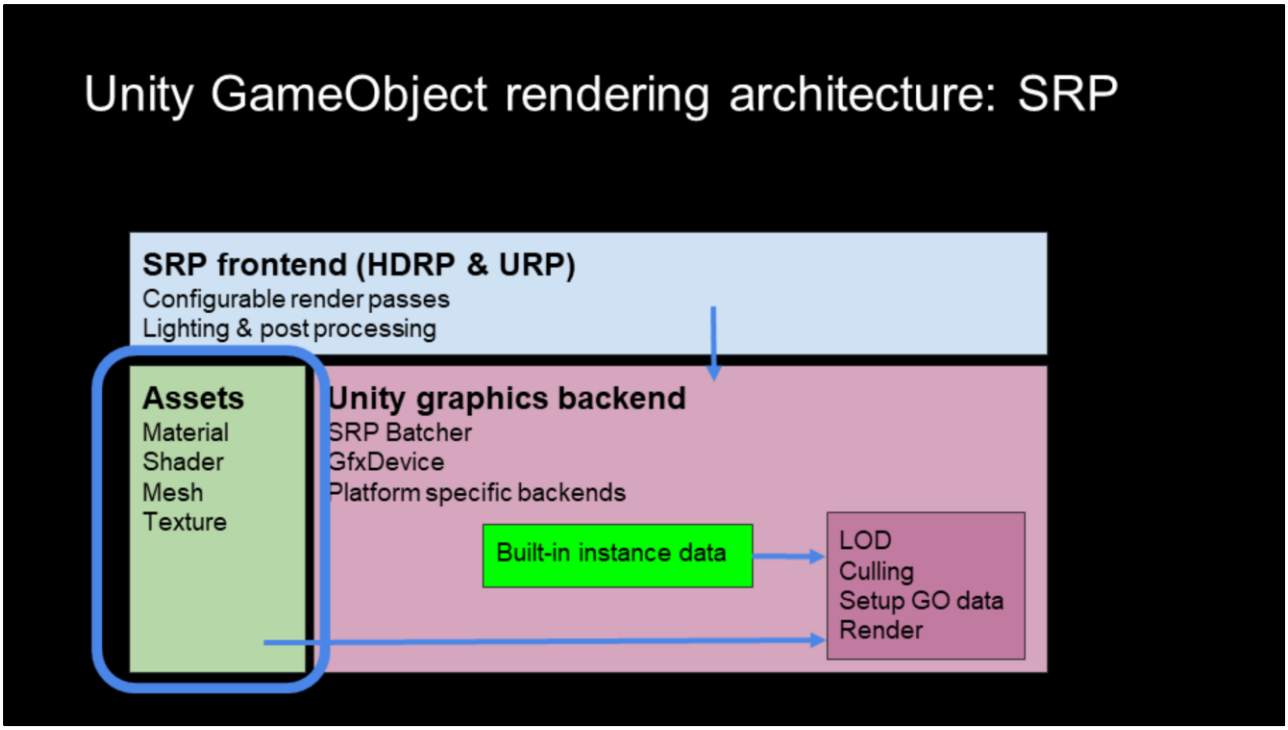
Contrasting our previous rendering technology we really wanted to play to the strengths our users see in Unity and extend those strengths to userland rendering.

That is:
-Customizability
-Easy to use interfaces
-Simple to understand asset abstractions

Let's talk about the architecture of the SRP.

One big design fundamental for our SRP abstraction is that it’s built to use many existing unity workflows and concepts.

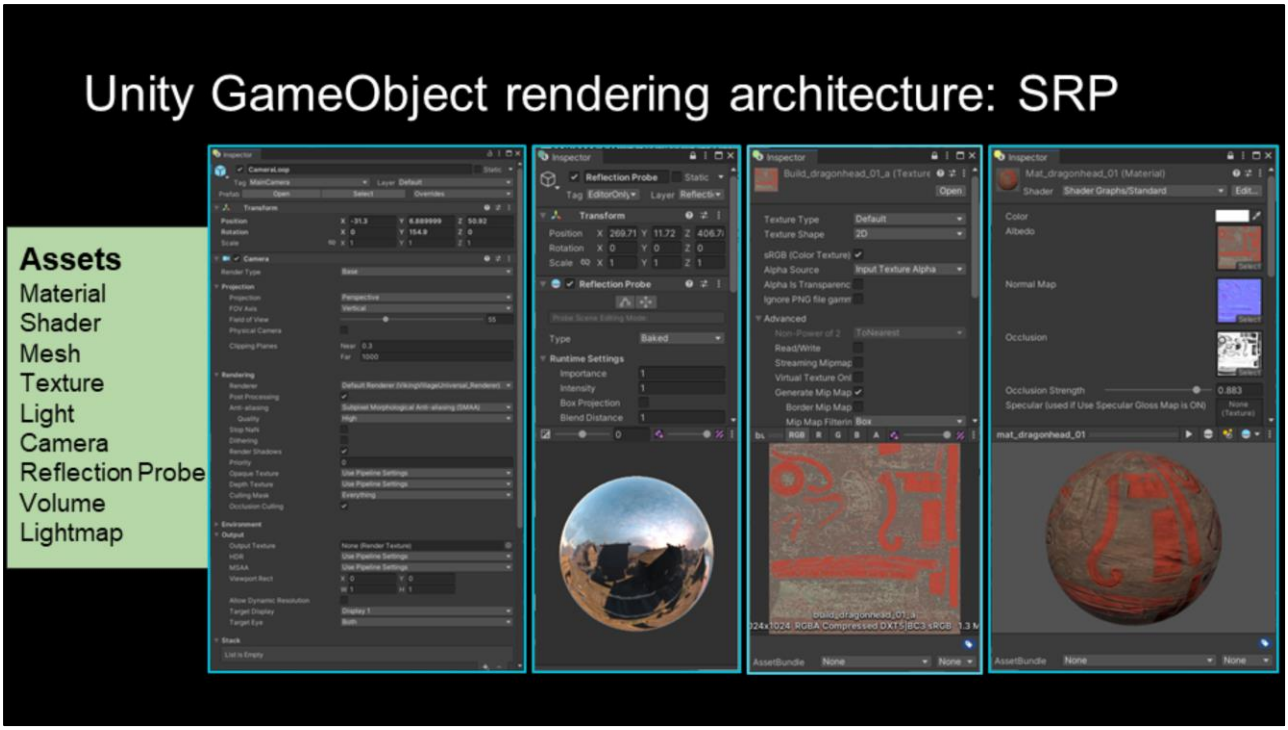
With that in mind let’s take a look at the primary building blocks that content authors work with in Unity.



Jumping back to our architecture you can see there is a block here for assets - this is a stand in for any authored data type that is consumed by the SRP's - and we have various content authoring workflows in Unity to construct this data.

If we start from this perspective - you can see that we still use the same Material, Mesh, Texture, and Shader Object that we used with the previous rendering technologies in Unity.

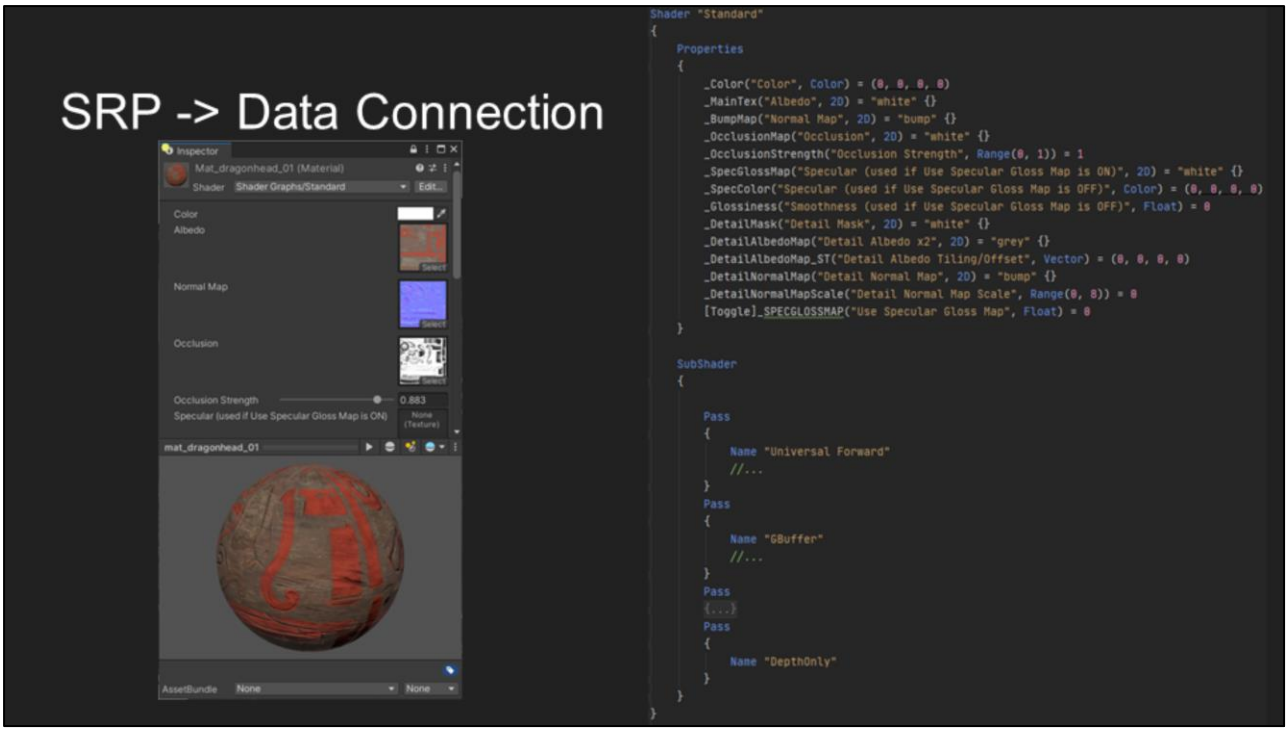
The contents of the shader might be different (pass names, constant buffers, algorithms) but the actual data types are the same.



What's exciting here for our new technologies is that we are utilizing existing _constructs_ for our data interface to rendering - a material in SRP Unity is still just a Unity material.

This provides users with a way to upgrade their existing projects into the new technology by either replacing the data in place, or running an upgrade process.

For URP and HDRP we wrote upgrade scripts to convert built-in content to the newer render pipelines.



In the end most of these asset types are ‘data containers’ with some algorithmic smarts and front end tooling around the outside of them. When it gets to the rendering portion of Unity we mostly just want to consume the settings as needed, for example binding a texture for a draw call or setting a color.

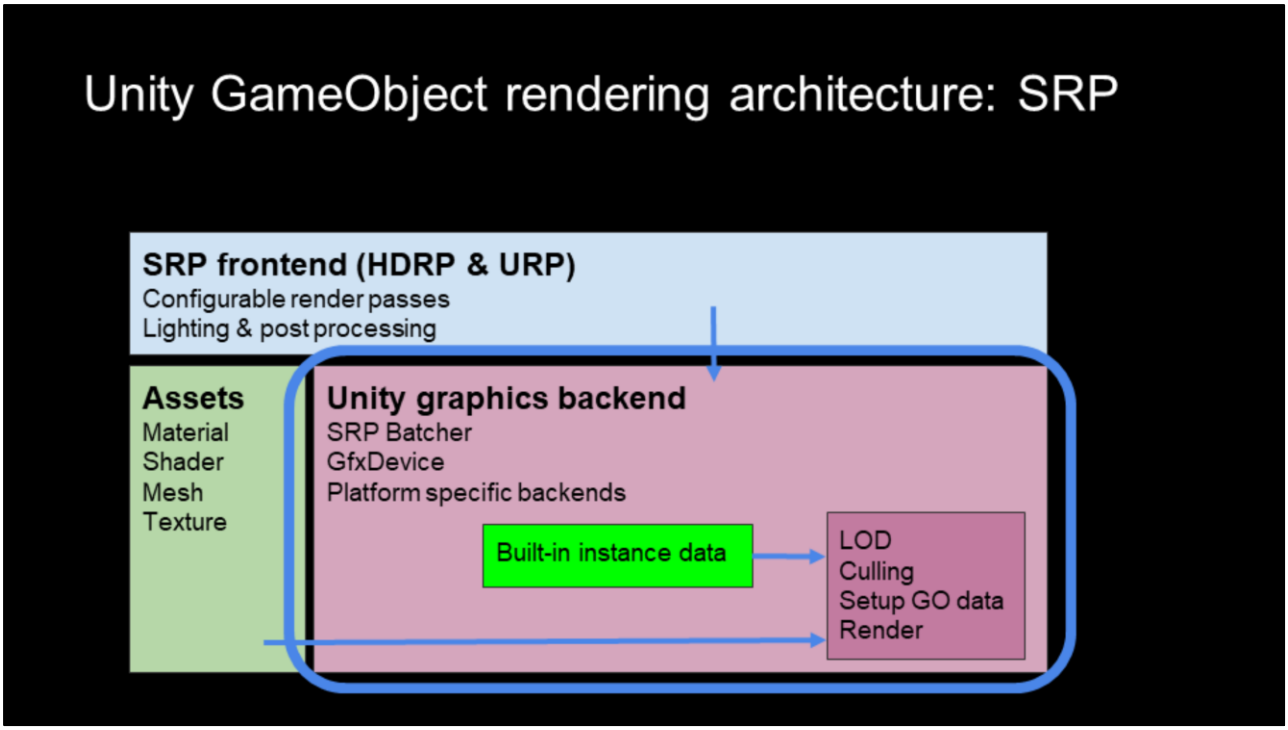
One asset area I want to cover in a little more detail is our material and shader system as this is a big area for customization when rendering.

Our shader object is essentially an interface which contains:
-A block of properties - such as colors, textures, vectors
-A Number of named passes containing HLSL that can use these properties

Our material object is essential a configuration of this Shader object that contains:
-Specific values for the data in the shader properties

In Unity a shader is essentially an interface, and a material is a specific override of that interface with concrete properties.

When performing rendering via the SRP API you are required to specify the specific ‘pass name’ you want to draw. This is the connection between the SRP and the shader side of rendering.

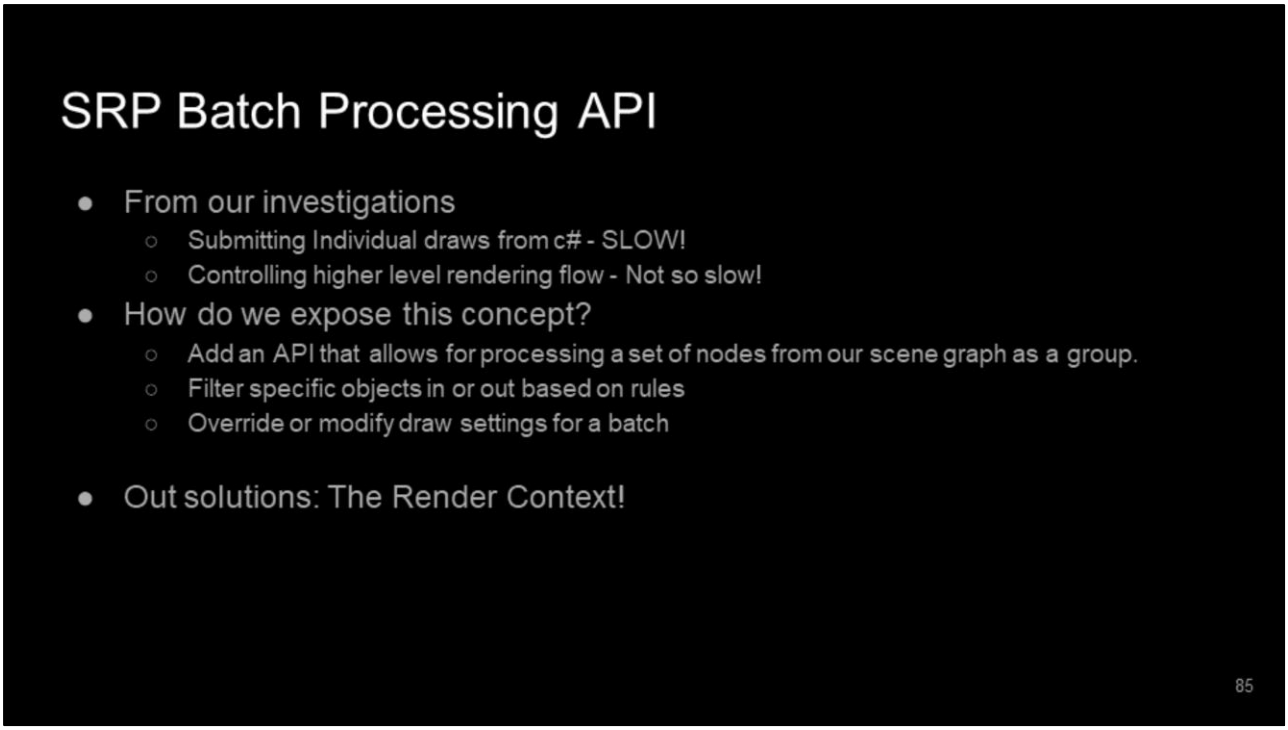


Now,, how do we utilize these assets when it comes to rendering -> these assets are data types that live in our graphics backend which are then attached to nodes on our scene graph. This means we have access to them at render time.

The core of the Unity engine is built in C++ this is where large parts of the runtime live. A lot of the functionality that you would associate with performance sensitive rendering lives in this layer: batching, gfxdevice interaction, threading, graphics jobs. Muh of this portion of the engine is heavily OOP based - this leads to some pointer soup and similar when performing rendering operations. Sebastian will talk about how we are improving this later.

This is also where our scene graph lives and many of the operations invoked on the SRP API trigger batch processing operations to this scene graph.

But I'm getting a little ahead of myself - What API do we have in SRP to interact with this graphics backend.



We needed this drawing API to be as fast as possible - it was a strict requirement for us and c# does not offer the performance we need for per node operations.

We converged on solution that allows for higher level rendering flow to be controlled from c#, but draws to be controlled as batches.

Essentially post culling we would have a list of all valid RenderNodes and be able to say: “Render the opaque ones front to back” or “Render the transparent ones back to front”.

This allows for rendering control flow to live in c# without drastically hurting runtime performance.

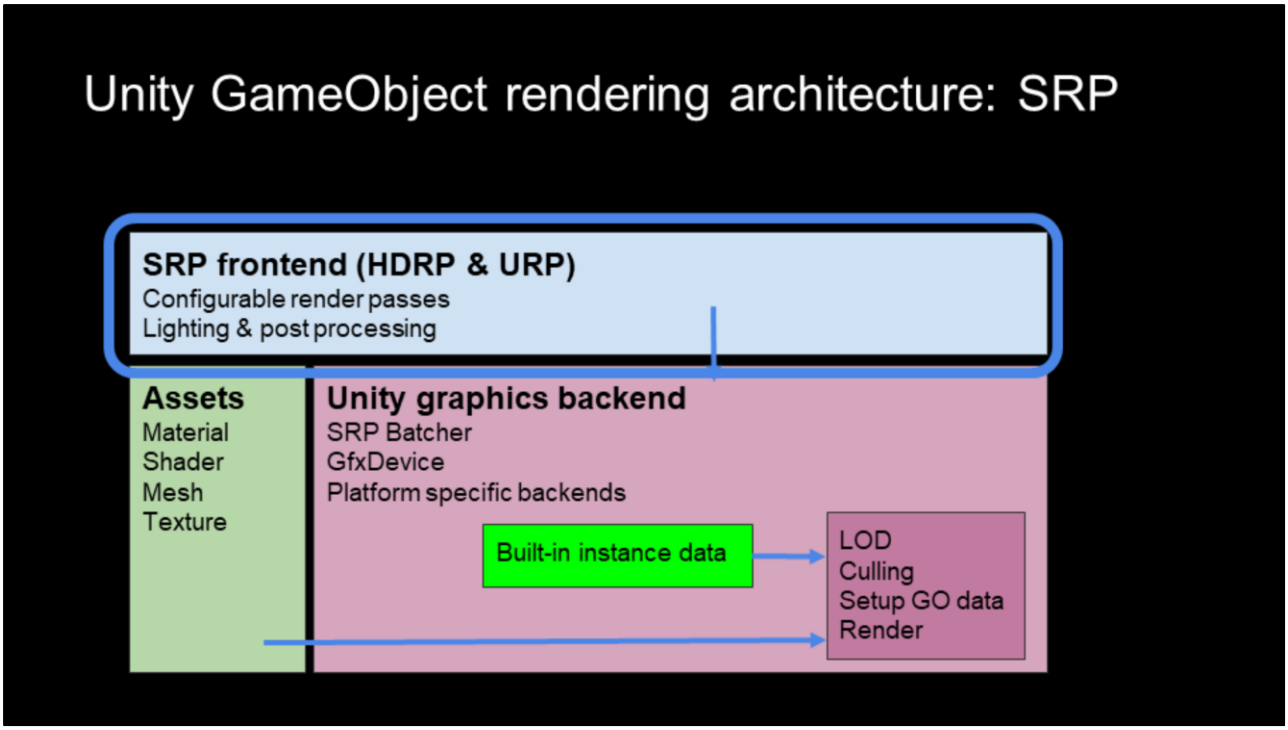
The API we arrived at for this was called the RenderContext - This is a proxy object that lives within our c++ layer but has a binding layer into c# - Holding this object allows access to our rendering API.

Looking at this mock render pipeline implementation you can see the granularity of operations that are possible on our context.

Like I mentioned before they are not super low level but they provide a solid granularity for working with rendering algorithms. This is the lowest level of our exposed scaffolding and on top of this we build many features like the render graph and similar.

As we discover new requirements of receive feature requests we extend this API to add the new functionality

For us this strikes the right balance between performance and usability and ties in well with the principles that we have established for our API.

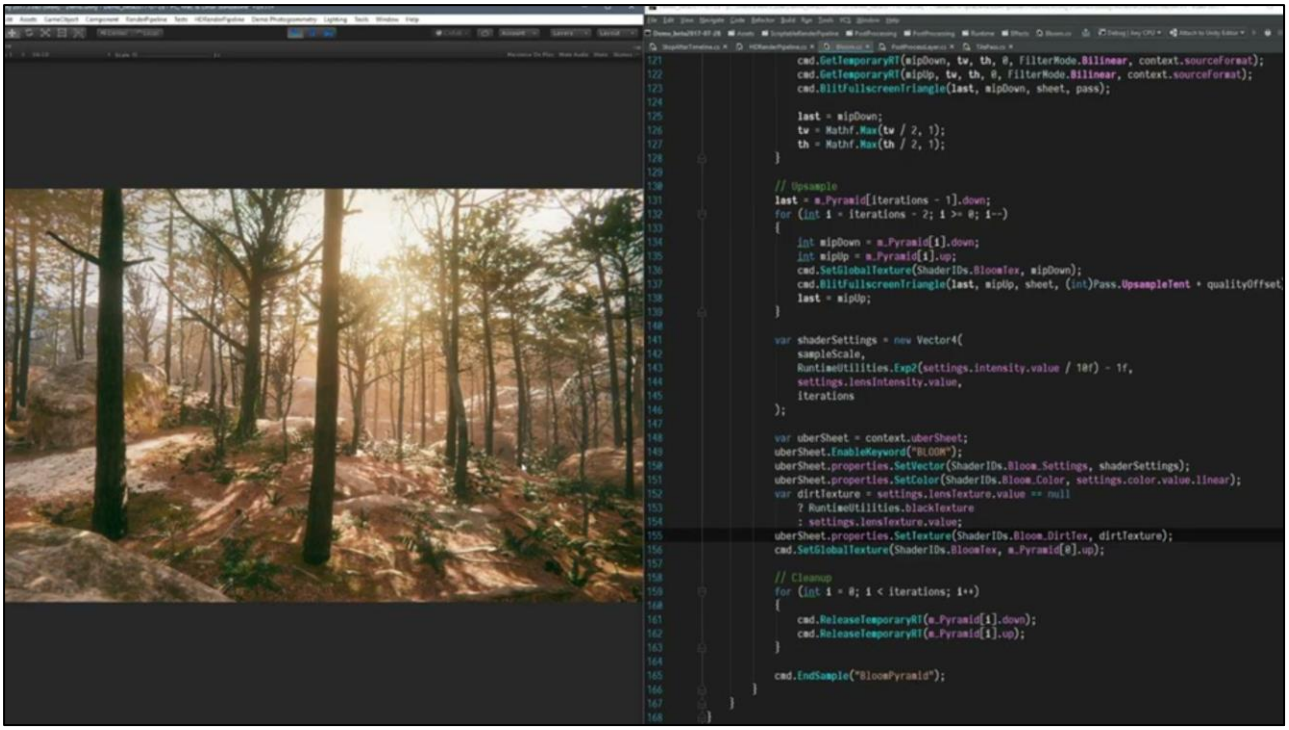


On top of this c++ layer we have our c# scripting layer and this is where URP, HDRP, or a custom render pipeline can be implemented.
This layer takes advantage of many of Unity's strengths as a development platform

Specifically:

- Our Ability for user code to be written in a sandbox away from from the core engine, a crash in this code won't take down the whole engine.
- And the ability to Leverage an existing well known programming language C#.

We also have a number of utility tools that live in this space as well that were needed to build both URP and HDRP. We'll dive into some of them later and how we have approached the idea of scaffolding when we built them.



```
                cmd.GetTemporaryRT(mipDown, tw, th, 0, FilterMode.Bilinear, context.sourceFormat);
                cmd.GetTemporaryRT(mipUp, tw, th, 0, FilterMode.Bilinear, context.sourceFormat);
                cmd.BlitFullscreenTriangle(last, mipDown, sheet, pass);

                last = mipDown;
                tw = Mathf.Max(tw / 2, 1);
                th = Mathf.Max(th / 2, 1);
            }

            // Upsample
            last = m_Pyramid[iterations - 1].down;
            for (int i = iterations - 2; i >= 0; i--)
            {
                int mipDown = m_Pyramid[i].down;
                int mipUp = m_Pyramid[i].up;
                cmd.SetGlobalTexture(ShaderIDs.BloomTex, mipDown);
                cmd.BlitFullscreenTriangle(last, mipUp, sheet, (int)Pass.UpsampleTent + qualityOffset
                last = mipUp;
            }

            var shaderSettings = new Vector4(
                sampleScale,
                RuntimeUtilities.Exp2(settings.intensity.value / 10f) - 1f,
                settings.lensIntensity.value,
                iterations
            );

            var uberSheet = context.uberSheet;
            uberSheet.EnableKeyword("BLOOM");
            uberSheet.properties.SetVector(ShaderIDs.Bloom_Settings, shaderSettings);
            uberSheet.properties.SetColor(ShaderIDs.Bloom_Color, settings.color.value.linear);
            var dirtTexture = settings.lensTexture.value == null
                ? RuntimeUtilities.blackTexture
                : settings.lensTexture.value;
            uberSheet.properties.SetTexture(ShaderIDs.Bloom_DirtTex, dirtTexture);
            cmd.SetGlobalTexture(ShaderIDs.BloomTex, m_Pyramid[0].up);

            // Cleanup
            for (int i = 0; i < iterations; i++)
            {
                cmd.ReleaseTemporaryRT(m_Pyramid[i].down);
                cmd.ReleaseTemporaryRT(m_Pyramid[i].up);
            }

            cmd.EndSample("BloomPyramid");
        }
    }
}
```

One other Unity advantage I wanted to highlight was due to the SRP's being built in the c# layer it allows us to extensively modify rendering code without recompiling the whole engine - This is a really big one - we can change anything in that c# layer and not have to leave the Unity editor tool.

We have really quick iteration and it allows us to easily experiment and prototype without feeling like Unity is getting in the way of our ideas.

Scriptable Render Pipeline

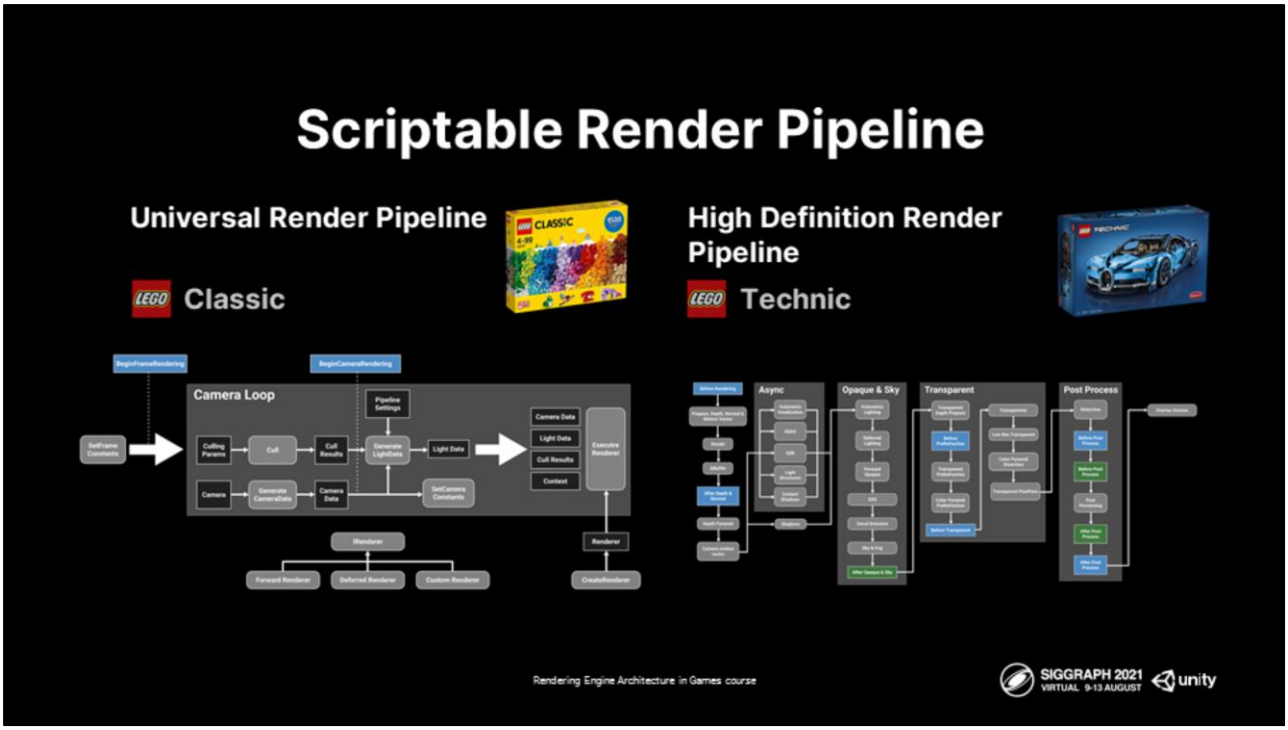
Universal Render Pipeline

LEGO Classic

High Definition Render Pipeline

LEGO Technic

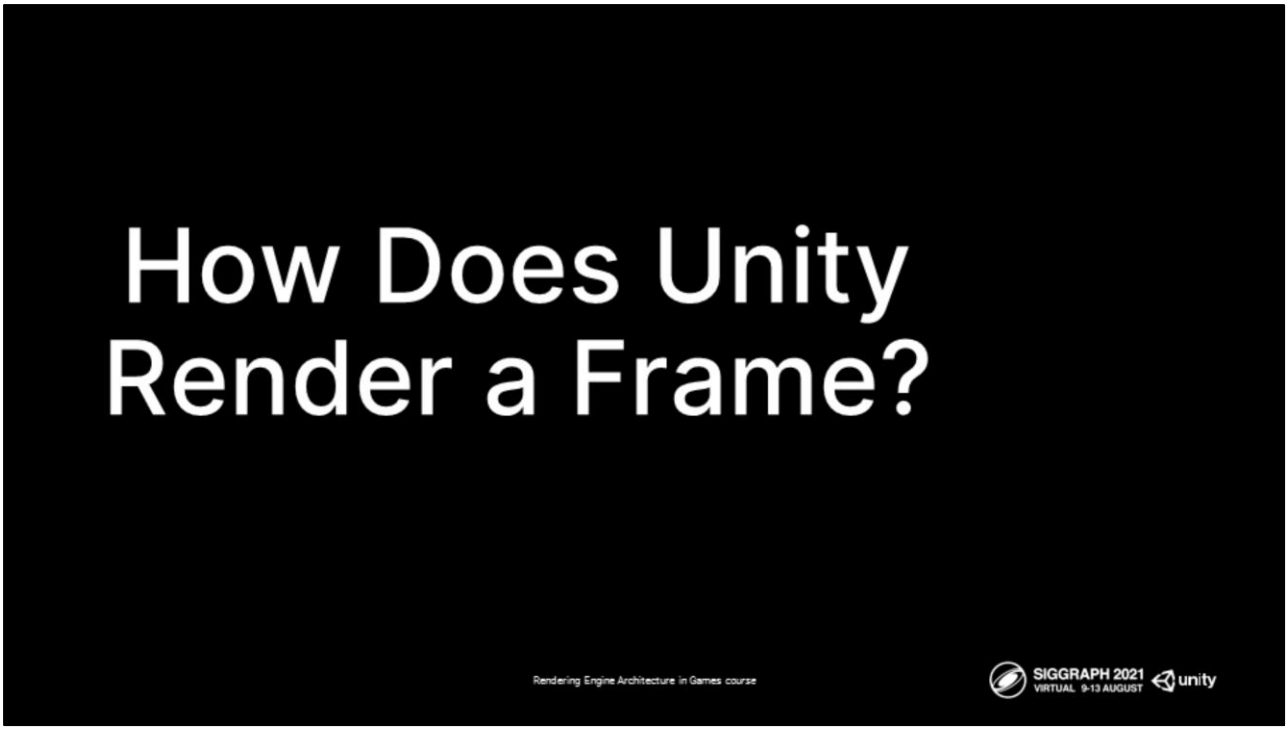
So as we have seen in Unity we have two rendering pipelines targeted at different use cases.



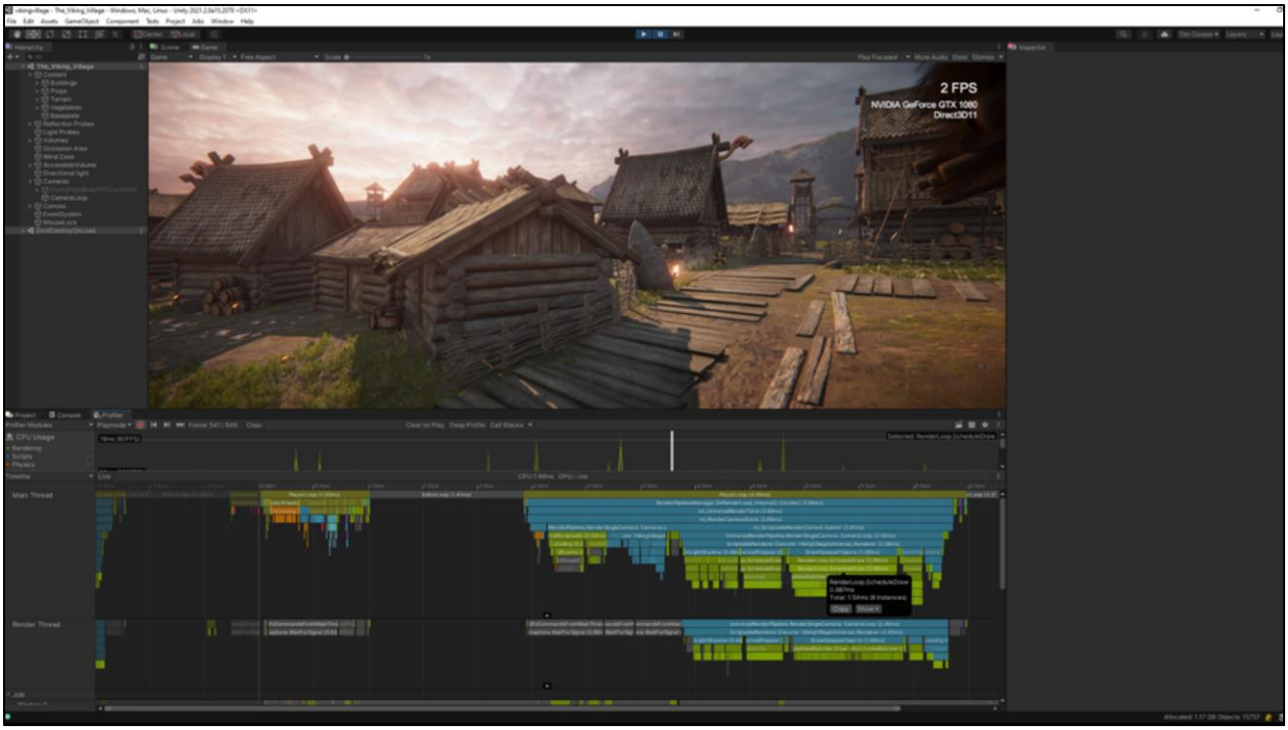
And as you can see they both have very different higher level implementations.

One of our biggest challenges has been about how we can expose both the low level rendering features, as well as higher level constructs in a way where they are usable for multiple render pipelines, including custom user pipelines.

As I have touched on we used the concept of scaffolds for the design of many of our features. That is; we have low level raw API's that you can use if you want direct access to rendering concepts - but we also have higher level constructs built on top of these that are "more usable" but offer less direct control.



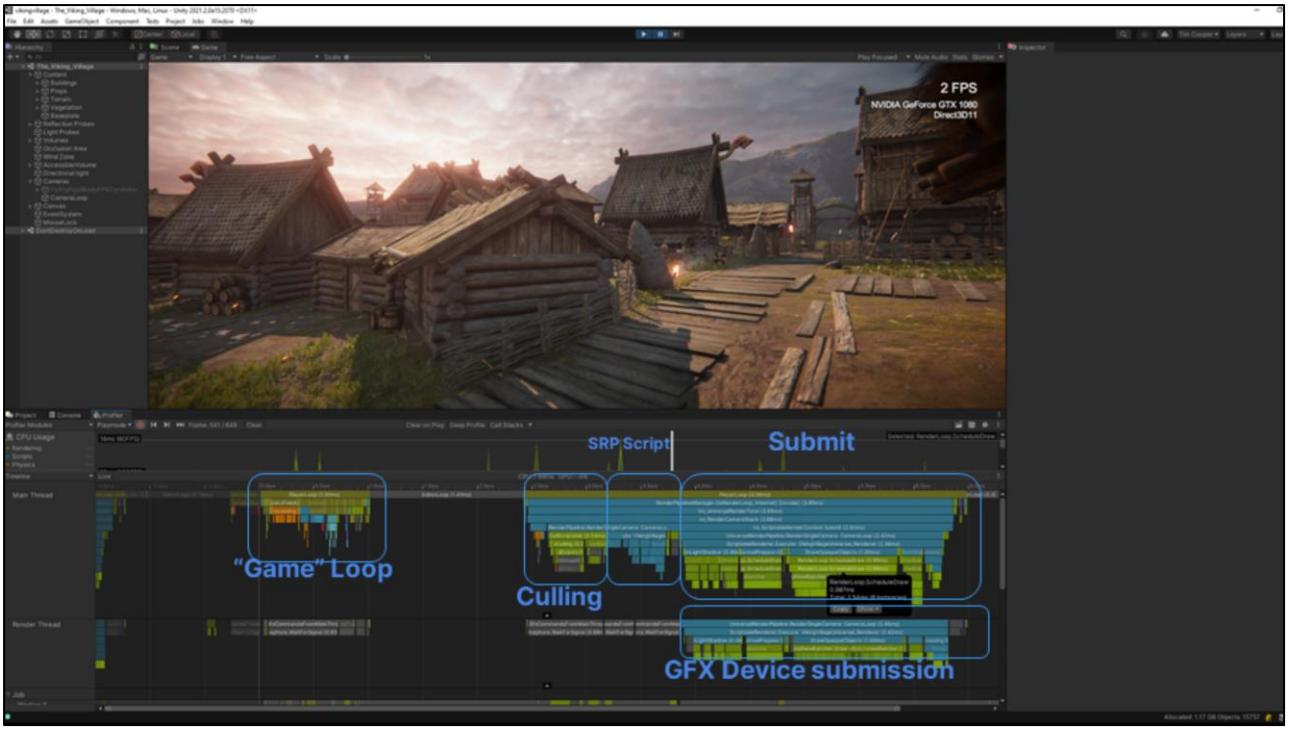
To delve into these we'll first need to understand how unity renders a frame.



To start with, current Unity is a heavily main thread focused engine due to it's extensive c# callback based playerloop.

The advantage of this is that Unity is _very_ fast and easy to prototype with - super low friction from idea to seeing it on frame

But needs extra attention when you are looking to scale up a project that becomes more demanding. you need to ensure you are offloading work to worker threads and playing nice with how our render thread / main thread interact.

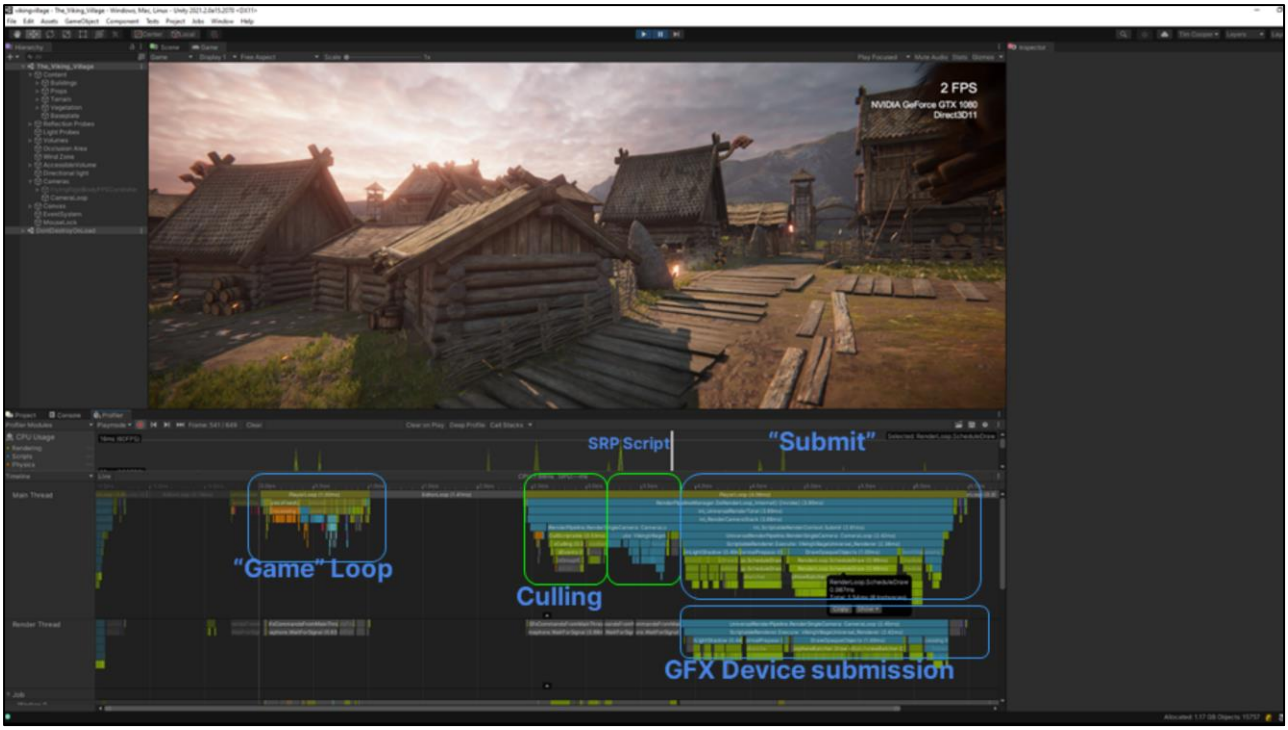


A frame in Unity is build in a number of stages I am using the URP renderer for this example.

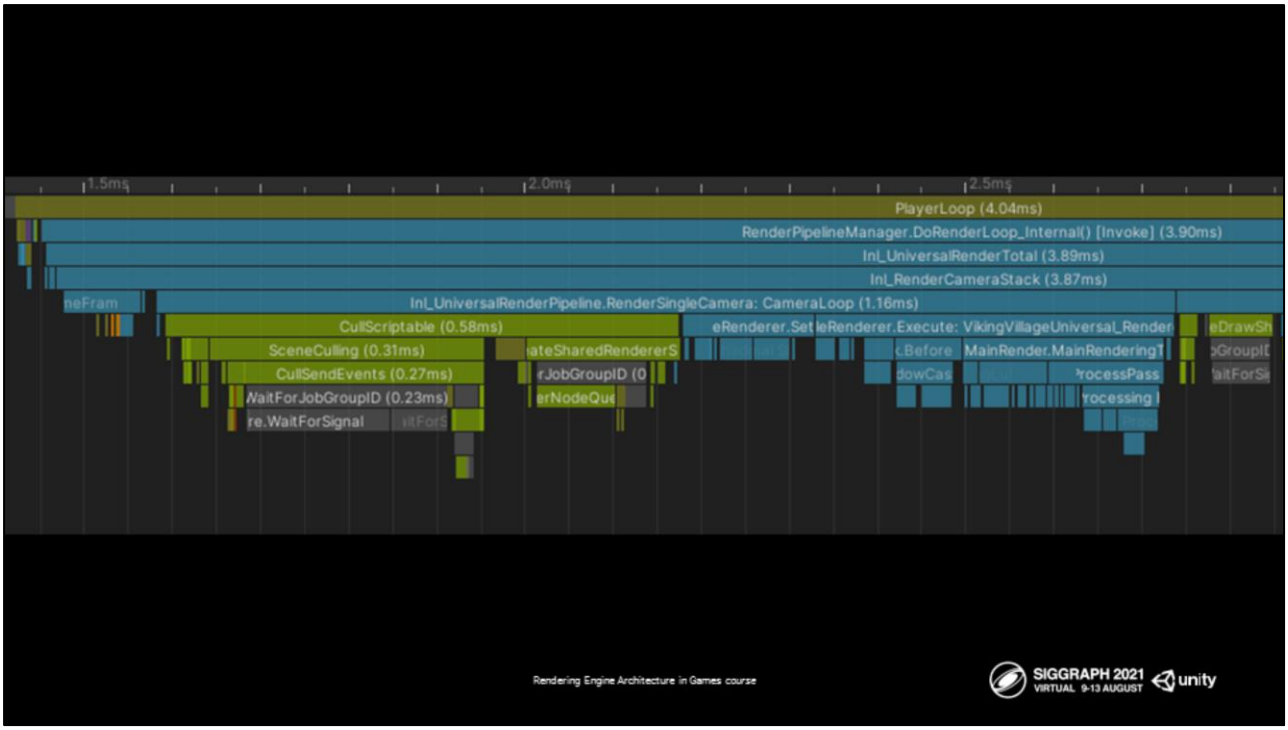
1) We have a *game logic* section in the frame. This is where non rendering systems and user script code is executed.

2) Then we start rendering:
- Step 1: Culling - Includes callbacks to userland code for visibility
- Step 2: Performing the rendering algorithm (on the main thread in userland script code)
- Step 3: Submitting the operations which includes figure out draw state, sorting, and calculating per object draw data (c++)
- Step 4: Offloading the rendering command buffer to the render thread

Each phase of this rendering will offload parts to worker threads, this will scale in a variety of ways depending on content, platform, and project configuration.

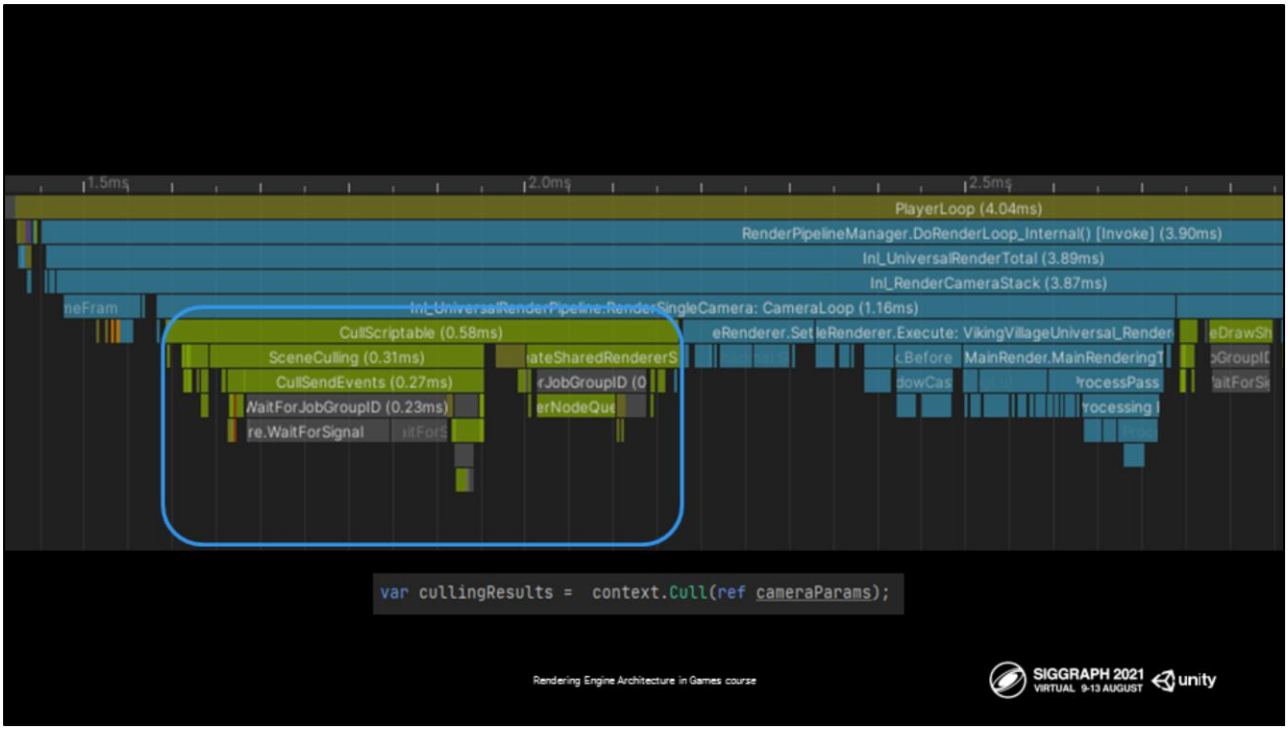


This highlighted section is the ‘render’ call into the c# user layer. Let's dig in to see what is happening here.



A note on coloring: C++ layer code is in green, and c# user code is in blue.

You can see in this frame that the URP render function is called, it will then call into c++ for the culling, before processing the rest of the URP script code in the blue portion.

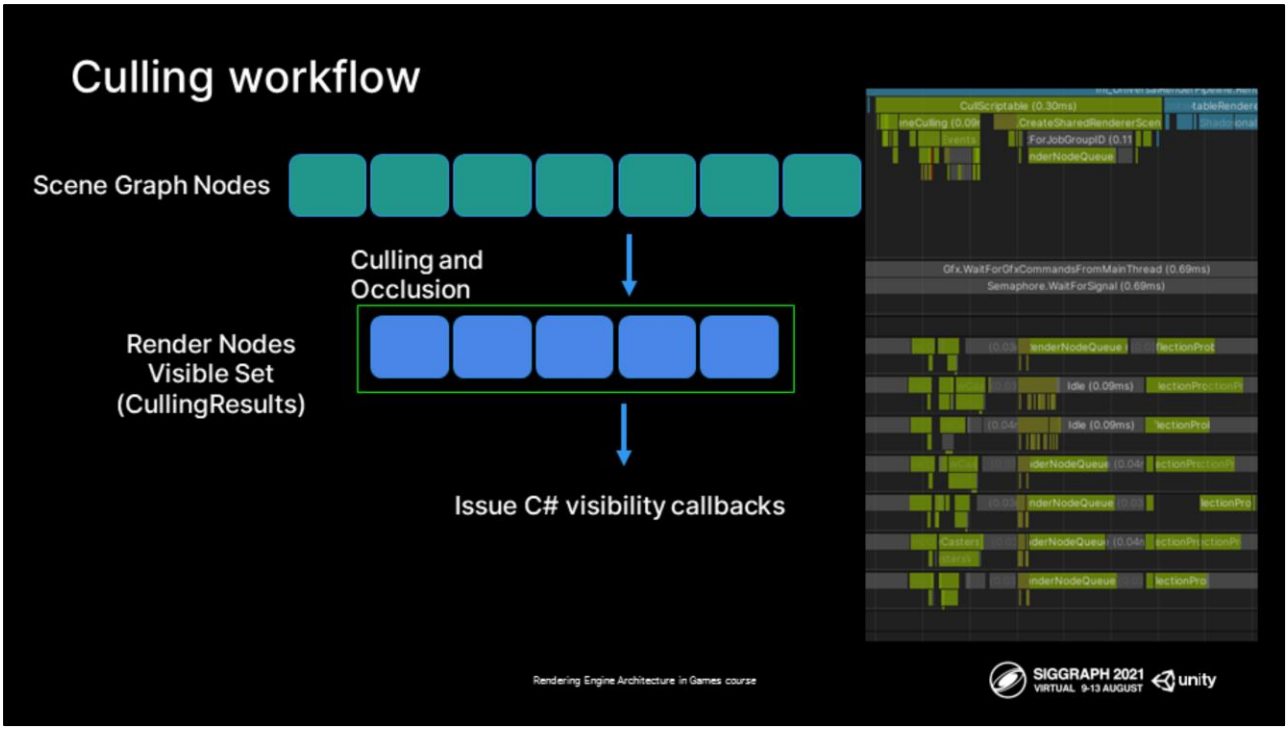


What is the workflow for our culling?

Let's start with intent - we want to perform a lot of mathematical heavy operations on our scene graph and get a renderable set of nodes. This means we need it to be performant and paralyzable so we actually favor performing the culling in C++ with a set of options that can be configured from our c# api.

The is done by a call to *context.Cull*. This will perform the culling operation with whatever camera settings are passed in. Flags exist for things like occlusion, rendering layers and similar.

We let this be configurable so that users can customize the call to their own specific rendering needs.



Whilst the culling itself is quite black box - it offers users a lot in terms of callbacks and ability to add custom code when things become visible or invisible, we have user-friendly workflows here.

Essentially each time a culling operation is called, we parse our scene graph and generate a list of renderable nodes that get wrapped into a culling result opaque handle.

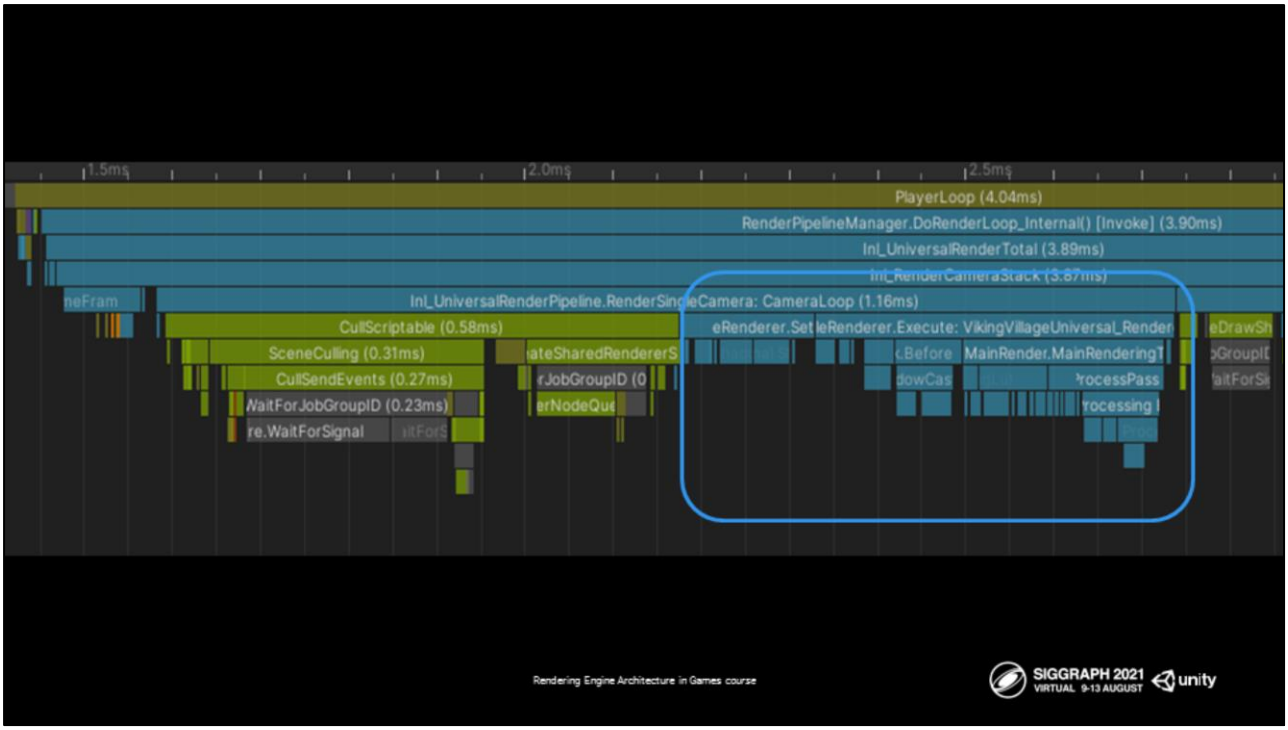
Each time this operation happens we also issue visibility callbacks both to internal systems as well as c# scrip. They are often used to perform calculations that should only be done when an object is visible like skinning or similar.

These at culling time callbacks have their pro's and con's - They allow for 'same frame' updates for visibility operations but add slowdowns as we need to call individual user scripts attached to objects.

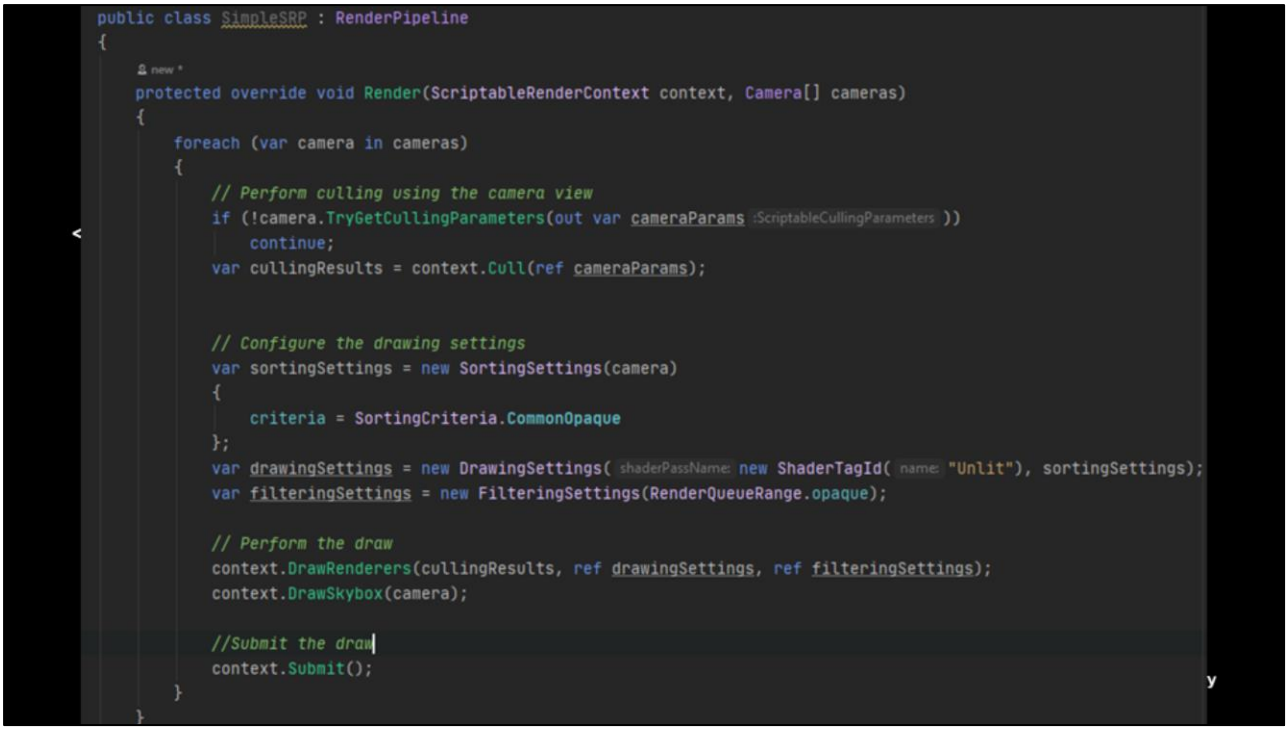
The flexibility and ease of use in terms of API has gotten a lot of projects a long way but this is one of the older systems in Unity and is an area which we want to improve in the future.

For an engine like Unity changes to systems like this can be difficult as we have to balance continued functionality of existing user projects with building and improving the technology. When we make changes to callbacks and similar we need to be very

careful to not break things.



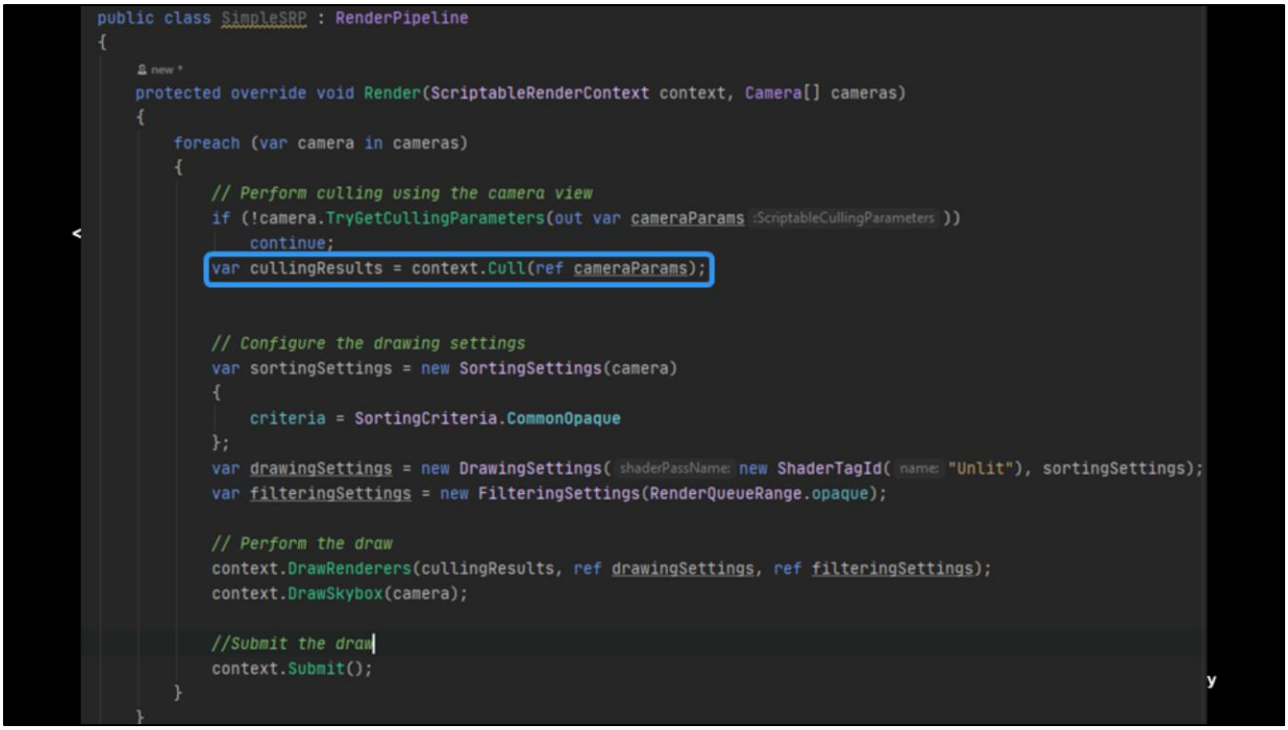
After culling you can then do an actual frame of rendering. The goal of this presentation is not to dive too deeply into the specifics of either HDRP or URP but instead look at the shared rendering API both pipelines use; so how do both pipelines submit work.



```
public class SimpleSRP : RenderPipeline
{
    new *
    protected override void Render(ScriptableRenderContext context, Camera[] cameras)
    {
        foreach (var camera in cameras)
        {
            // Perform culling using the camera view
            if (!camera.TryGetCullingParameters(out var cameraParams :ScriptableCullingParameters ))
                continue;
            var cullingResults = context.Cull(ref cameraParams);


            // Configure the drawing settings
            var sortingSettings = new SortingSettings(camera)
            {
                criteria = SortingCriteria.CommonOpaque
            };
            var drawingSettings = new DrawingSettings( shaderPassName: new ShaderTagId( name: "Unlit"), sortingSettings);
            var filteringSettings = new FilteringSettings(RenderQueueRange.opaque);

            // Perform the draw
            context.DrawRenderers(cullingResults, ref drawingSettings, ref filteringSettings);
            context.DrawSkybox(camera);

            //Submit the draw
            context.Submit();
        }
    }
```

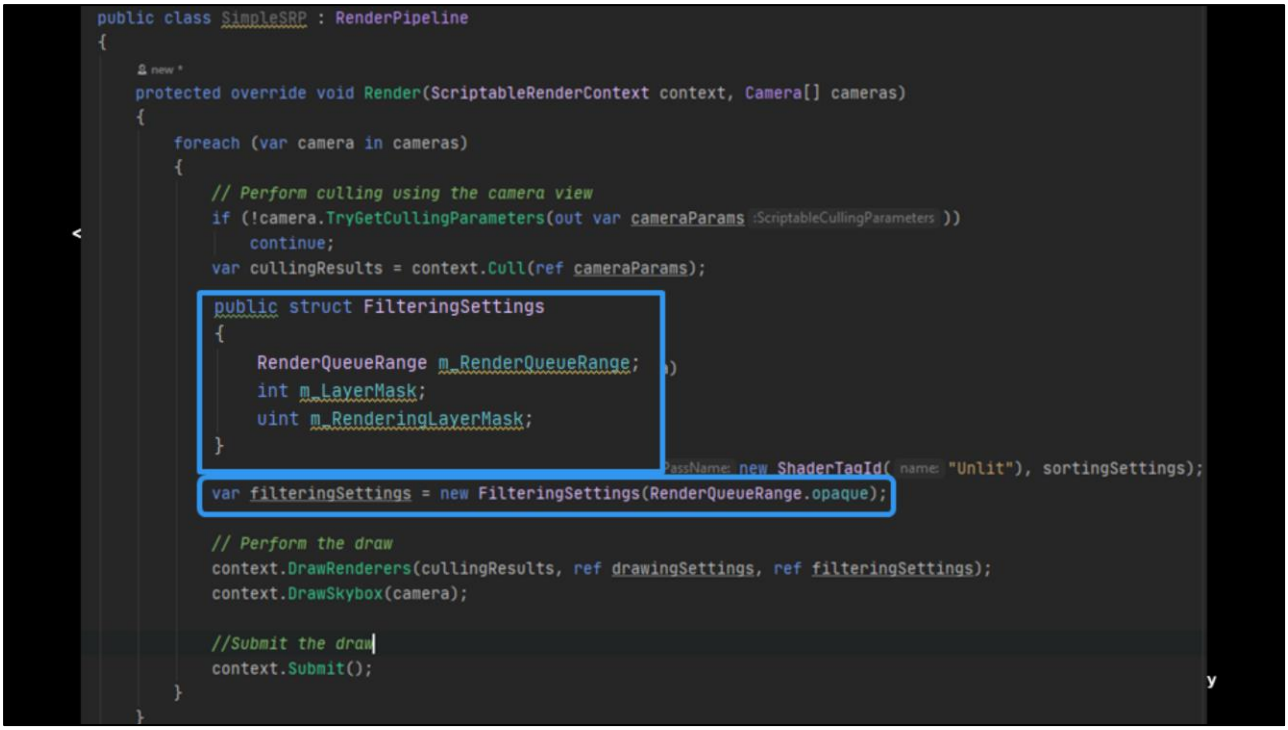
We covered the rendering context earlier and how it performs batch processing on the returned render nodes from the culling operation but how do we define these batches?

When drawing with SRP you need four things:



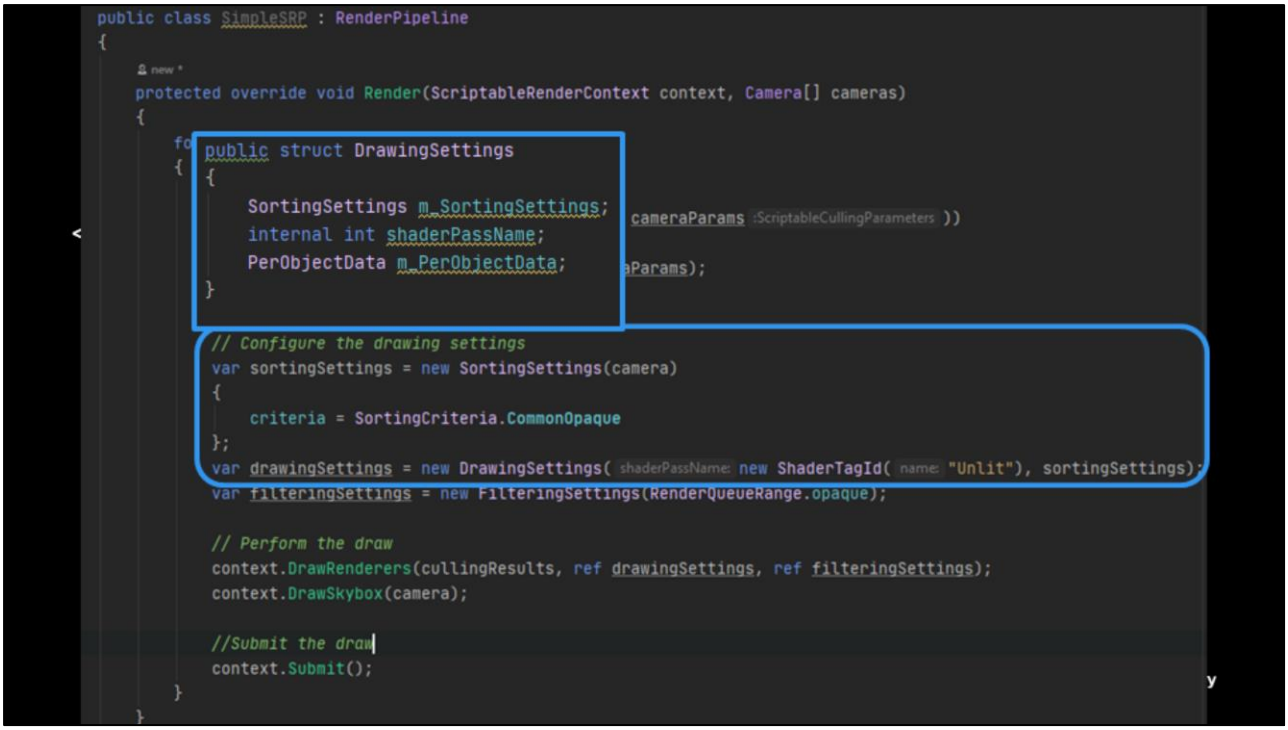
```
public class SimpleSRP : RenderPipeline
{
    new *
    protected override void Render(ScriptableRenderContext context, Camera[] cameras)
    {
        foreach (var camera in cameras)
        {
            // Perform culling using the camera view
            if (!camera.TryGetCullingParameters(out var cameraParams :ScriptableCullingParameters ))
                continue;
            var cullingResults = context.Cull(ref cameraParams);


            // Configure the drawing settings
            var sortingSettings = new SortingSettings(camera)
            {
                criteria = SortingCriteria.CommonOpaque
            };
            var drawingSettings = new DrawingSettings( shaderPassName: new ShaderTagId( name: "Unlit"), sortingSettings);
            var filteringSettings = new FilteringSettings(RenderQueueRange.opaque);

            // Perform the draw
            context.DrawRenderers(cullingResults, ref drawingSettings, ref filteringSettings);
            context.DrawSkybox(camera);

            //Submit the draw
            context.Submit();
        }
    }
```

1) The culling results handle we have covered



2) A filter to filter out these culling results - This will filter the rendernodes with some additional rules.

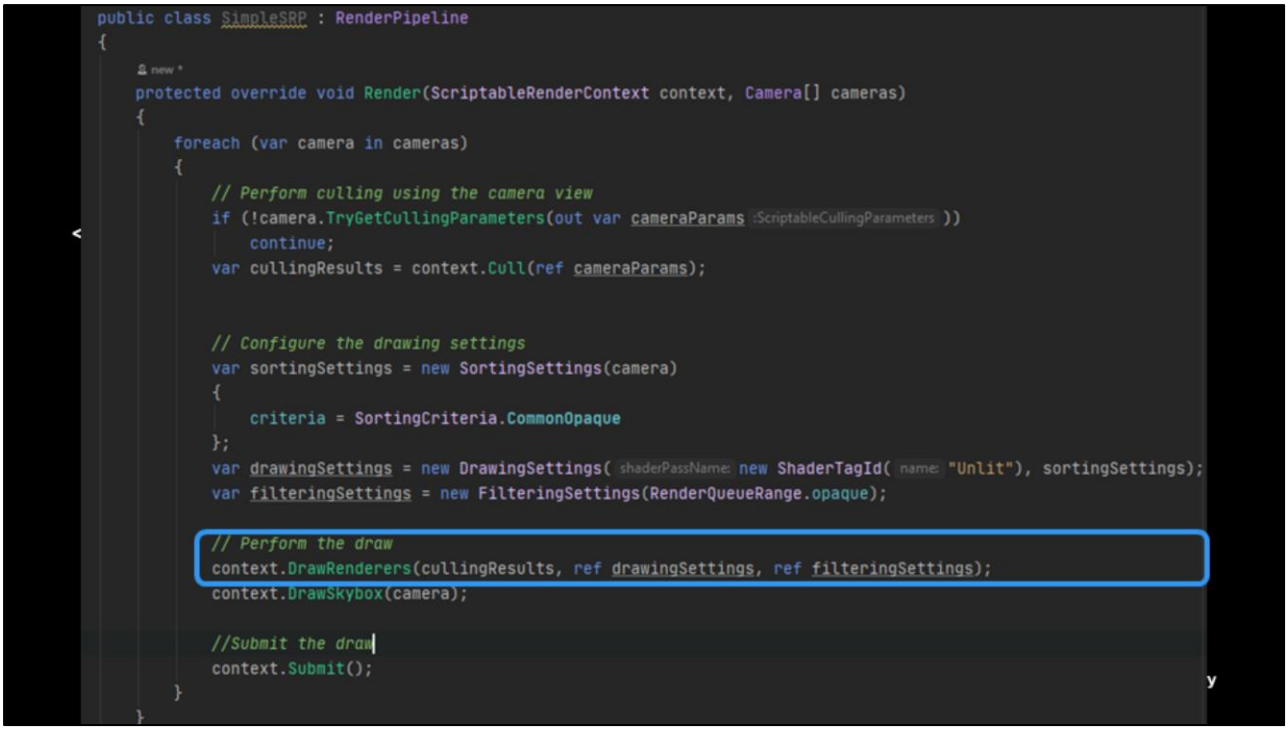
Filtering is generally done on two parameters:
1) ‘user assigned layer’
2) Render queue from the material. The render queue specifies if the object is opaque or transparent or ‘other’ user-implied name.



3) What drawing settings to use when rendering with the rendernodes that pass filtering

That is:

1) How the nodes should be sorted for rendering. You will want to render transparent back to front, but opaque the other way for example
2) The shaderpass to use in the shader attached to the rendernode
3) Any per object data you want to bind for the draw (previous frame matrix, light probe data etc)



```
public class SimpleSRP : RenderPipeline
{
    new *
    protected override void Render(ScriptableRenderContext context, Camera[] cameras)
    {
        foreach (var camera in cameras)
        {
            // Perform culling using the camera view
            if (!camera.TryGetCullingParameters(out var cameraParams :ScriptableCullingParameters ))
                continue;
            var cullingResults = context.Cull(ref cameraParams);


            // Configure the drawing settings
            var sortingSettings = new SortingSettings(camera)
            {
                criteria = SortingCriteria.CommonOpaque
            };
            var drawingSettings = new DrawingSettings( shaderPassName: new ShaderTagId( name: "Unlit"), sortingSettings);
            var filteringSettings = new FilteringSettings(RenderQueueRange.opaque);

            // Perform the draw
            context.DrawRenderers(cullingResults, ref drawingSettings, ref filteringSettings);
            context.DrawSkybox(camera);

            //Submit the draw
            context.Submit();
        }
    }
```

4) The draw call to be added to the context.

Just to reinforce - SRP does not have access to the individual meshes, transforms, or rendernodes - this allows us to perform ‘batching specific’ operations in the backend for the selected draw configuration.

In Unity we have taken the approach of ‘broad phase culling’, then fine grained filtering.

So take for an example a scene that has objects sharing both opaque and transparent sub objects, like this car here - it has windows and panels.

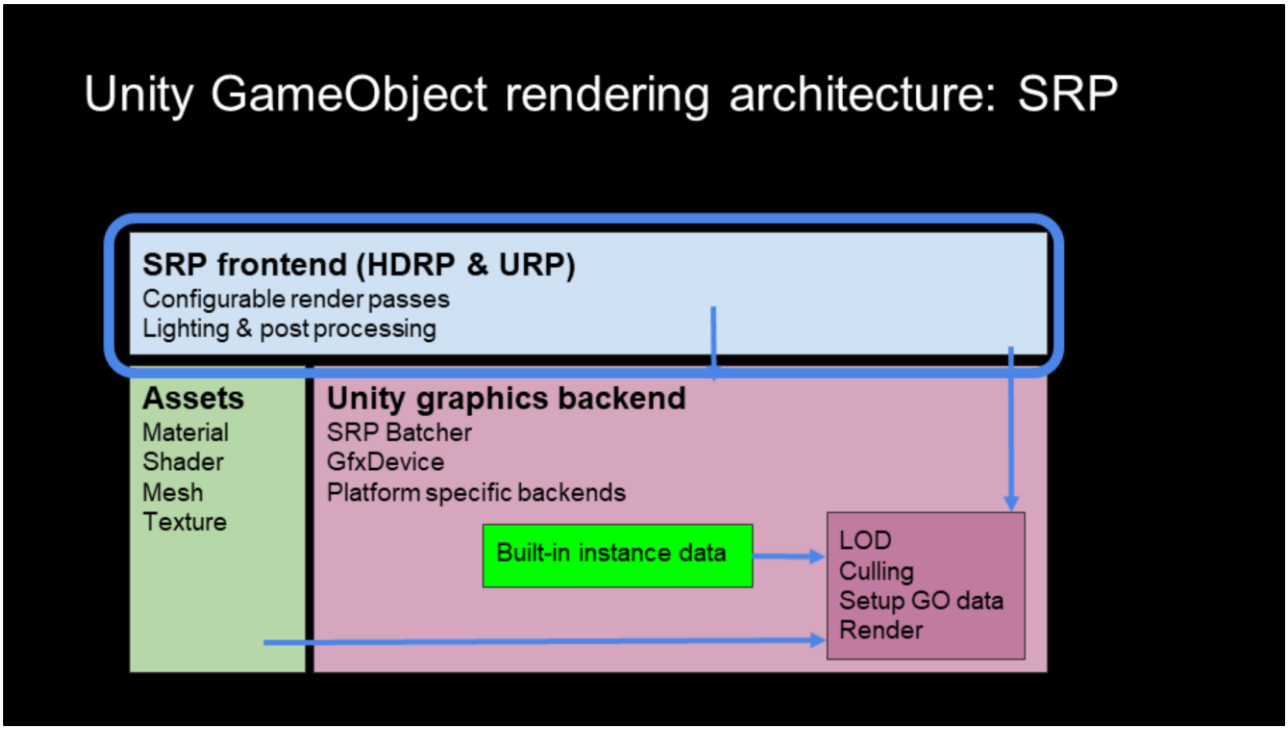
In Unity culling will generally return all these meshes as ‘having passed visibility’ if they are within the view.

We will then use draw filtering to select which gets rendered for each DrawRenderers.

This strikes the right balance for us between performance and flexibility.

If we wanted to render opaques to prime a z-buffer, then transparents we would likely render opaques front to back then transparents back to front as a separate pass.

We are also adding in the concept of ‘prefiltered draw lists’ that run off the main thread to pre filter our cull results for improved draw time performance and to help us go wider.



It’s often useful to have low level access to the render context - but it’s pretty raw and not fully feature-rich.

Let's examine the SRP frontend where we have feature-rich tooling

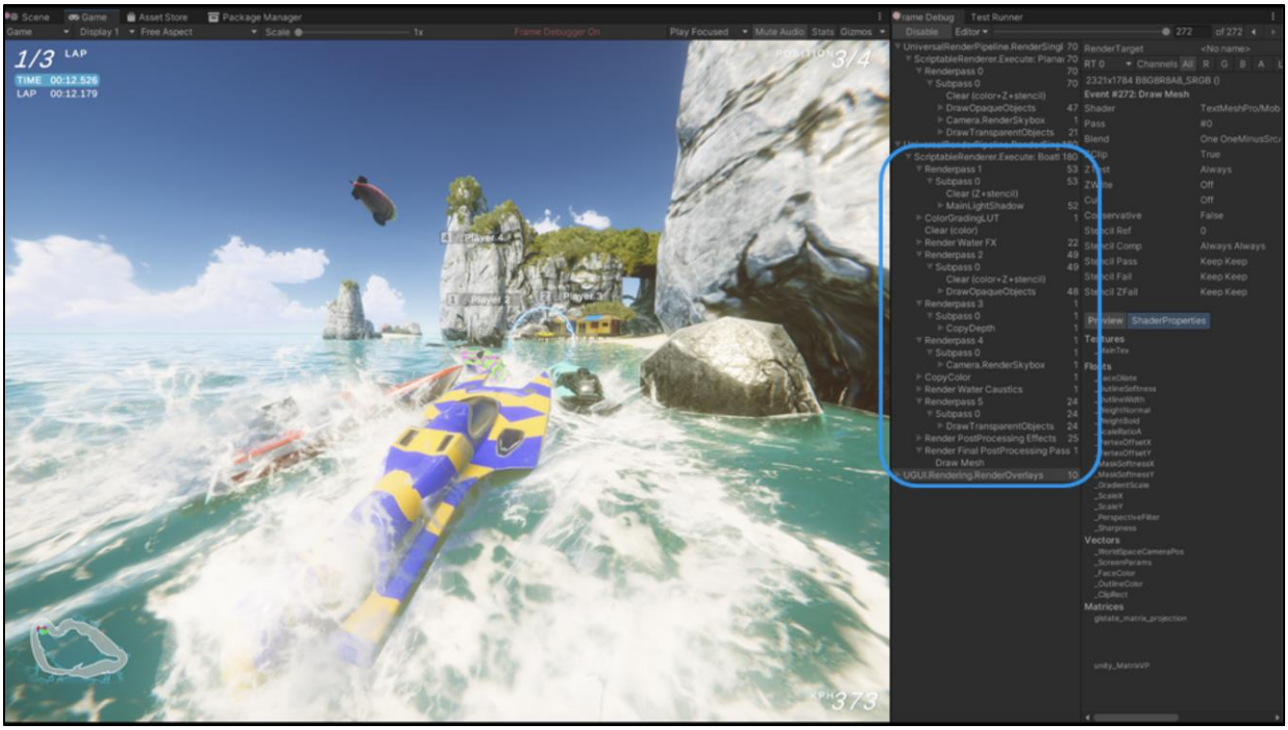
We wanted to provide more functionality out of the box for people writing SRPs and some good libraries that would make it easier to ‘do the right thing’. This serves us at Unity by giving us the tools we need to build URP and HDRP, but also helps users write their own SRPs.

SRP Toolbox

SRP frontend (HDRP & URP)
Render Graph
Render Pass API
Improved Batch operations
Improved Batching
Material overrides
Volume System
Texture Atlas System
AND MORE!

On top of the rendercontext in our c# layer, where we built this out. These tools have a focus on ‘performance by default’ and ease of use.

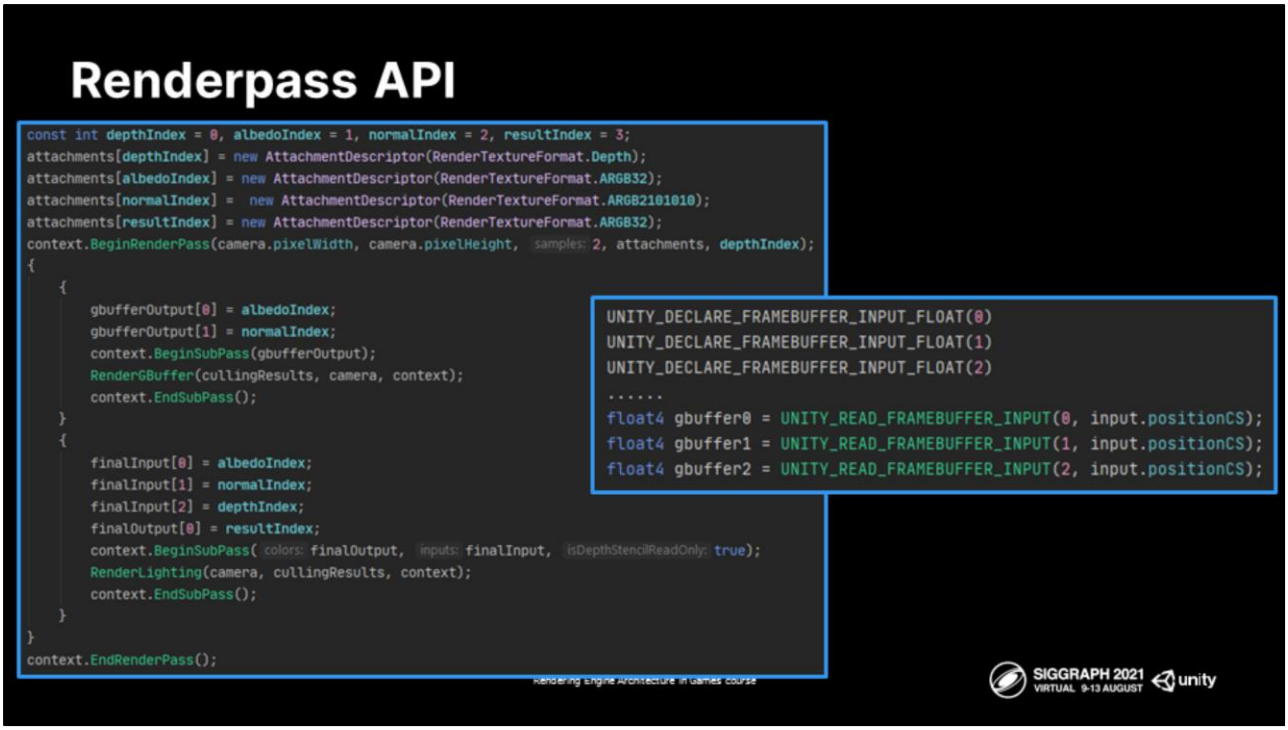
As you can see we have a number of tools - but we will only take a look at some of the bigger ones.



Let's talk a little about mobiles… mobile rendering is generally executed on tile based renderers… a side effect of this is that they are very sensitive to resolving GPU memory back to main memory - that is if you want to render to texture then bind that as a texture for use as a read target in another pass, you need to read that back before you can bind it.

It’s a bandwidth intensive and slow operation and many mobile games suffer performance issues by not being able to ‘stay on tile’ as much as possible. Many engine developers have tried porting tech to mobile without paying close attention to this and run into problems - we’ve even made this mistake a number of times with features we have developed.

We wanted to make this MUCH easier to do properly and not end up in a bad path. So we developed a userland renderpass API.

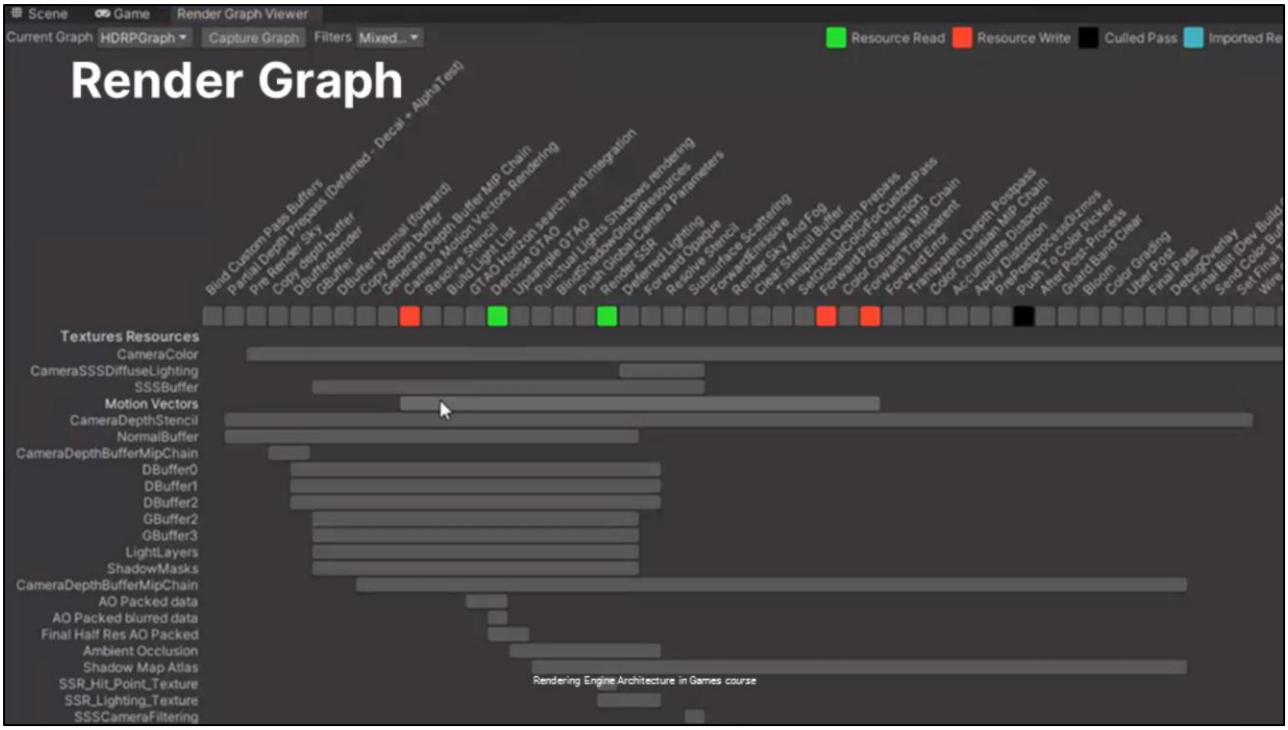


The way this works is that on the context you define when a render pass starts using the exposed API along with the attachments that get used in that pass. For reach sub-pass you can then use these attachments how you would see fit.

Here we have a simple deferred renderer where one pass fills a g-buffer, and the second pass performs lighting.

If you look at the shader code here on the right you can see we provide some macros for reading from these frame buffer inputs. On mobile these will become framebuffer fetch operations. On devices that don't support this (like most PC's) we will behind the scenes bind this as a render texture for sampling.

We provide one API that benefits mobiles heavily but still works *well* on other devices. This allows you to write code once that has good performance characteristics across the range of hardware unity supports.



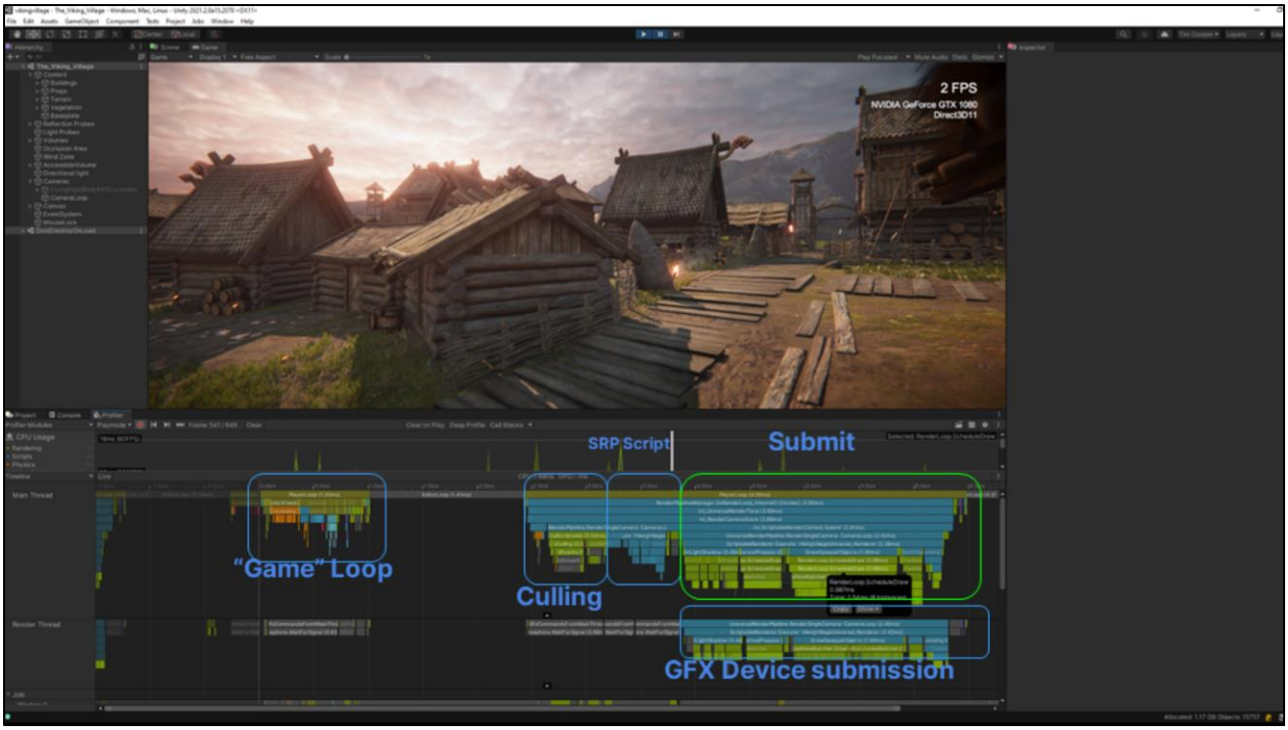
Renderpass is great for situations where it's possible to stay on tile and share the same pixel data but as we knows that's not all that's needed in rendering - you'll often need to read from neighboring pixels, use a texture from a previous frame, or use a texture in a location quite far from it's creation.

Now there are a number of ways to manage this - manual texture management, some texture pooling class or similar.

We wanted to be more explicit about tracking graphics resources across the frame so we built a rendergraph that lives (once again) on top of our rendercontext but interacts with it.

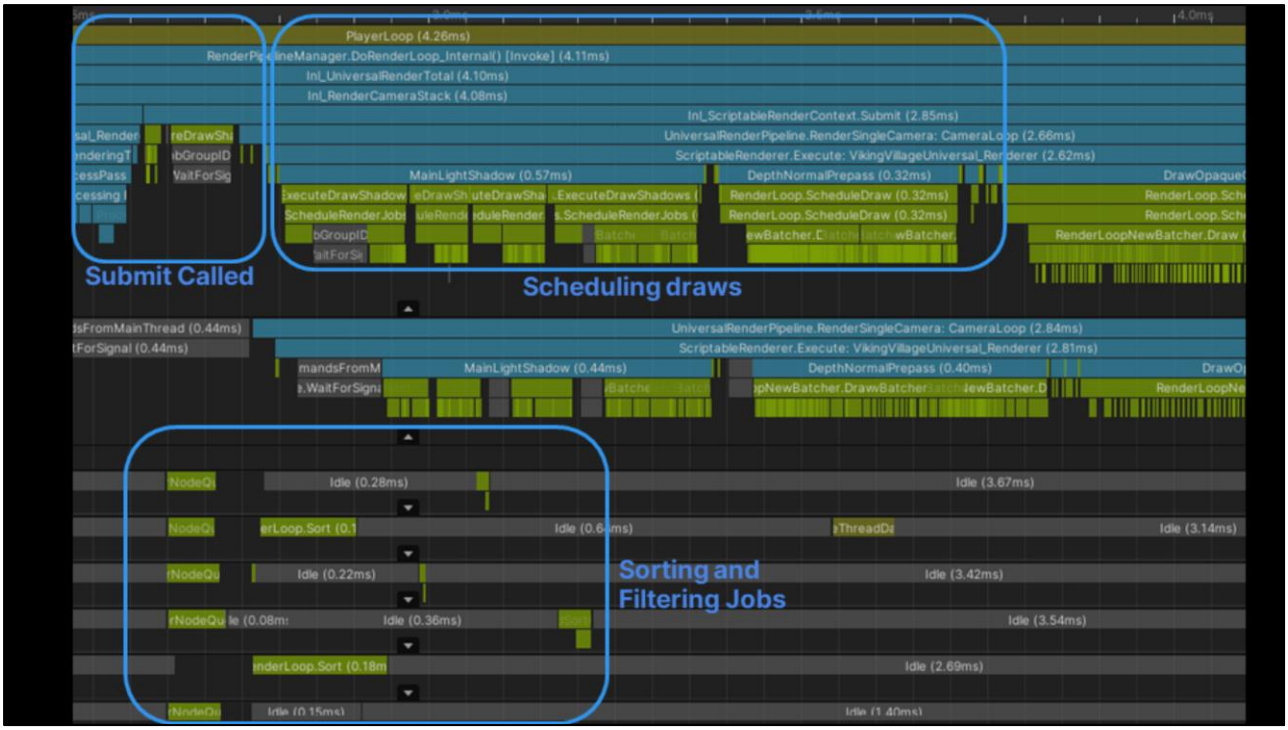
By using the rendergraph, it's possible for us to lower our high memory usage watermark and ensure that we are not using more memory than you need to.

Our goal is to combine our frame graph and render pass API to have he best of both worlds and effects that can scale nicely across the range of devices.



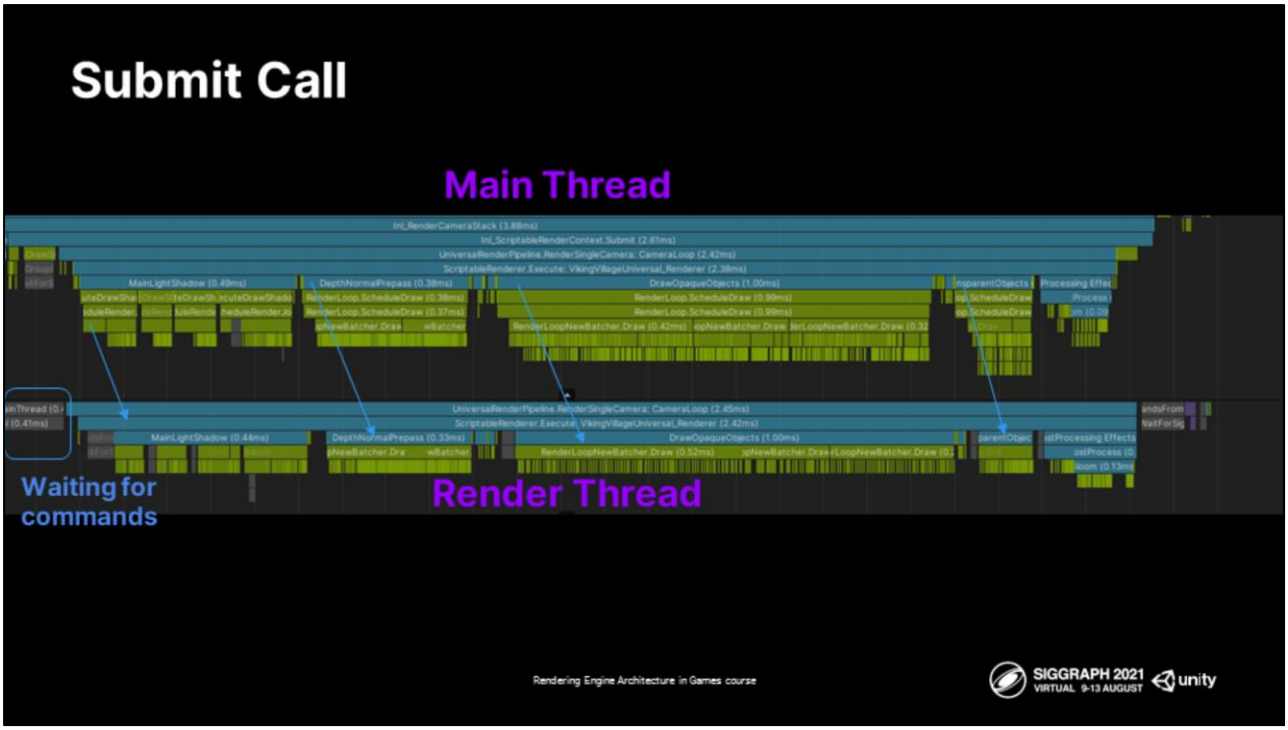
And we are back to our higher level frame - We use all of these tools to build out a set of commands to inject into the rendercontext - that is the goal of the script callback - to define the rendering flow.

But to kick off the device submission we need to call submit on the context. This allows the author of the pipeline to control when work is kicked off. You can be smart about this to reduce bubbles and to manage submission.



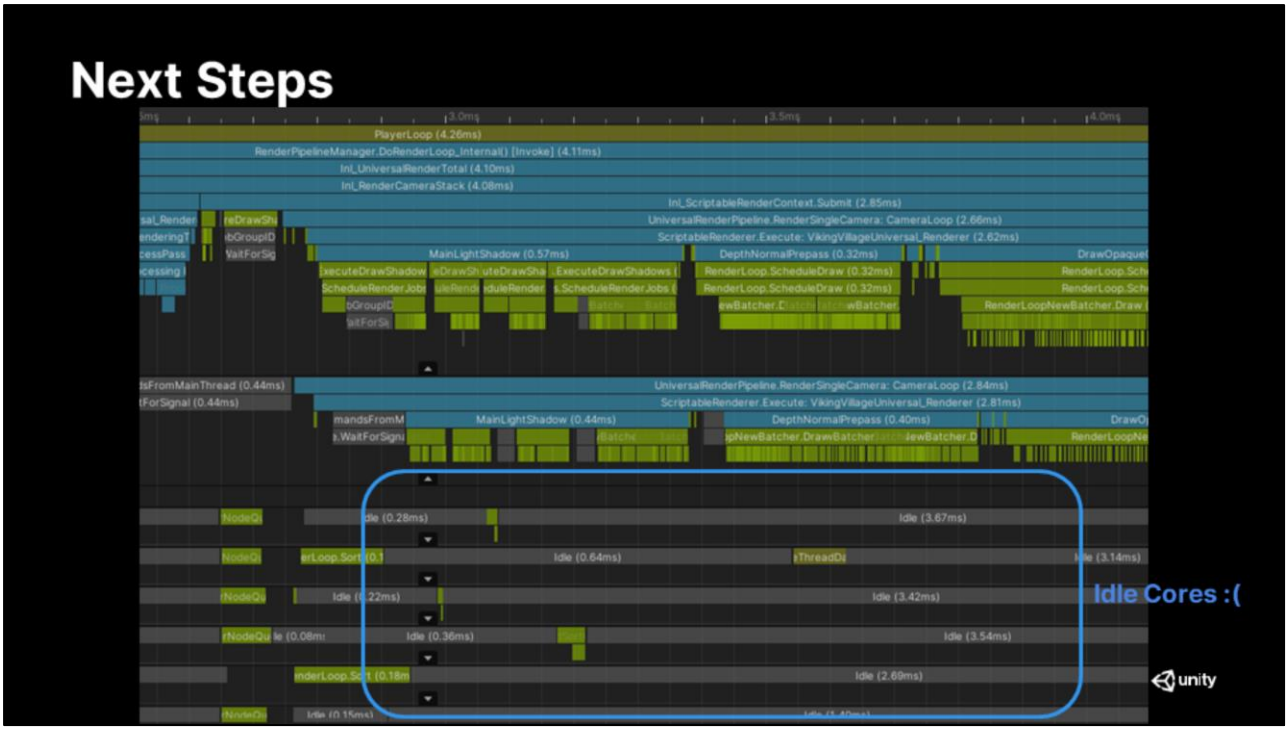
When submit is called we kickoff a number of jobs - and return control flow back to the main thread. These jobs: execute the filter, and build a list of nodes to render then pass that down as consumable to the GFX device layer.

You may notice that we have a large block in the timeline here associated with the CPU cost of rendering, Sebastian will talk about how we have a new architecture here that reduces this processing due to having persistent GPU state which allows us to reduce the number of rendernodes we have to process and to scale the work to multiple threads"



Commands are passed to the render thread in a producer / consumer model where the graphics device backend can consume the commands in a platform specific way. Some backends directly submit the commands. Others use native graphics jobs to process the command stream.

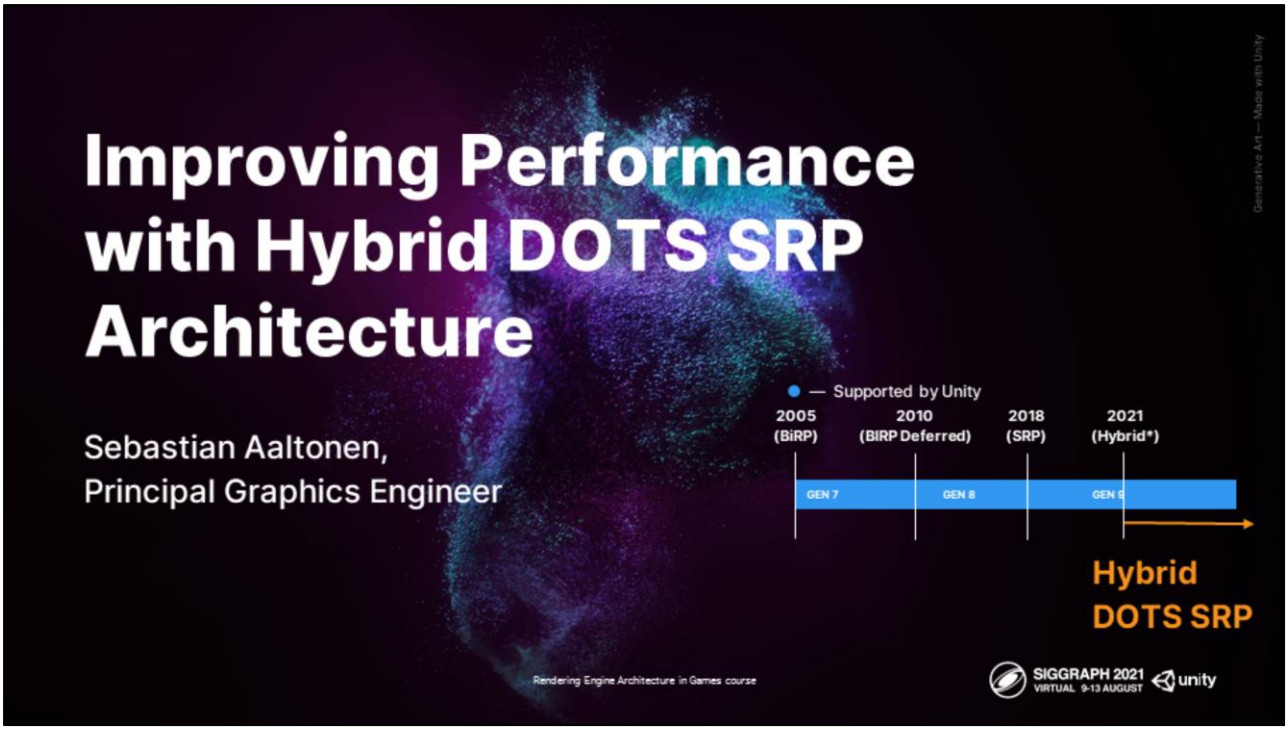
This renderthread is allowed to overlap the next frame - so the mainthread can start processing right when all submits are completed.



I've talked quite a while about how we have build and organised rendering with the SRP layer at unity but it doesn't solve all the issues we want to address in our rendering stack. You may remember that the SRP is a batch processing system - working on groups of nodes rather than individual nodes.

Our next step is to go much wider.

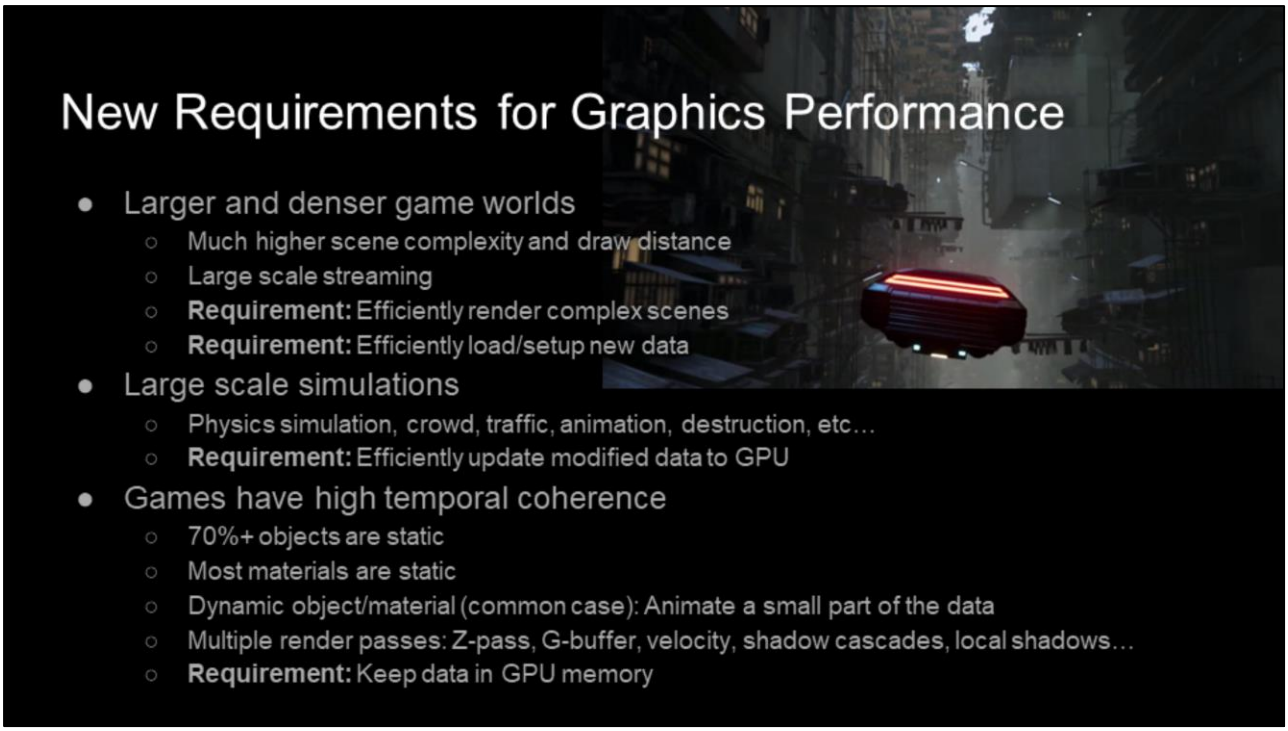
I will pass over to Sebastian to talk about the architecture we have in this area and the decisions we are making.



Hi, I am Sebastian Aaltonen. I am working as a principal graphics engineer at Unity.

Today I am going to be talking about our Hybrid DOTS SRP rendering architecture, which is combining these two new pieces of technology, DOTS and SRP, together.

This new technology no longer supports the old built-in rendering pipeline (BiRP).

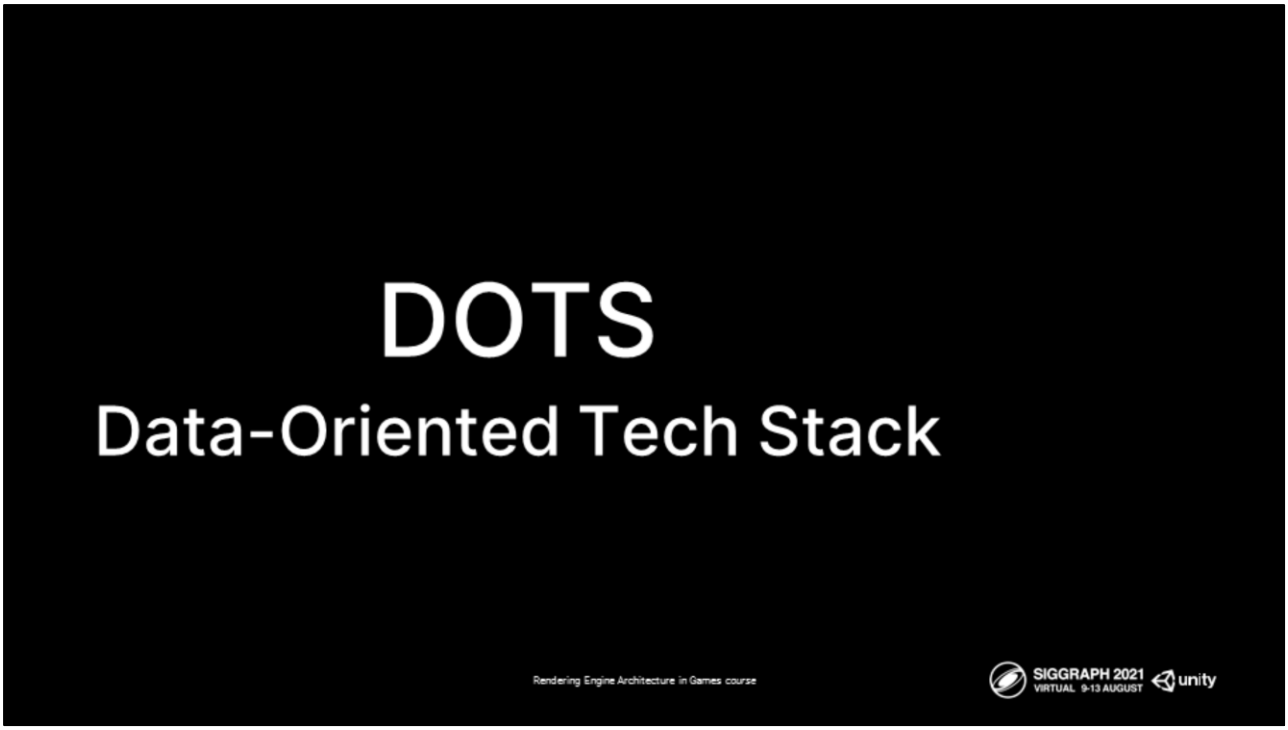


Let’s start with the reasons why we need changes in our technologies.

Modern games are becoming more realistic and demands towards larger and more dense worlds are growing. Renderer needs to efficiently push lots of draw calls and data streaming needs to be highly efficient.

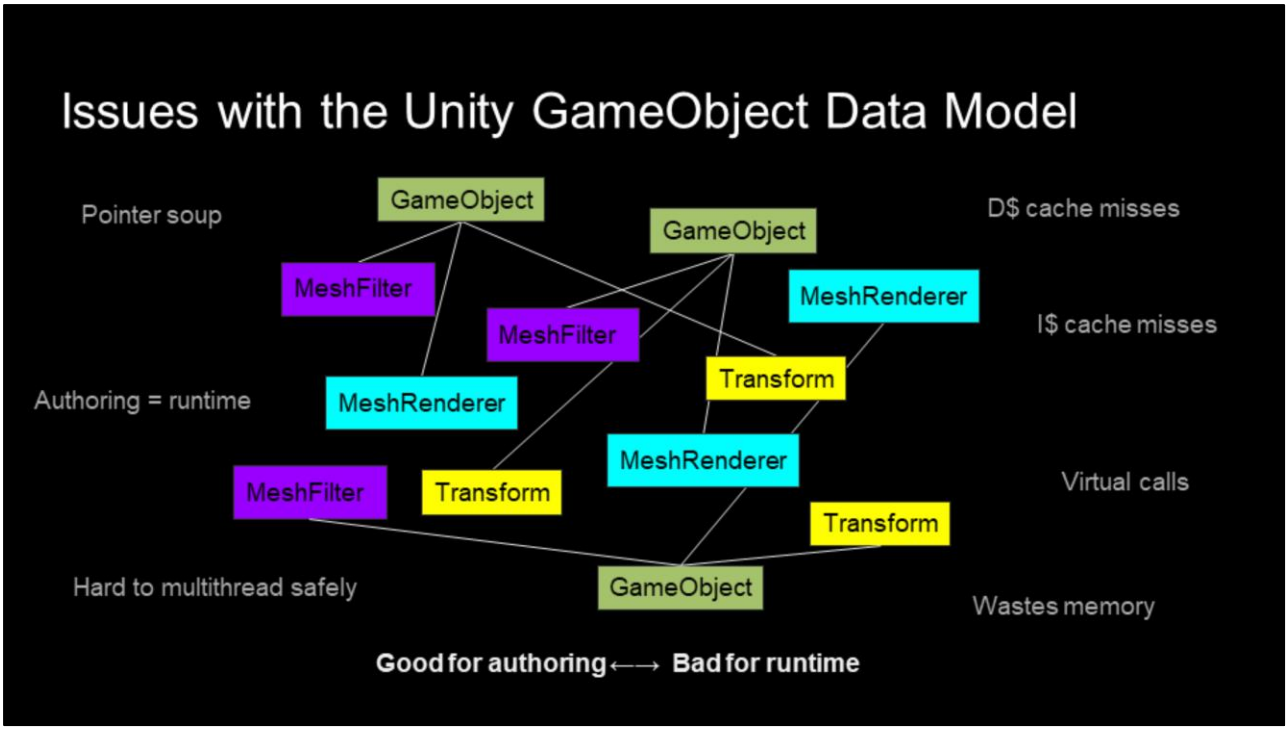
The growth in CPU core counts means more simulation and more dynamic behavior. Our data structures and threading models need to handle this efficiently.

Games tend to have a high degree of temporal coherency. Even though we have more CPU cores to drive simulation, the increases in density, world size and draw distances mean that the huge majority of drawn objects will remain unchanged from the previous frame. Modern renderers are also rendering the same object to multiple render passes every frame. We don’t want to upload the same object data again and again for every draw call.



DOTS architecture is the cornerstone of Unity's new improved runtime.

Let's delve to the reasons why chose to build DOTS and what it provides us.

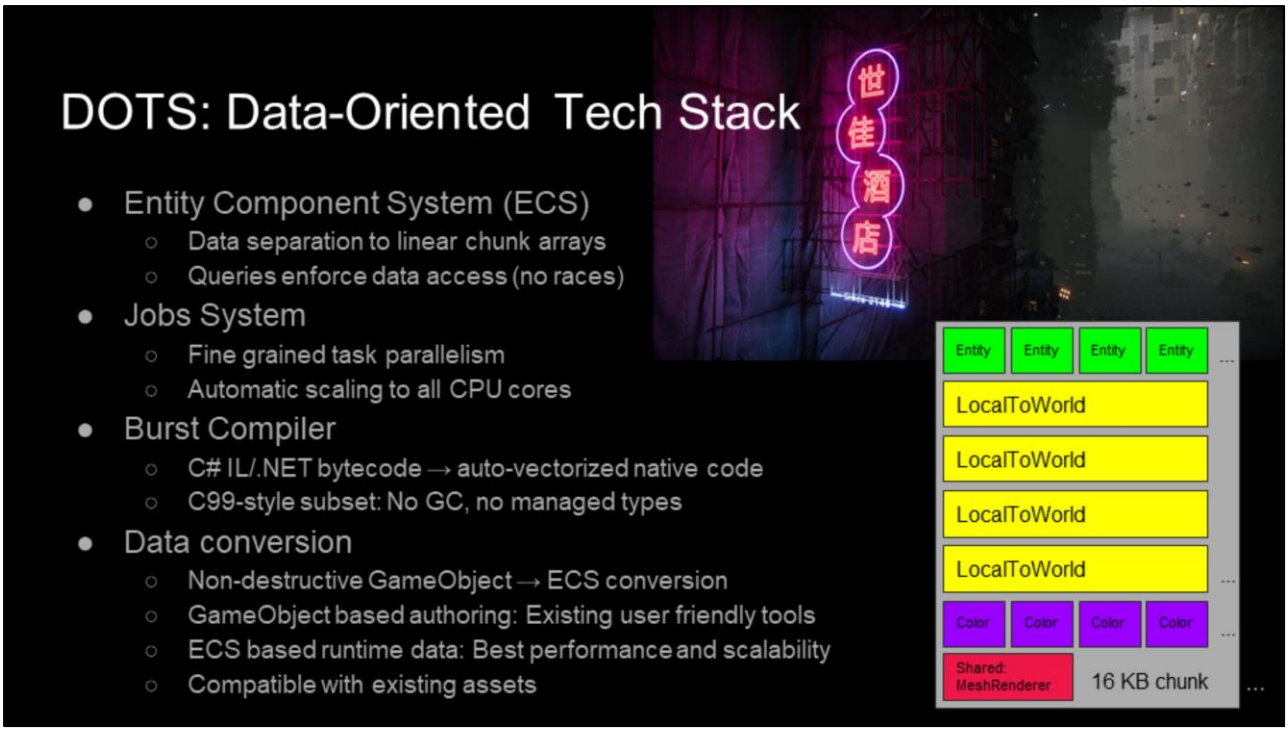


Unity is using a classic object oriented data model with deep class hierarchies. This kind of data model is great for content authoring and prototyping, but the runtime performance is lacking.

GameObjects form a complex pointer soup style data structure, where it's hard to restrict access. Fat objects and pointer chasing adds a lot of cache misses, and virtual calls make it difficult to reason about data processing.

It's hard to multithread a data model like this safely. So we can't simply scale the performance up by adding more threads.

In order to meet the increasing demands of high end developers, we must find a better data model for the runtime, designed ground up for modern multicore CPU architectures.



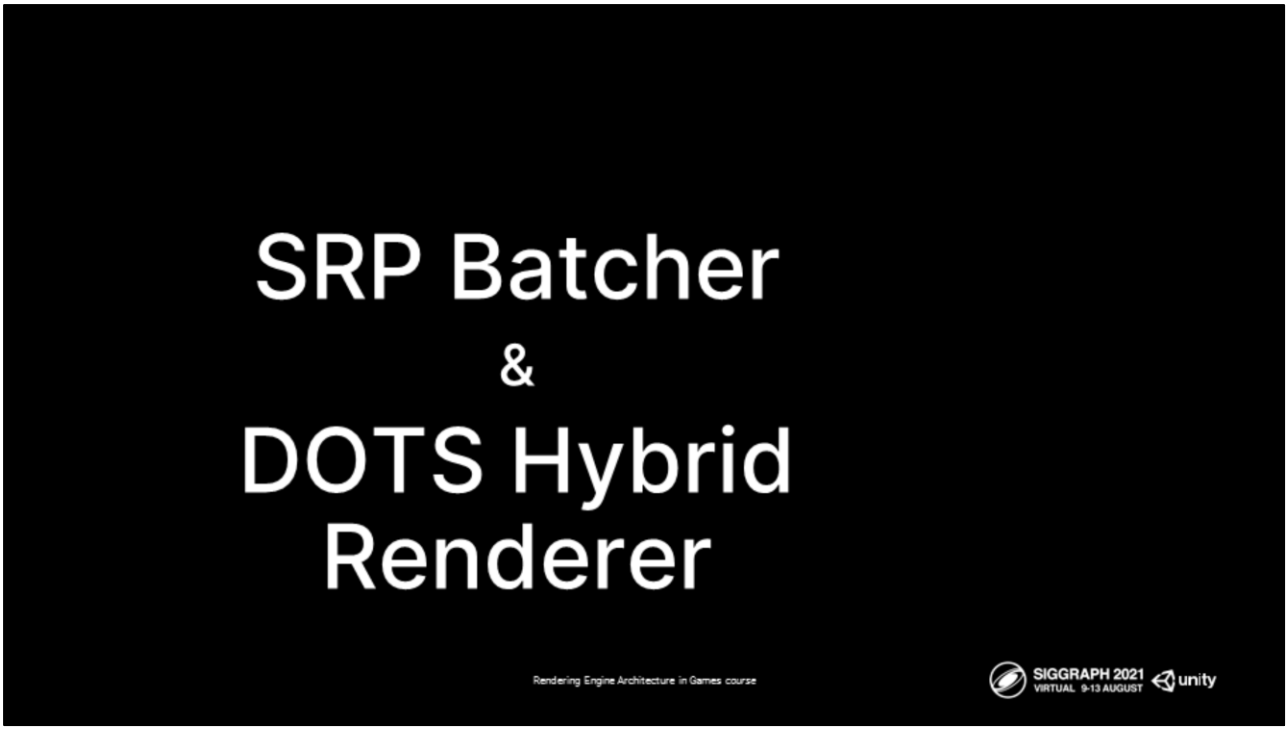
Our solution for meeting the runtime performance goals is DOTS. This is a brand new data-oriented tech stack with a new data model, a new threading model and a new compiler.

The DOTS ECS forms the foundation of the new data model. It separates components to chunks of linear data based on their archetype, and offers an efficient query interface for processing data safely without concerns of data races.

The DOTS Jobs System provides fine grained task parallelism, scaling to all the available CPU cores on mobile, console and desktop hardware.

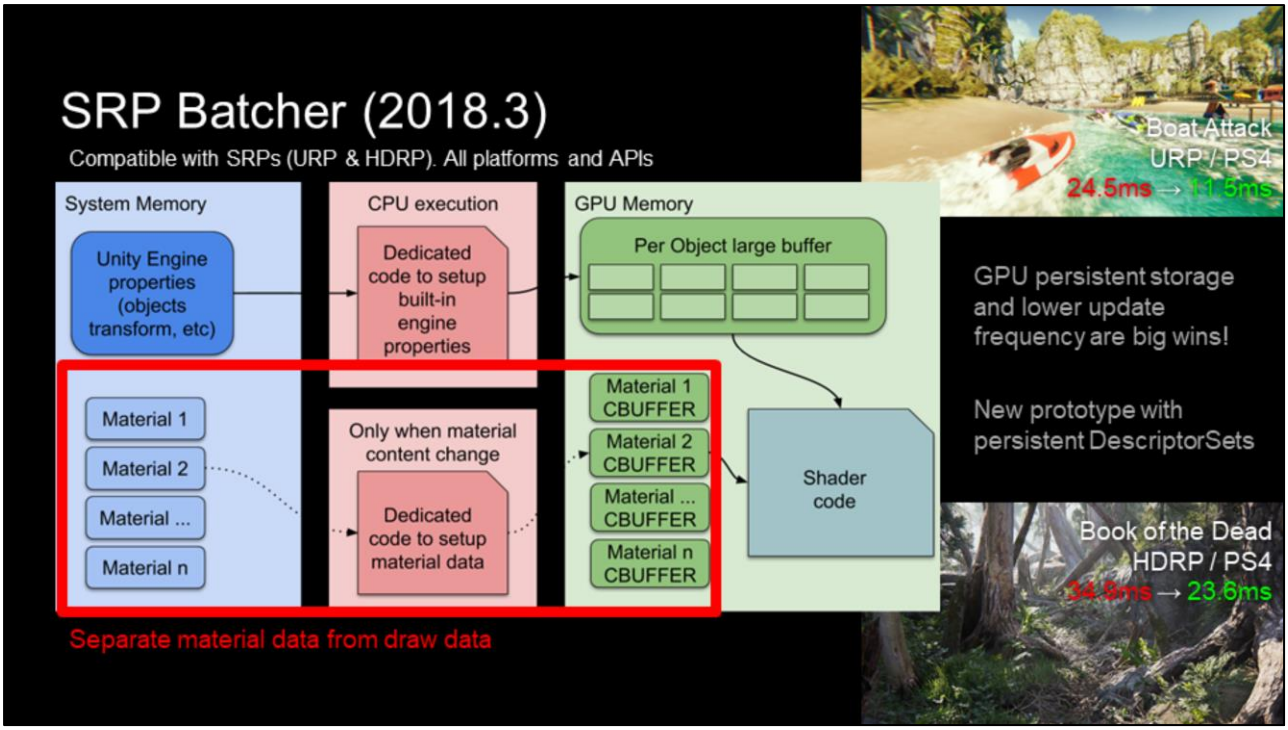
The Burst Compiler is a new C# auto-vectorizing compiler covering a C99 subset of the C# language. No garbage collection or managed types.

DOTS is using a non-destructive data conversion process. Authoring is done using the existing assets and GameObject tools. Iterative conversion from GameObjects to DOTS ECS in running in the editor in the background, resulting in optimal runtime data layout and performance. While retaining good iteration time and improving editor responsiveness in large scenes.



Now, let’s shift our focus to the graphics side.

I will introduce the graphics technologies we use to meet our performance needs.

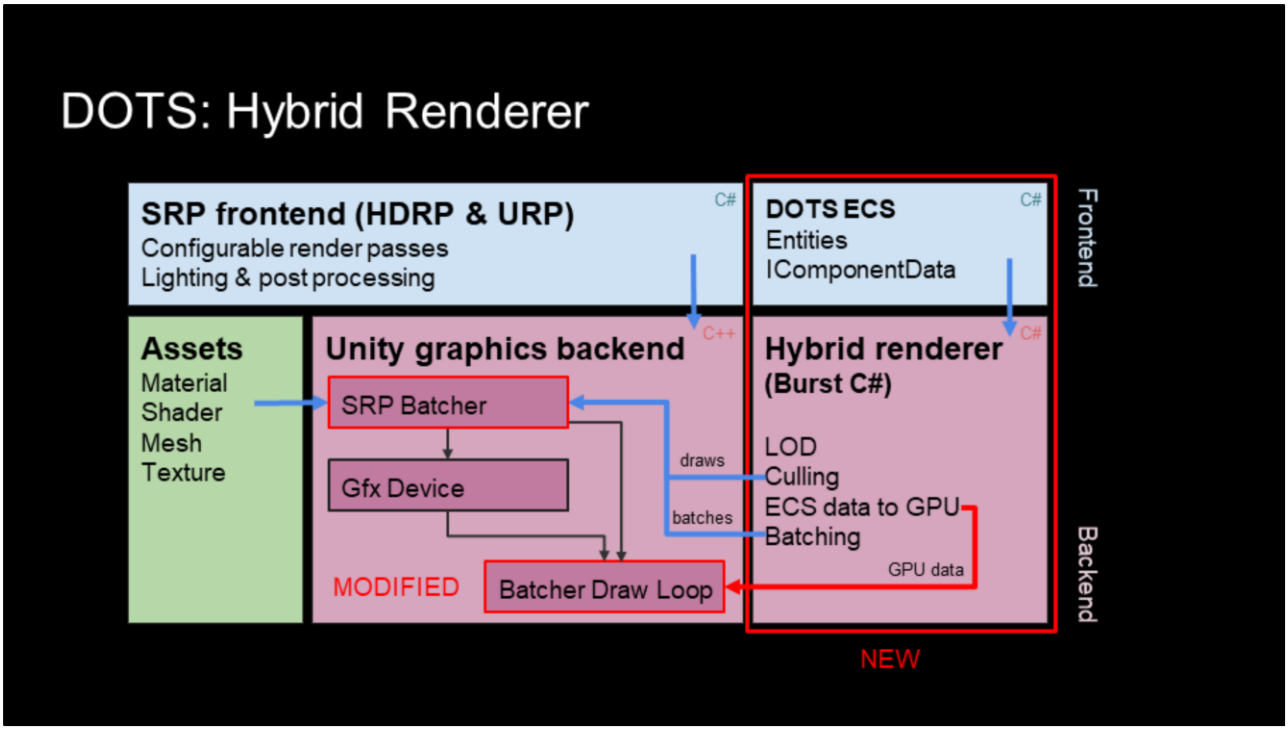


SRP Batcher was originally released two years ago. The goal was to improve the performance of GameObject rendering with the scriptable pipelines.

This was achieved by two means:
First: The SRP Batcher is building batches of compatible renderers to avoid the costly shader changes
Second: The SRP Batcher is fully separating the material storage and upload from the high frequency draw loop. The material data is now persistent in the GPU memory, and uploaded only when changed.

These changes make the GameObject rendering much faster. In the best cases, we see the total frame time to drop to roughly half of the original with SRP Batcher.

These results gave us increased motivation in finding more performance wins by making more data GPU persistent. We have a new prototype with persistent DescriptorSets, which is a new feature in Vulkan, DX12 and Metal, albeit the naming varies across APIs. This feature allows us to persistently bind all the descriptors together ahead of time, and change the material at runtime with a single low level API call, resulting in a big performance win.



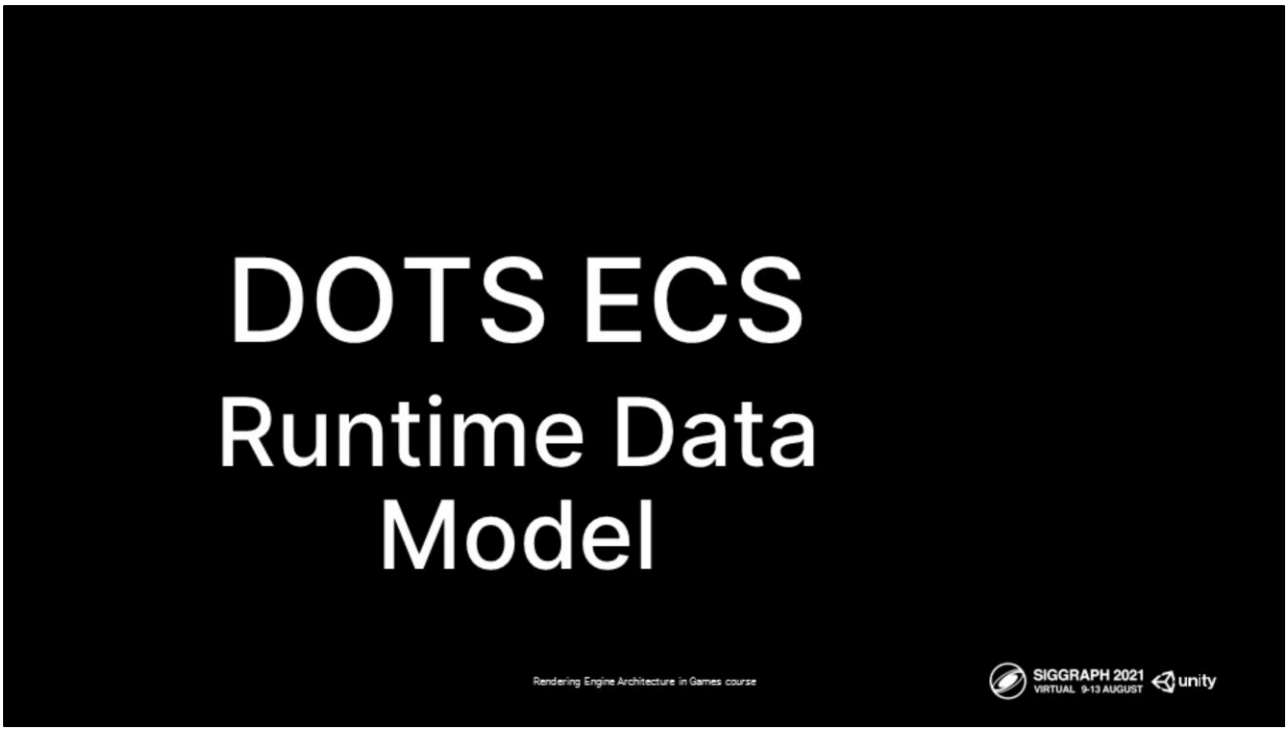
Now, let's talk about the hybrid renderer: The Hybrid Renderer is a new technology that connects DOTS ECS to the Unity's existing rendering architecture.

This allows us to keep the existing graphics authoring tools in place, while replacing the runtime with a faster one. The SRP frontend stays the same, we use the existing assets and big parts of the existing graphics backend.

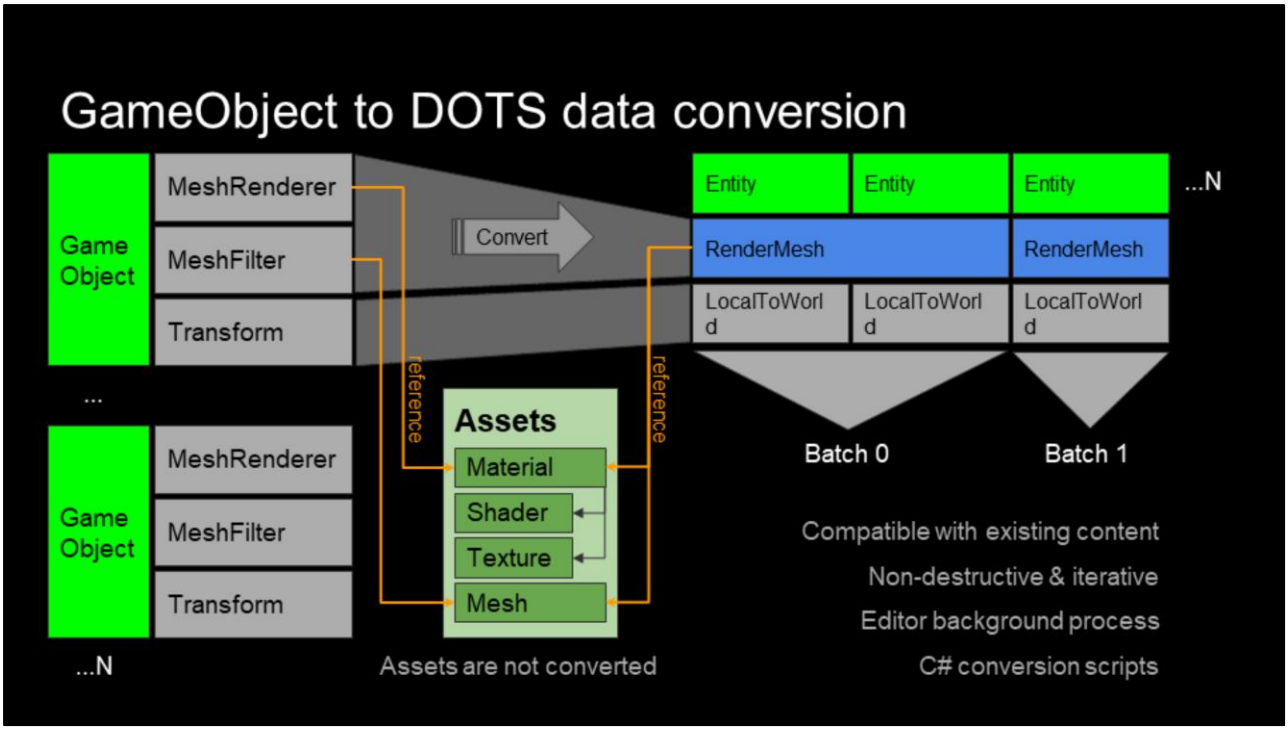
The SRP Batcher exposes a new C# API for submitting the batches and the draw data from external sources.

The Hybrid Renderer collects the DOTS ECS data, builds persistent batches and calculates the visibility. Draws and Batches go to the SRP Batcher, while the ECS data is uploaded directly to GPU, avoiding the slow existing data pipelines.

Most of the Hybrid Renderer code is written as Burst C# jobs, enabling SIMD codegen and allowing it to scale perfectly to any amount of cores.



Now, let’s dig deeper to the DOTS ECS data model to get good understanding how the CPU data is processed.



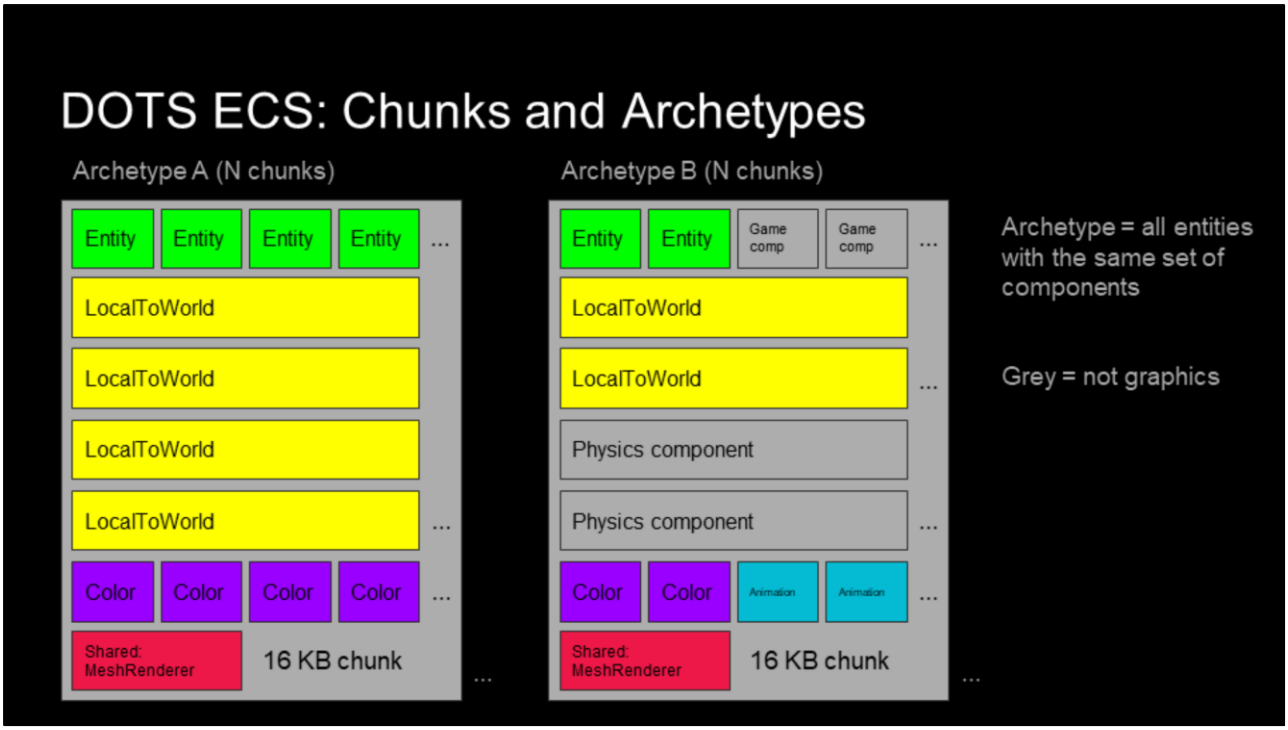
Let’s start with the DOTS conversion process. We convert existing authoring GameObject data to efficient runtime data layout based on DOTS ECS entities and components.

Conversion is non-destructive and iterative. It runs in the editor as a background process.

Sub-scenes in the editor can be open or closed for editing. Closed sub-scenes are converted to ECS entities, while open sub-scenes are represented solely as GameObjects. This keeps the editor responsive when editing large game worlds.

The component set conversion scripts are written in C#. The user can write their own conversion scripts for their custom GameObject component sets.

Assets are not converted. The asset references are simply copied to the DOTS components.

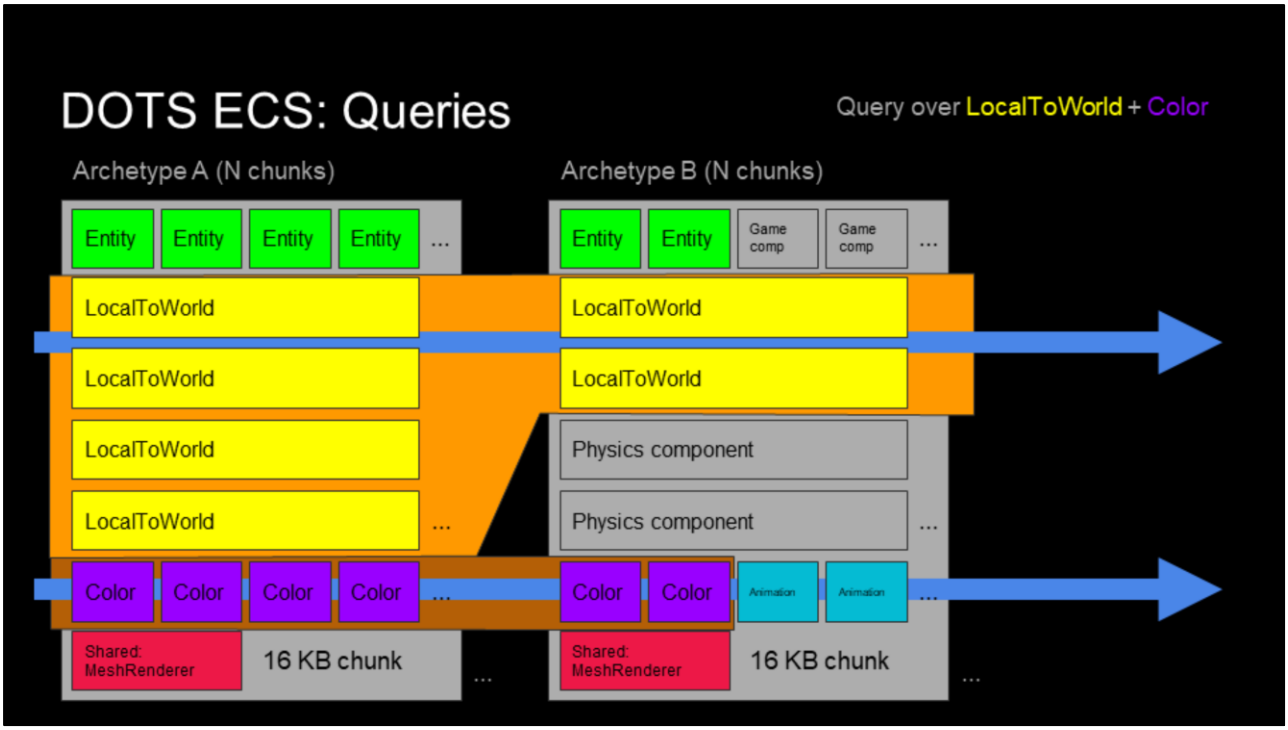


DOTS ECS has a chunk based data model. Entities are split to archetypes based on their component set. Entities with different archetype are split to different chunks. DOTS uses fixed size 16 KB chunks. This makes memory allocation trivial.

Each DOTS chunk contains an array per component. Each array in the chunk has the same amount of elements, each matching the same entity. There's a special array for entity identifiers that is present in every chunk. The combined size of the arrays is always 16 KB, minus metadata and alignment of course.

On the left hand side, you see a chunk of archetype A: This archetype contains the following components: LocalToWorld matrix, material Color override component and a MeshRenderer shared component which is shared between all entities in this chunk.

On the right hand side, you see a chunk of archetype B: This archetype has some additional components related to animation, gameplay and physics. Chunks of this archetype fit less entities as the components take more space.

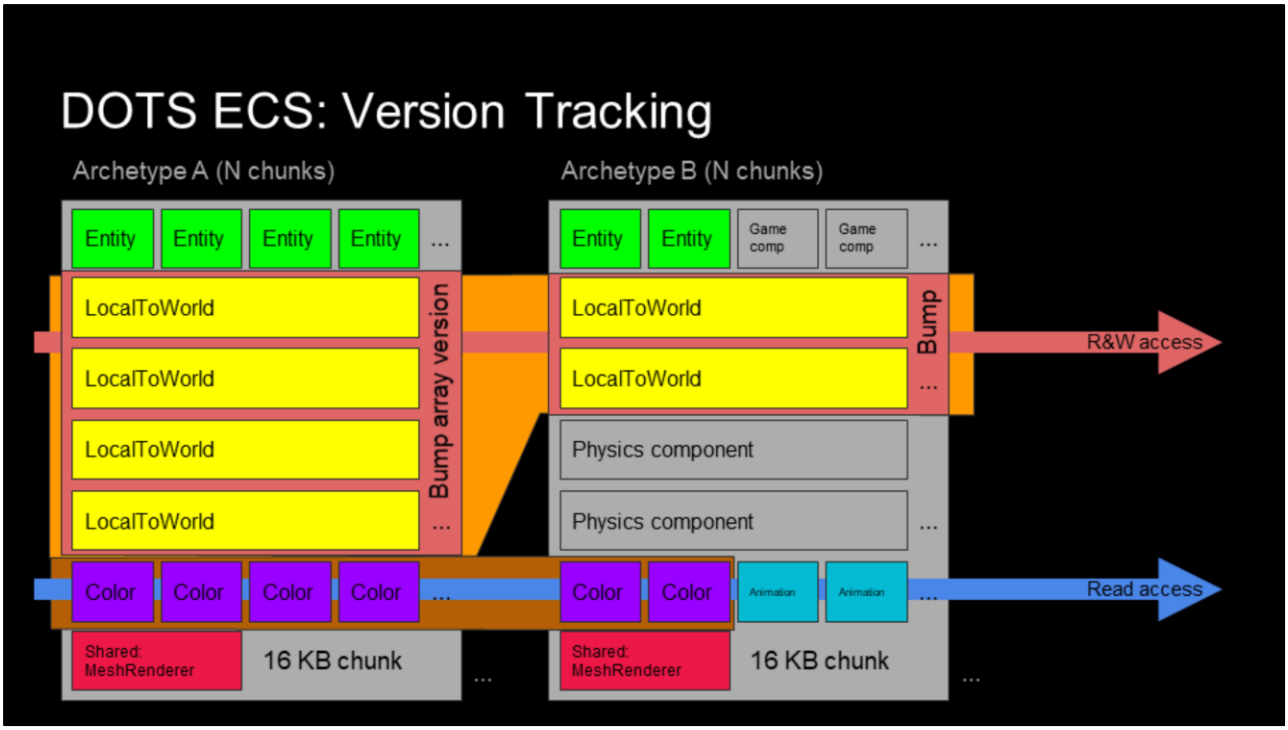


The main way of processing DOTS data is ECS queries.

A DOTS query is an inner-join query over a set of components. Each archetype is matched against the query component set. If the archetype contains all of the query components, all chunks of that archetype are included in the query.

The query processes though the selected component arrays of each chunk. Only the components in the query are touched. Other unrelated data is not touched, resulting in almost perfect cache line efficiency.

Since the chunk component arrays are contiguous, the Burst compiler can efficiently auto-vectorize the inner iteration loops.



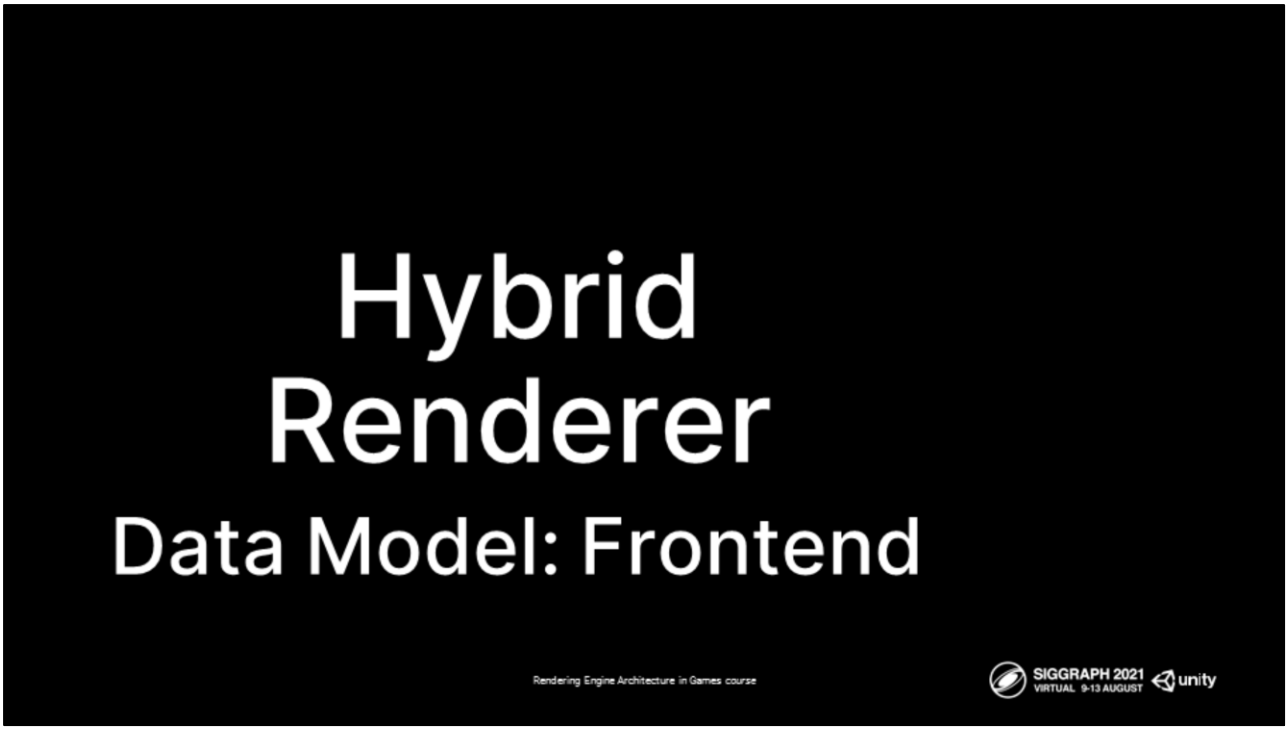
Change tracking is a hard problem to solve efficiently and robustly. It's easier to make it a responsibility of the content creators. They define what is static and what is dynamic. Unity doesn't want to add complexity to content creators.

Fortunately the DOTS architecture has a nice solution for this problem. DOTS queries define read and write access properties to each component included in the query. This helps with scheduling, as multiple reads of the same data are race free and can be executed concurrently.

Since the write access is tracked explicitly and misuse is guarded by the compiler, we know which component arrays in each chunk were potentially modified.

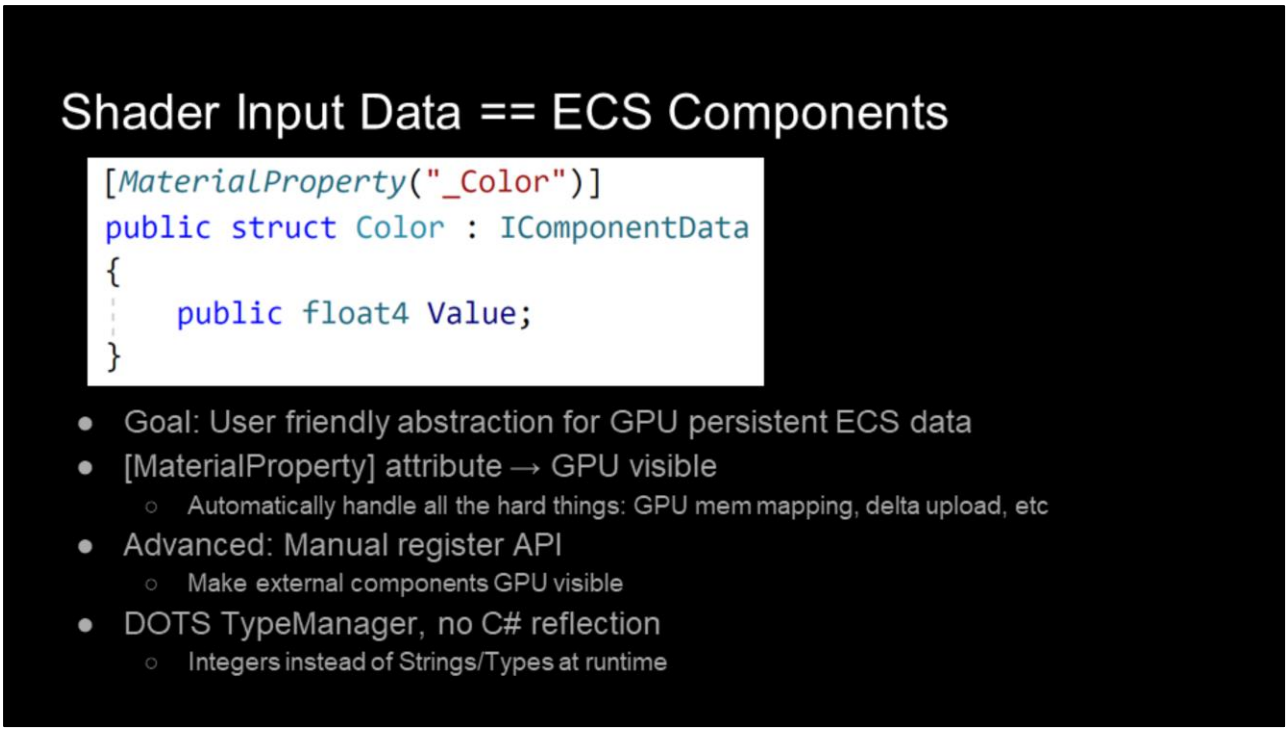
To implement a "free" data version tracking system, we add a version number to each component array in each chunk. When write access is requested, the array version number is bumped to the global version counter, which is monotonically increasing. Systems store previously seen global version counter value. This value can be used as version change filter in future queries, to limit the query over chunks that have changed since the system saw them previously. This change tracking system is more robust than dirty flags and doesn't require any additional bookkeeping.

We rely heavily on DOTS change tracking in the hybrid renderer.



Now that you understand the basics of DOTS data model, let’s discuss about the hybrid renderer.

Let’s start with the user facing frontend.

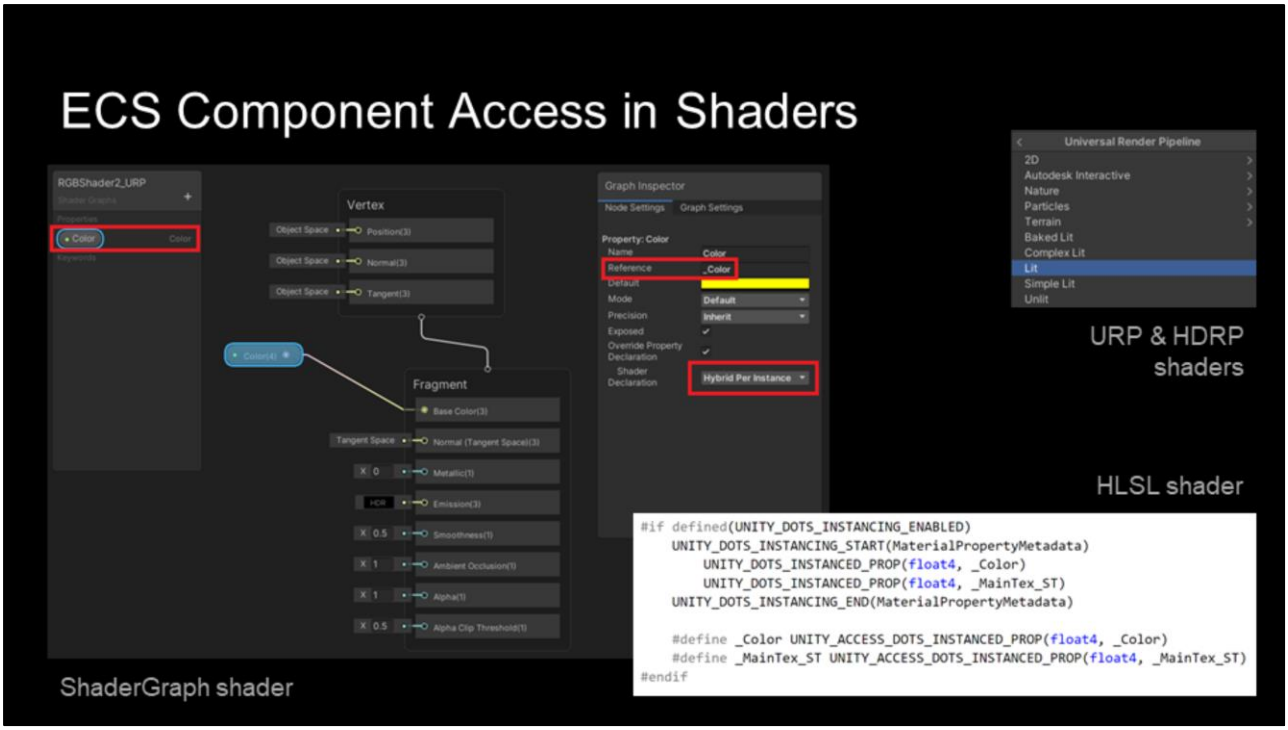


Unity is all about easy abstractions to the user. Our goal is to make setting up shader data as simple as possible, while retaining high performance and data persistence.

The user simply adds a MaterialProperty attribute in front of their ECS component struct, and we mirror the component in GPU memory, and automatically keep the GPU data up to date. There's nothing else the user needs to know.

For advanced users, we also offer an API for manually registering GPU visible components. This is convenient when you want to enable GPU access of components from external libraries, and don't want to change their source code.

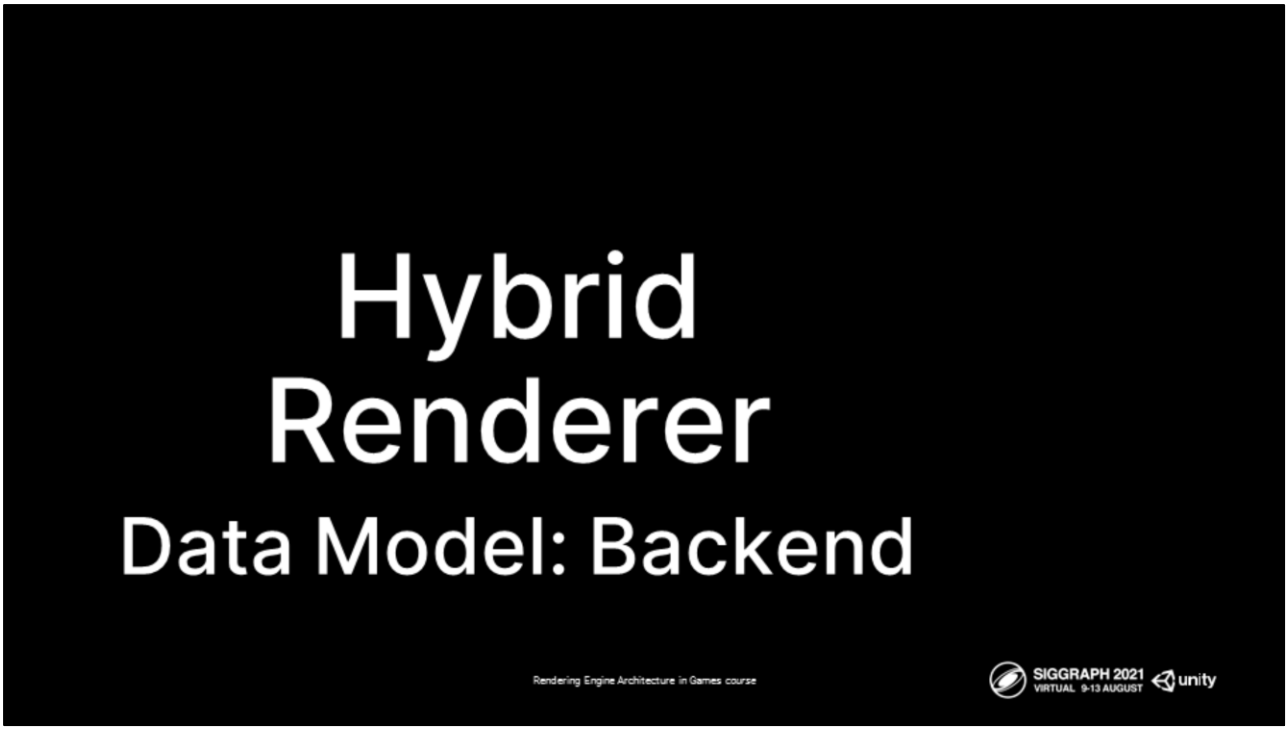
DOTS has a custom TypeManager implementation that provides a subset of C# reflection features. Instead of the slow C# Type and String classes it uses integer identifiers. We only use reflection at startup to avoid any additional runtime costs.



To make the ECS data accessible in shaders, we extend the existing Unity's HLSL GPU instancing macros. This is a small change in the macro ecosystem.

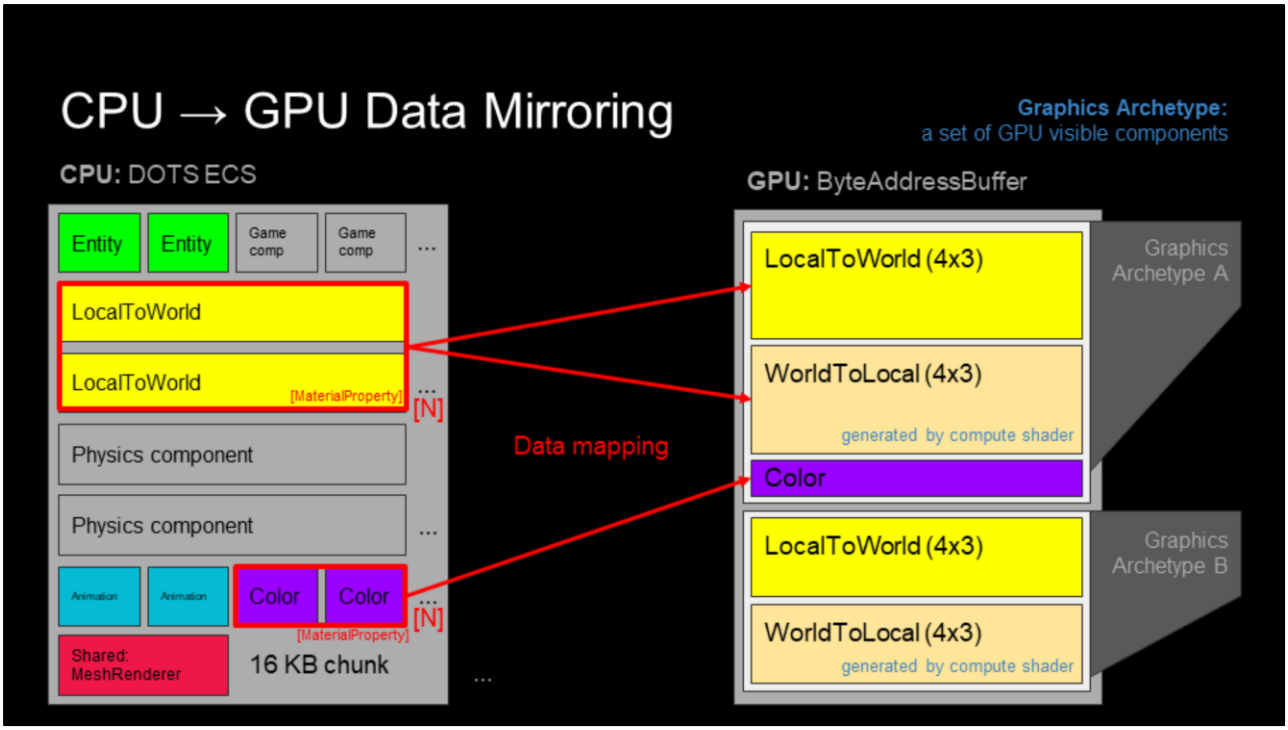
ShaderGraph supports new codegen with these new macros for all input properties. You simply select hybrid instanced in the dropdown menu.

All existing Unity's built-in shader inputs and most of the built-in URP and HDRP shader inputs are now compatible with DOTS ECS instanced data sources.



That’s it for the frontend.

Now let’s talk about how the ECS component mirroring to GPU is actually implemented.



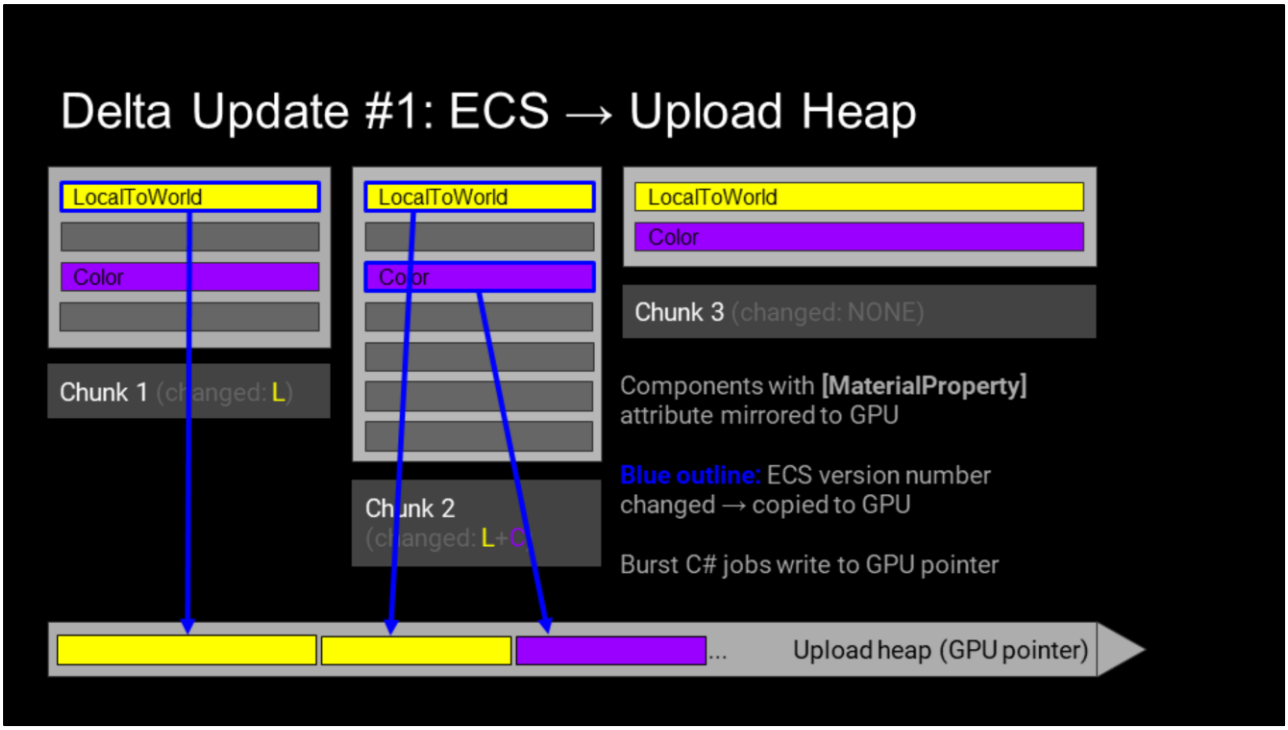
Let's get back to our analysis about temporal coherency and the SRP Batcher. We noticed that GPU persistent data is giving us big performance wins.

Instead of re-building the instance data for each draw call and sending it to GPU memory, we allocate persistent GPU memory for each GPU visible ECS component.

We use a large ByteAddressBuffers to store the ECS data in GPU side.

Each ECS chunk belonging to the same DOTS archetype has identical data layout by definition. We introduce a new concept called "Graphics Archetype". Graphics archetype is a subset of the DOTS archetype. It's the set of GPU visible components.

Each entity belonging to the same graphics archetype has the same GPU data layout. Thus we can store them as N contiguous lockstep arrays, one for each component, sub-allocated from the ByteAddressBuffer. This makes the data address calculation trivial in the shader.

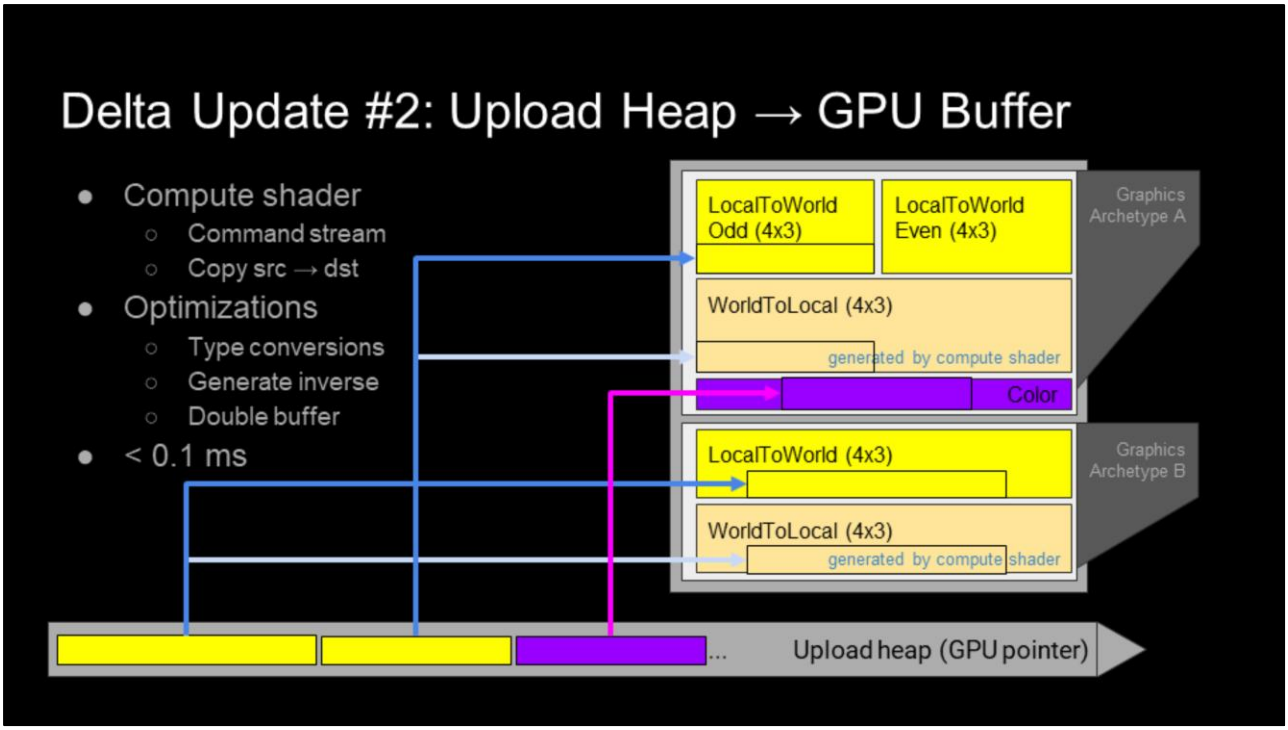


I will now explain how the data update from CPU to GPU works.

If you remember the DOTS version tracking, it's easy to guess how we manage the component delta update from CPU to GPU.

We do a query over all graphics components. In the query, for each chunk, we compare each component array version number with the previous frame's global version counter stored in the uploading system. If the version number is larger, we know that somebody has modified the data since we last uploaded it.

The query job writes a simple command stream containing source and destination data offsets for each segment of data. Then we run a following job with access to mapped GPU upload heap pointer. It copies the actual data arrays from the modified chunks to the GPU visible memory. All of these jobs are Burst compiled and scale to all CPU cores.

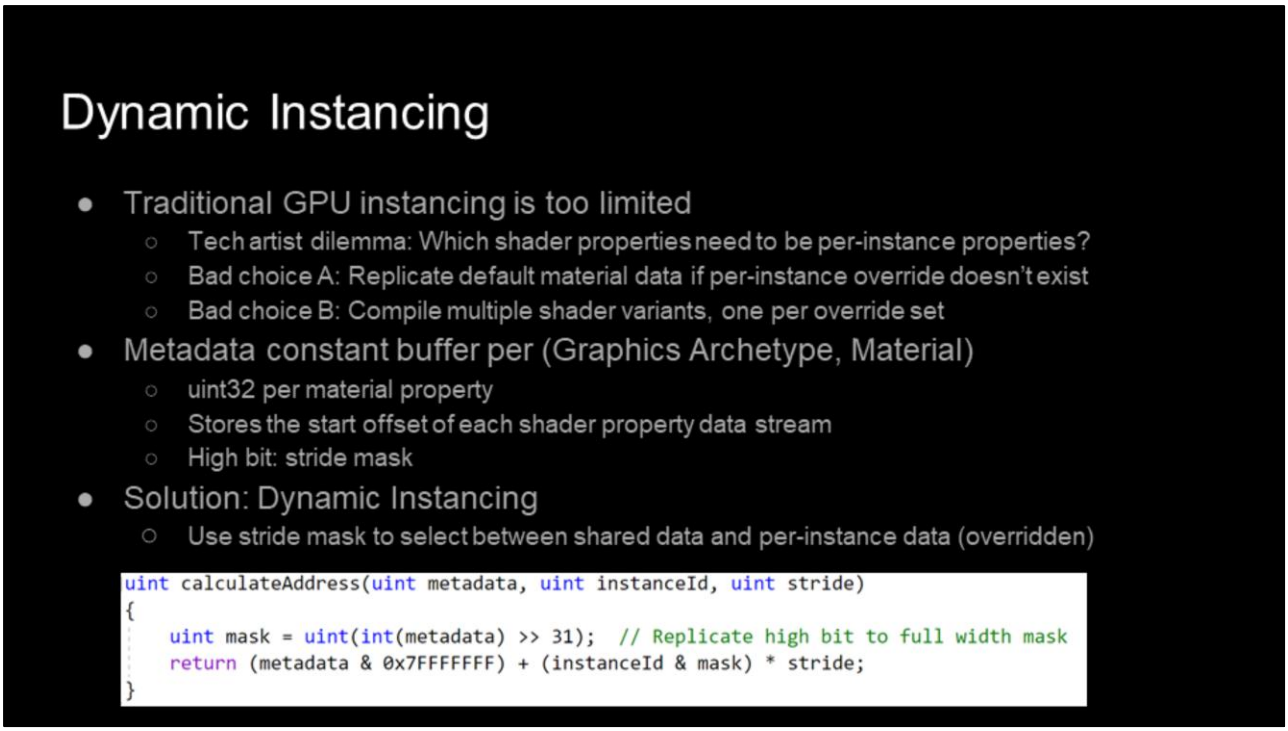


The second step of the data upload process is to go through the linear upload heap data array and scatter the modifications around the large ByteAddressBuffers containing the mirrored ECS state.

We execute this step using a compute shader. The compute shader reads the simple command stream containing the copy source and destination addresses and uses a wave wide loops to efficiently copy the data.

In some cases we prefer to have different GPU data layout. For example 4x3 matrices save 25% of memory and bandwidth compared to 4x4 matrices. Half floats are also preferable for many GPU data.

Our compute shader supports data conversions, and we also handle some special cases like inverse matrix calculation and previous frame matrix double buffering. This is very nice since ALU work is practically free in our copy shader, since it’s bandwidth bound. As the data amplification happens in GPU side, our approach significantly reduces the CPU to GPU bandwidth cost.



Now let's talk about the way the shader is accessing this data.

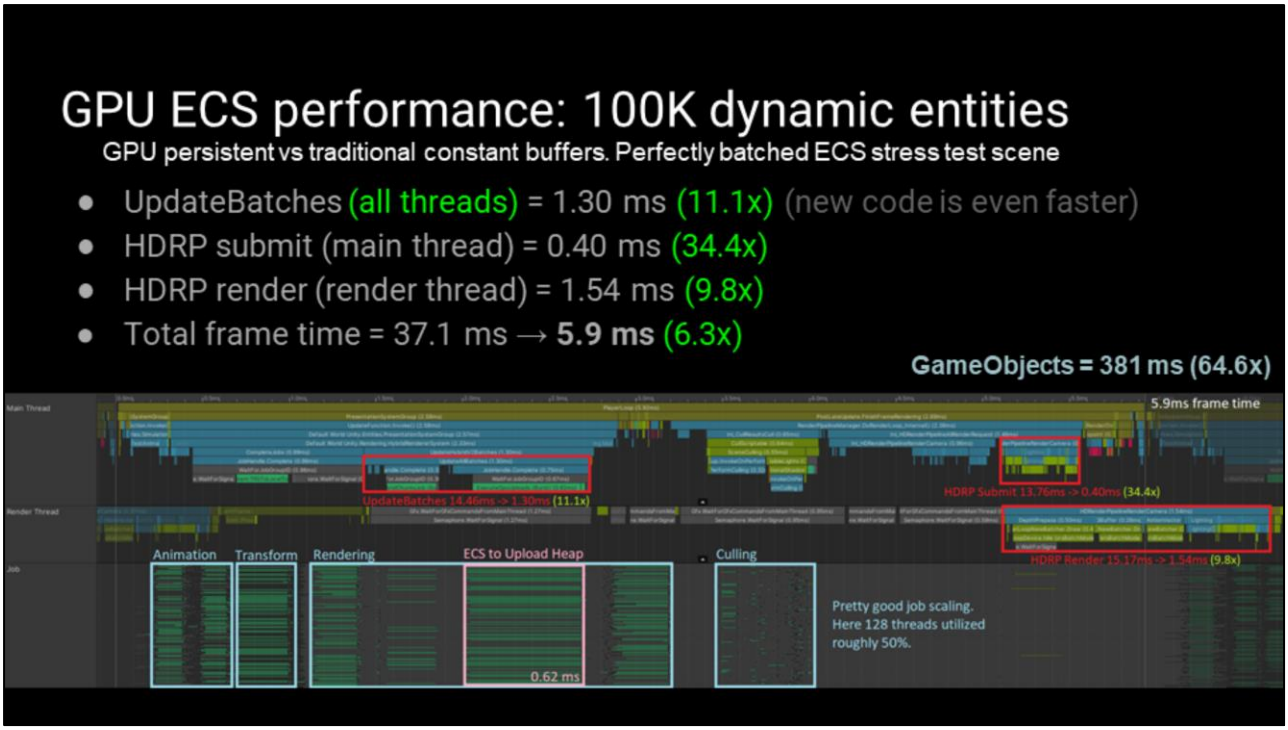
Traditional GPU instancing has known issues. The author of the shader needs to define a hardcoded set of properties that can be overridden per instance, while other properties came from the material. If this set is too large, you end up replicating material default values all over the GPU memory. If the set is too small, the artists feel restricted and can't deliver the content variance they want. Or they use more materials reducing the batching efficiency.

Compiling multiple shader variants for different data layouts is a solution to this problem, but most of us are already dying under the combinatorial explosion, so this path is a no-go.

We are creating a metadata constant buffer for each (graphics archetype, material) combination. This buffer contains the start offsets of each shader property data stream. We steal the high bit of the offset for a stride mask. This way we can force all instances to read from the same shared memory location, instead of reading from contiguous memory addresses. This allows us to store non-overridden material data as single value instead of replicating it per instance. Saving a lot of GPU memory.

Our address calculation code costs two extra ALU instructions. We use arithmetic shift to replicate the mask bit, and binary AND to mask the instance id. Metadata is a scalar register, so it's practically free. No added vector register pressure.

And now, I will show you some performance results.

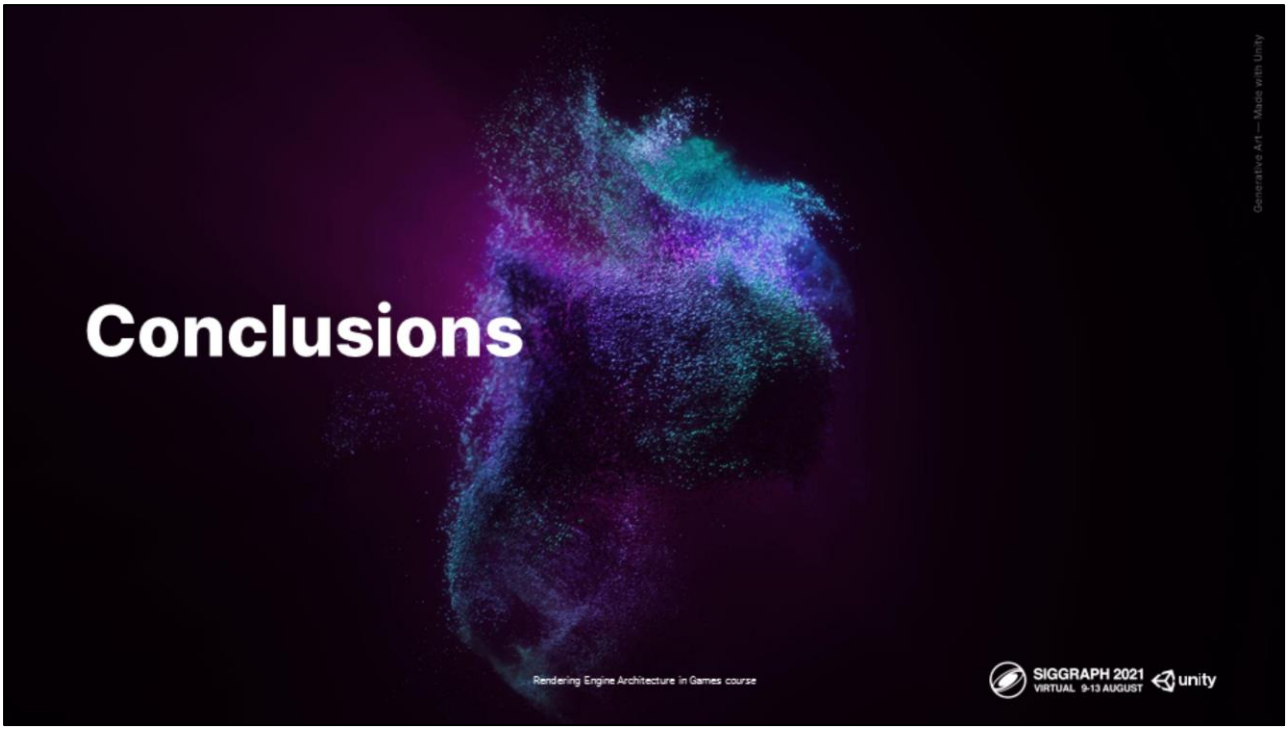


Here, we are comparing the new GPU persistent data model against and older constant buffer based data model. Both versions are using DOTS ECS.

Our new data model sidesteps the slow main thread data path. We write instance data directly to the GPU upload heap from Burst jobs. This avoid the instance data setup costs for the HDRP software command buffer recording and playback. We see roughly 10x to 30x increases in throughput.

This stress test scene runs around 6x faster with the new data model. However this is a perfectly batched scene with no game logic, so it represents the best case scenario. We do however update every entity transform matrix and override color every frame, demonstrating the power of our new partial delta update mechanism.

If we compare DOTS with GameObjects, the situation is even more clear. Throughput difference is over 60x.



So if we look back at our principles at the beginning, how did we do with our architecture? Where do we go next?

Key Principles: Customizability

- Developed 2 unique pipelines on top of the core SRP architecture
- Enabled other developers build their own render pipelines and ship
- URP is proving to have a great deal of customizability hooks
- HDRP is adding more performant customization options over time

LEGO® Builder's Journey
URP: iOS
HDRP: PC (including ray tracing | DLSS 2.0)

We've successfully built two different rendering stacks based on the SRP architecture which are flexible and powerful enough to continue unlock creativity in people that choose to use or modify Unity. We've had numerous games ship successfully on our render pipelines, and more are shipping in the coming years. We've also had success stories on enabling developers ship games building their own pipeline.

[Lego Builder Journey: URP on iOS, HDRP on PC]

Key Principles: Scalability & Platform Reach

- SRP achieved intelligent scaling of low-to-high-end
  - URP allows us to have single project for all our platforms
  - HDRP scales gracefully from high-end ray tracing | PC to consoles
  - Projects ship successfully simultaneously on both RPs

- Remaining work: Cross-RP workflows
  - Some asset types can already target multiple SRPs in a single asset (shaders, textures)
  - Next: single project for both SRPs, cross-RPs materials, lights, settings, and more

We've accomplished a lot of our goals wrt scalability and platform reach. The SRP layer and tooling enables users and pipeline developers alike to have access to the full platform range that we support at both the low level (for tuning) and higher level (scaffolded APIs), providing a pathway for **performant** scalable rendering.

We continue to invest into this to keep improving our scalability. Next, we are building easy workflows for sharing content across both RPs - some assets can already target multiple RPs (such as shader graph), next, we are working on adding support for single cross-SRP project, settings | materials | lights assets and more.

A part of that journey is also finding a good way to programmatically author shaders through a powerful yet consistent interface (akin to the surface shaders of BiRP) to allow programmers to quickly create shaders that target all three render pipelines.

Unity runs across a lot of platforms, for us it's really important for customers to have access to these platforms in a performant and easy to use way.
With URP, you can have better performant single project targeting the full range of our platforms, from low- to high-end. On HDRP, we have great scalability from super high-end ray tracing experiences on PC down to Xbox One consoles.

(Example: Road 96)

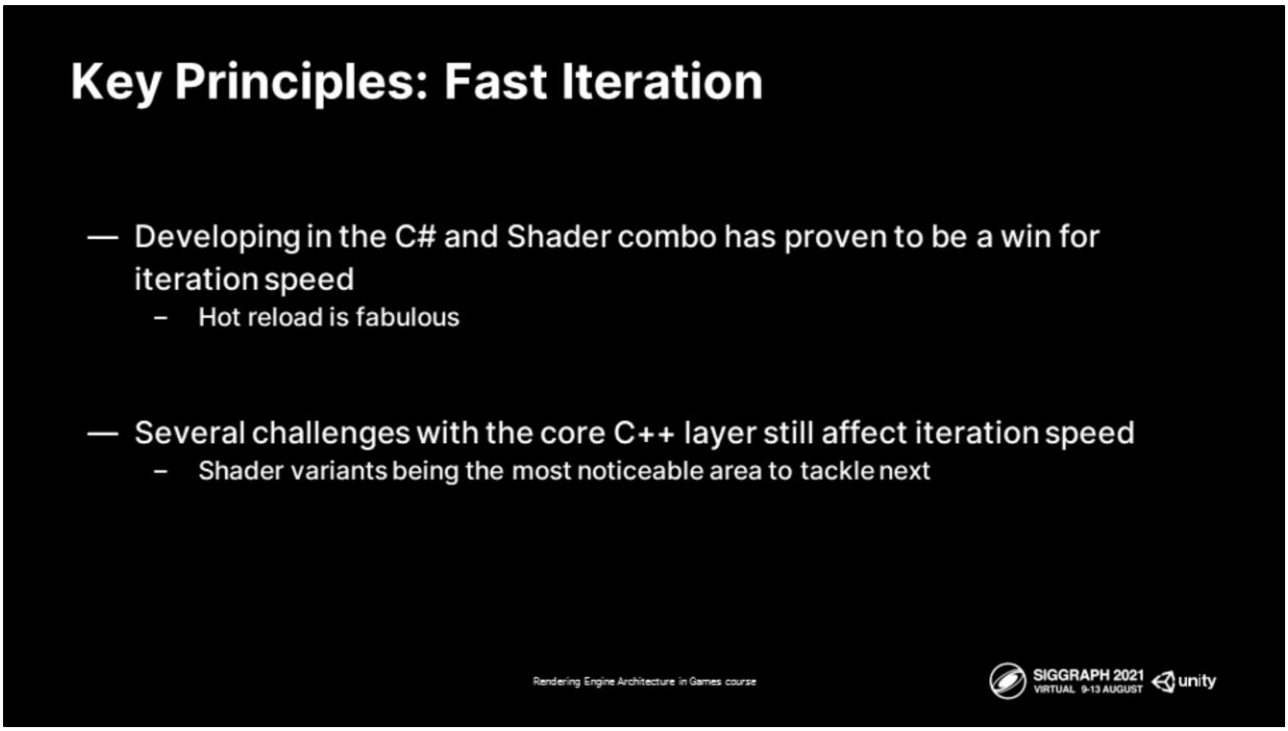
Key Principles: Performance

— Significant performance improvements on all platforms with URP | HDRP; even better with Hybrid
— New performant runtime with Hybrid without a full rebuild

— *Next*: further investment into lower-layer improvements, ray tracing performance improvements, novel graphics features, performance tooling.

URP and HDRP show significantly faster performance versus BiRP on CPU and GPU, and hybrid DOTS SRP is delivering further drastic performance improvements. With Hybrid, we are able to build a new runtime for solving the performance issues without having to rebuild everything from scratch.

Next we invest further into lower-layer improvements (culling, DirectX, Vulkan, ray tracing), advanced features such as VRS & mesh shaders, and continue optimizing each render pipeline.



Developing in the C# and Shader combo has proven to be a win for iteration speed. Additionally, developing in a modular fashion in the SRP land has also been a bonus. Yet to get to truly great iteration speed, we need to tackle next the problem of shader variants, which, with every option added to each pipeline, continue to grow. That’s an area we’re going to dig deeper into next.

Evolution takes time, but it’s worth doing

Rebuilding train tracks under a moving train is hard, yet no perfect solution exists for a large user-base code base.

Large ecosystem requires extra care:

- Engage early to ensure smooth conversion
- Provide rich set of upgrade helpers
- Deprecation journey is long, plan accordingly

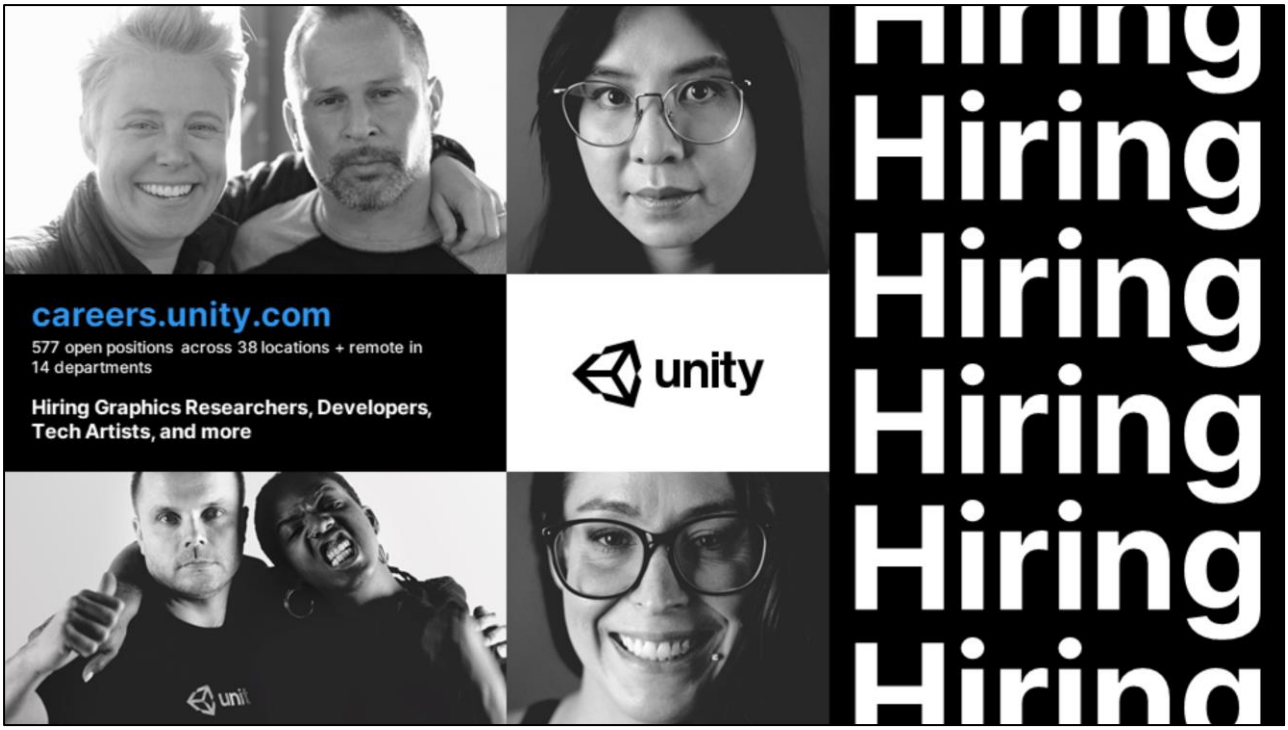
Overall, we feel we’re achieving many of our goals and learning a ton of great lessons along the way.

Rebuilding the renderer architecture in a living game engine with many customers can be a complex effort. We learned valuable lessons about evolving our ecosystem through the architectural changes, from creating smooth upgrades for our customers to the new architecture, to making it easy for the ecosystem to target the full gamut of our pipelines. One takeaway is that while it’s clear that the interfaces’ stability is beneficial, stability of the data interfaces is more important. By having stable data interfaces we can run the same content on different rendering backends.

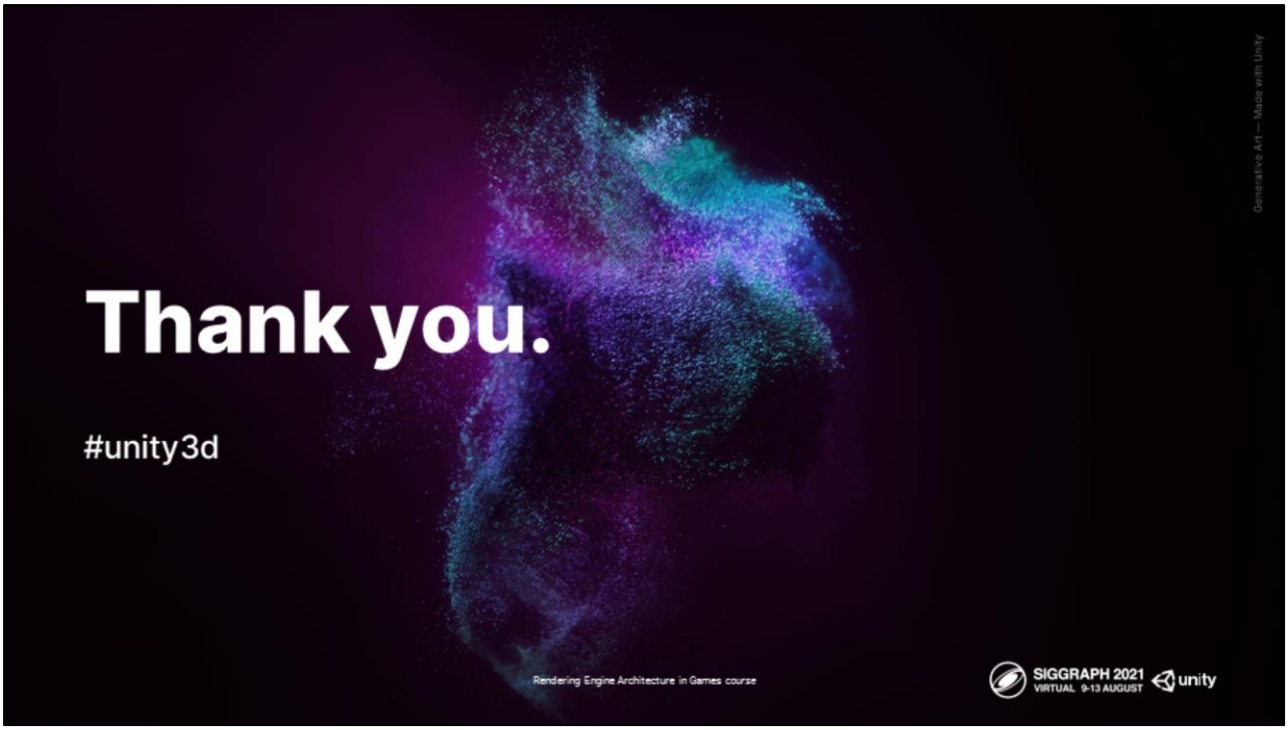
The other important aspect we’ve lived through is that, for third-party engine like ours, sunsetting architecture is a very long journey, depending on how widely adopted it may be. So at the moment all of the pipelines are available in the engine as we have a lot of productions successfully shipping on BiRP. it’s important that we plan for this investment appropriately for the lifespan of the full architecture bring up.

Overall, we feel we’re achieving our goals and learning a ton of great lessons along the way. And we’re hardly done!

We wanted to express our thanks to the Unity Graphics & Platforms Team for helping build this architecture, as well as to **Ali**, **Mathieu**, **Felipe**, **Sebastien**, **Aras**, and **Pierre** for various help on the presentation. The feedback that **Steve**, **Angelo**, **Michael**, and **Peter-Pike** provided helped this presentation tremendously - thank you all so much!



Last but not least, Unity continues hiring graphics researchers, developers, tech artists and more world wide. If you found our architecture interesting, come join us, help us evolve and improve it further! We are also hiring principal rendering engineers for the graphics innovation group.



Thank you for listening to our talk!

Q&A