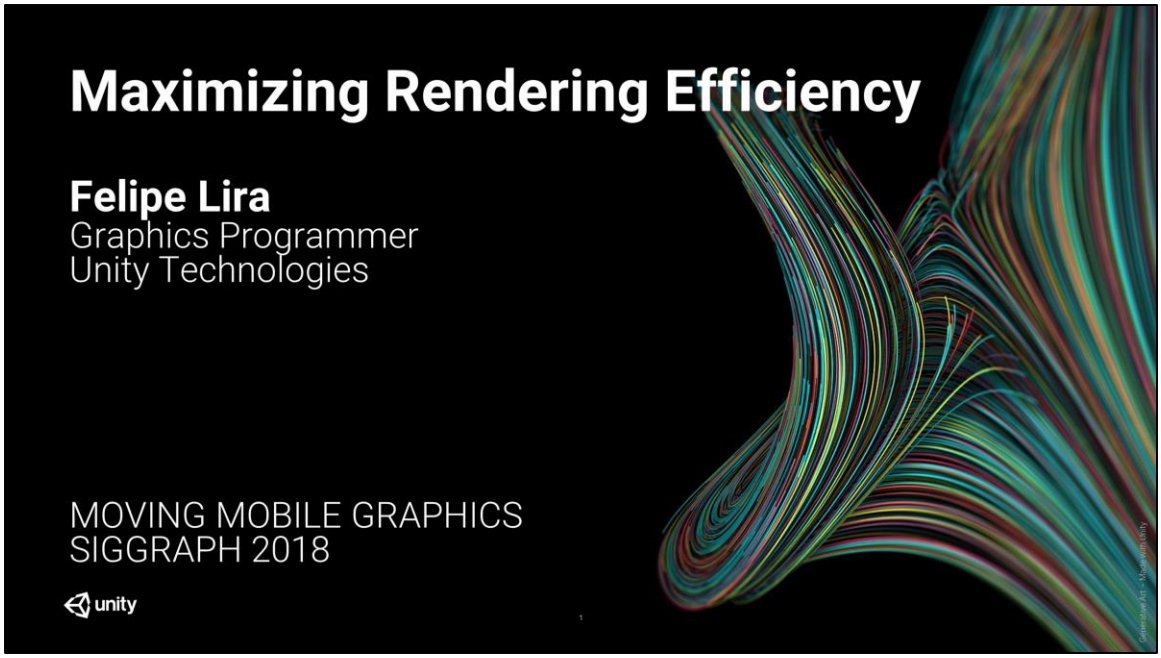
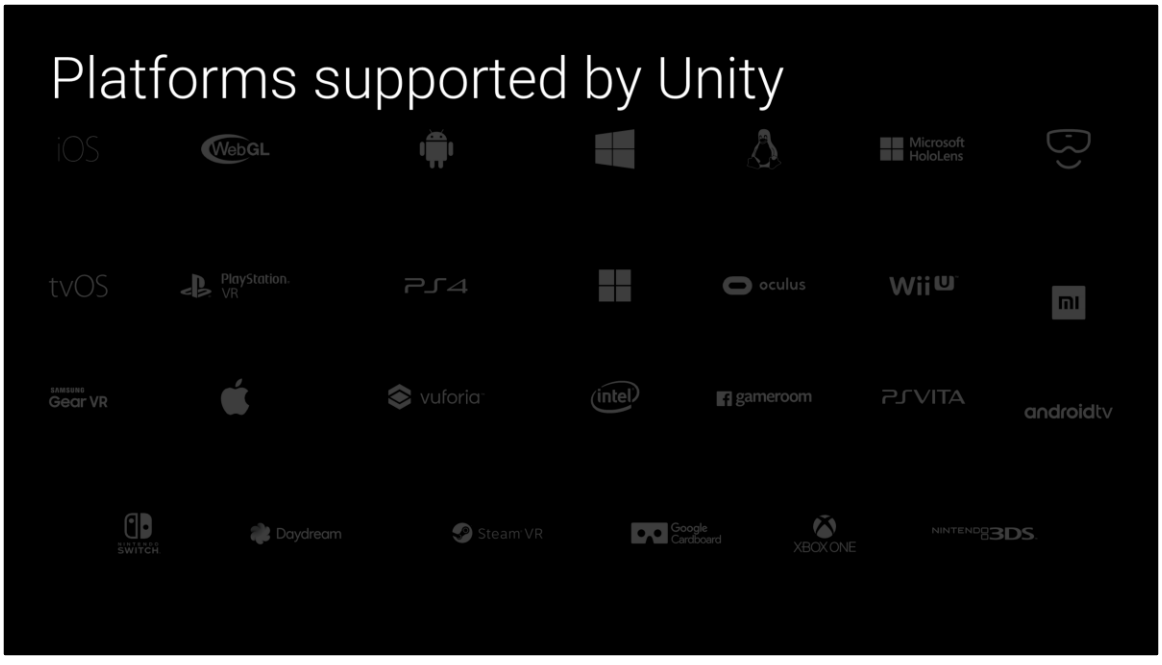
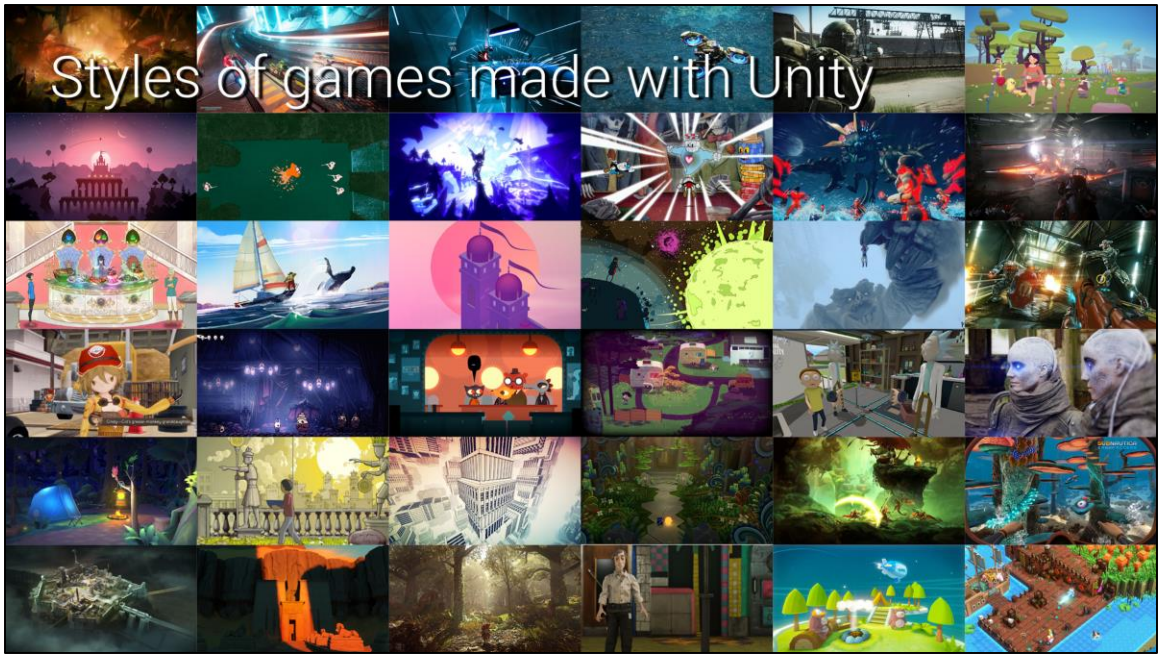


Hi, I'm Felipe Lira. I work as graphics programmer at Unity and today I'm going to talk about the latest work in Unity to maximize performance for mobile platforms.

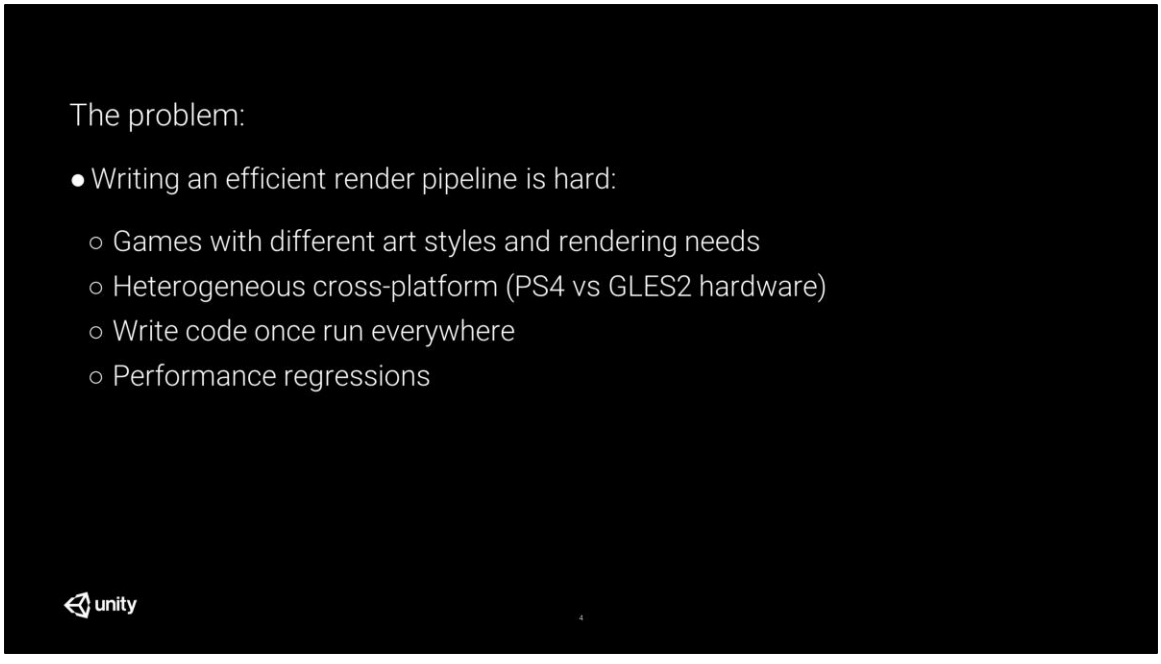
Optimizing a game for a specific platform might be challenging. That involves knowledge about the hardware architecture and getting acquainted with the manufactures profiling tools.



Now consider the amount of platforms Unity supports.



… and the amount of different rendering styles game developers want to achieve.



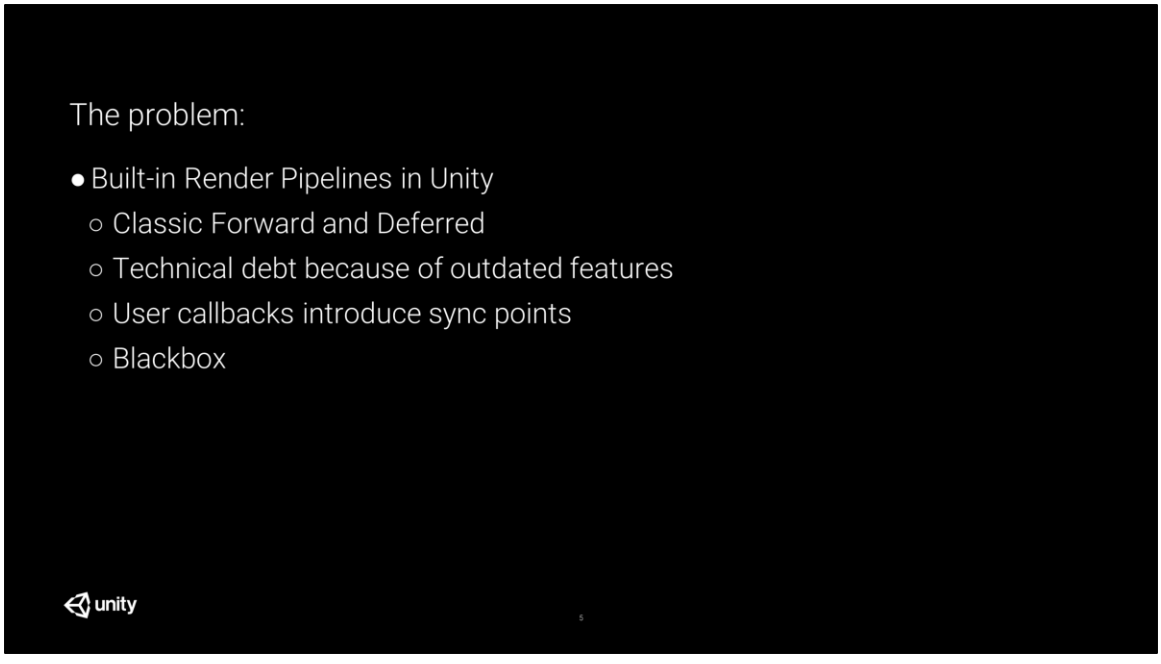
It’s hard to provide a single render pipeline that serves well all of these different games and platforms. Games vary between stylized and physically based. Some require many realtime lights while others might only need a single directional light or no light at all.

On top of that, an approach that works really fast on a PS4 most likely will not scale well for mobile. Even if you just considering mobile platforms the device fragmentation makes it hard to have a single and optimized rendering approach for all devices.

Ideally when performing an optimization we want to profile all platforms to make sure there is not a performance regression. Otherwise we might end up in a “running in circles” situation that we increase performance for some platforms while regressing on others.

Unity also supports a broad range of platforms. There are a lot of games being developed for GLES 2.0 capable phones and this is an important segment of our business.
Whenever we develop a new feature we have to make sure it works on all platforms.

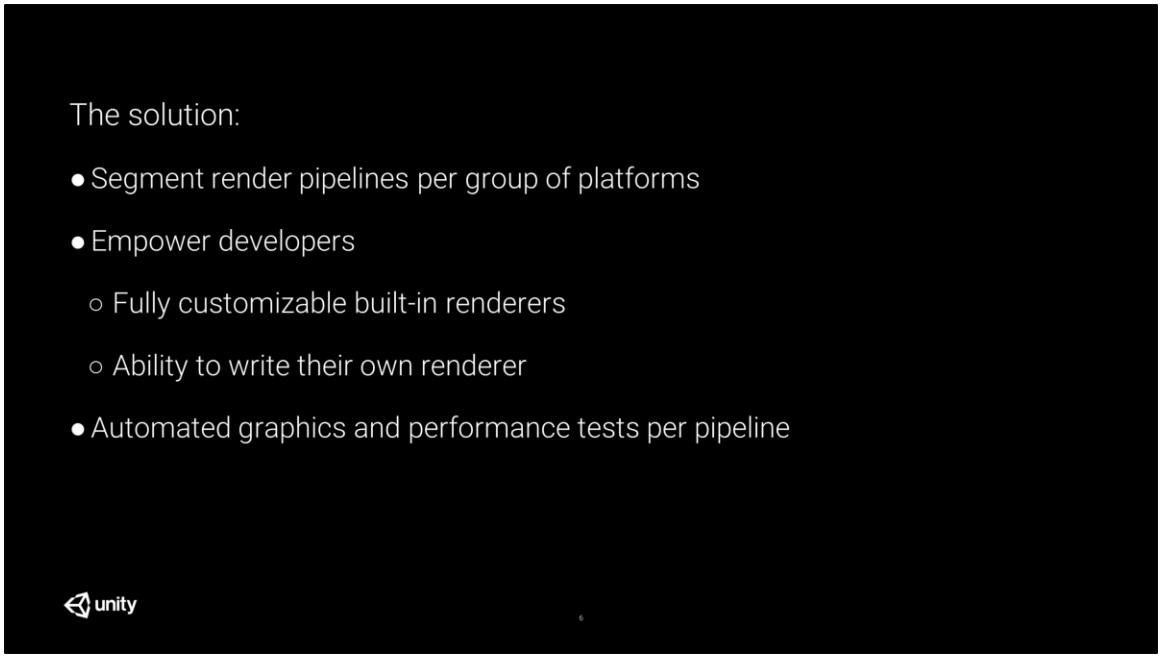
That means we have to provide fallbacks to older platforms if these features can’t be supported by them. This increases greatly the cost of development and maintenance.



Traditionally Unity supports a classic forward and traditional deferred pipeline as built-in pipelines. These pipelines are at least 10 years old now and we want to deprecated some rendering features we designed back then but can't break backwards compatibility. That leaves us with undesirable technical debt we have to maintain.

Over the years we added customization hook points into the engine in the form of callbacks. Some of these callbacks introduce sync points in our pipelines. For instance, users have a callback point after culling has finished. That means we have to wait on culling results to be able to provide them the list of visible renderers.

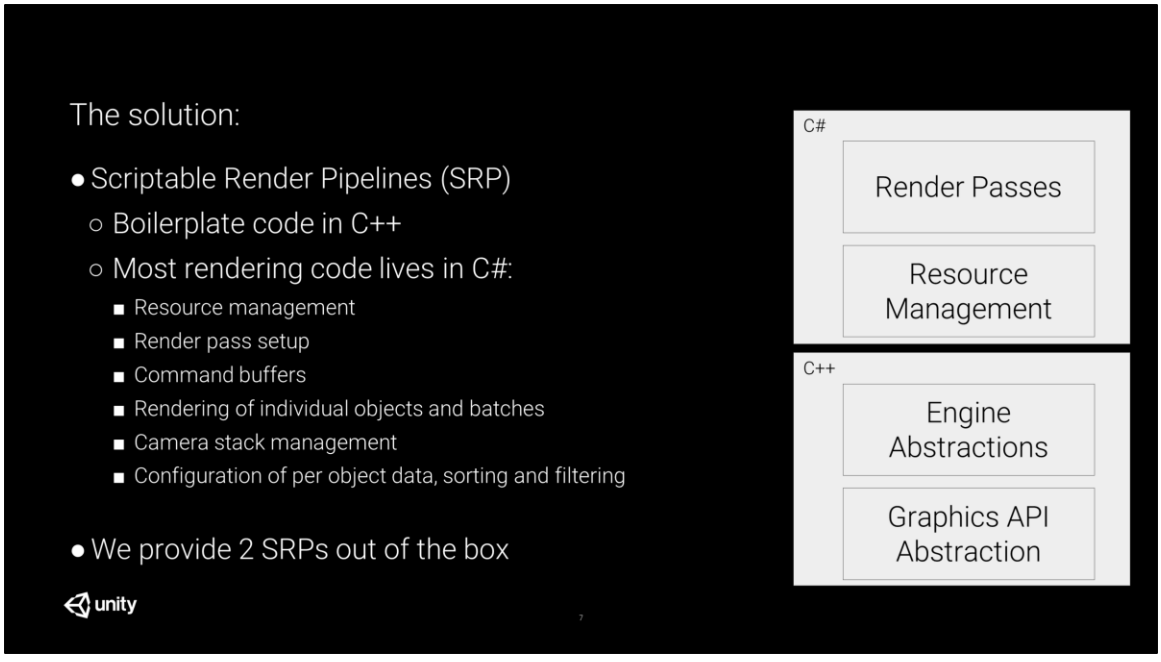
Another major downside is that the pipelines are a black box. User can't see what is happening and customization is very limited.



It seems clear that the solution is to develop multiple render pipelines segmented by platforms. If we want to truly marvel at mobile performance we need to have a specialized renderer for it.

We also need to empower our developers so they can easily tailor render pipelines to their specific needs. They need to be able to strip everything from the render pipeline that they don’t need while still being able to add custom render passes for specific effects.

By segmenting pipelines into platform groups we also decrease the burden of development and pitfalls of performance regressions in platforms that are not the render pipeline focus.



We provide to users what we call “Scriptable Render Pipelines” With SRP we expose to user land C# rendering API. This allows any developer to create a render pipeline in C#.

In C++ we have our graphics API and engine abstraction layers. This consist of for instance of graphics device, render surfaces, shader management, culling, light management just to name a few.
In C# we expose a list of cameras to be rendered and a render context that developers can enqueue graphics commands to it. Developers can also control when the context is submitted to improve job scheduling. When overriding Unity with a Scriptable Render Pipeline all rendering will go through it. That means game, editor, preview and reflection probe cameras.

We provide 2 SRPs out of the box for users. They can use them as reference to implement their own from scratch or simply customize one of them for specific needs.

The solution:

- HD Render Pipeline
  - Supports PC, Mac (Metal), PS4 and Xbox One
    - Designed to render fast on console
  - State-of-the-art rendering features
    - Mixed forward/deferred Tile/Cluster renderer
    - Physically based rendering
    - SSS, Anisotropy, Iridescense
    - Physical light unit
    - Area Lights
    - Volumetrics
    - Unified lighting

unity

State of the art rendering for PC and Consoles.

The solution:

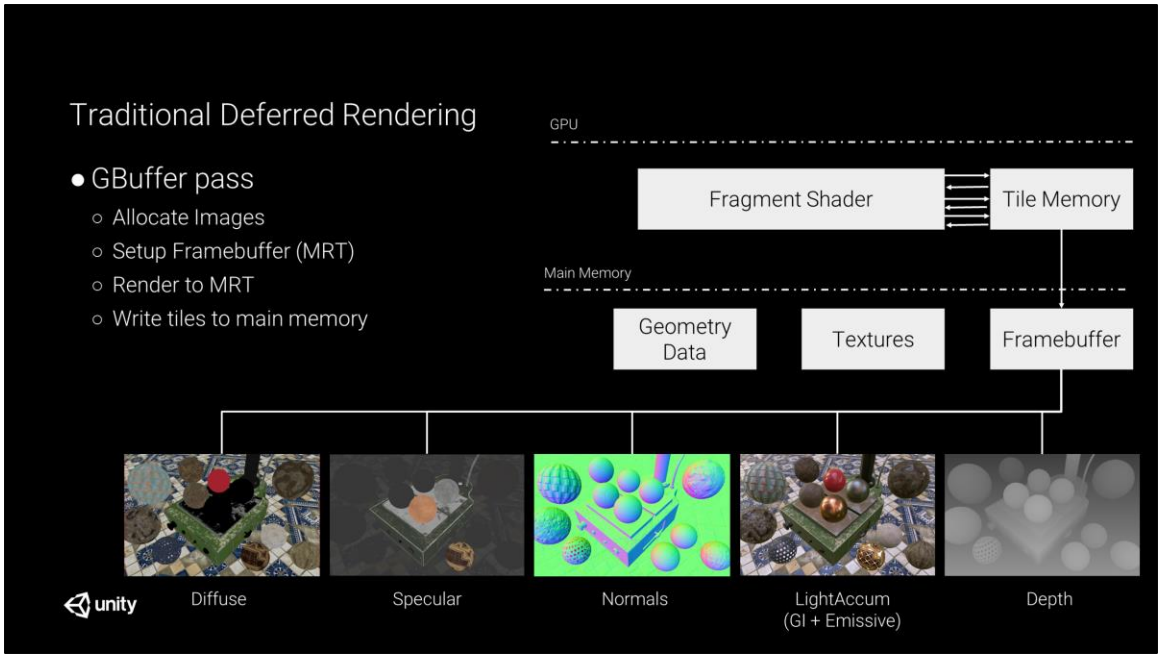
- Lightweight Render Pipeline
  - Supports most Unity platforms
    - Designed to render fast in mobile
  - Easily upgradable from Built-in pipelines
  - Similar but improved workflow compared to Built-in
  - Single pass forward renderer
  - On-Tile deferred renderer (WIP)

unity

Designed to perform efficient rendering in tiled based GPUs. LWRP provides a similar but improved workflow compared to Built-in pipelines and a material upgrader to easily convert projects to LWRP.

Why is traditional deferred rendering bad in mobile?

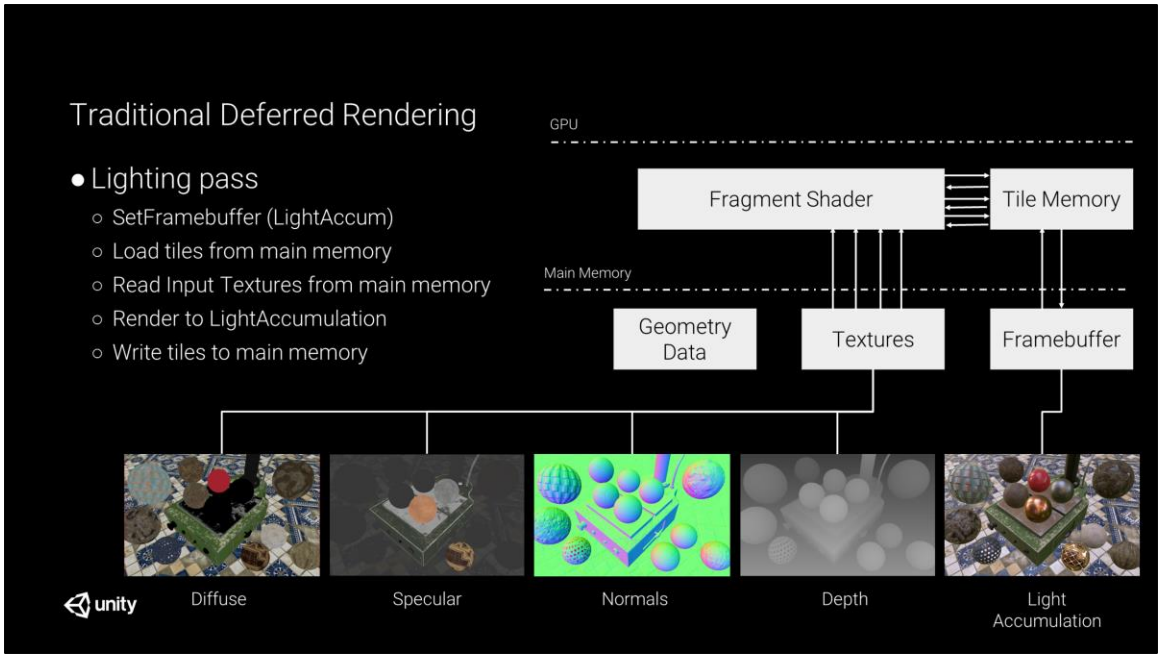
unity



Now let’s take a look how a traditional deferred renderer works and why this is bad for mobile.

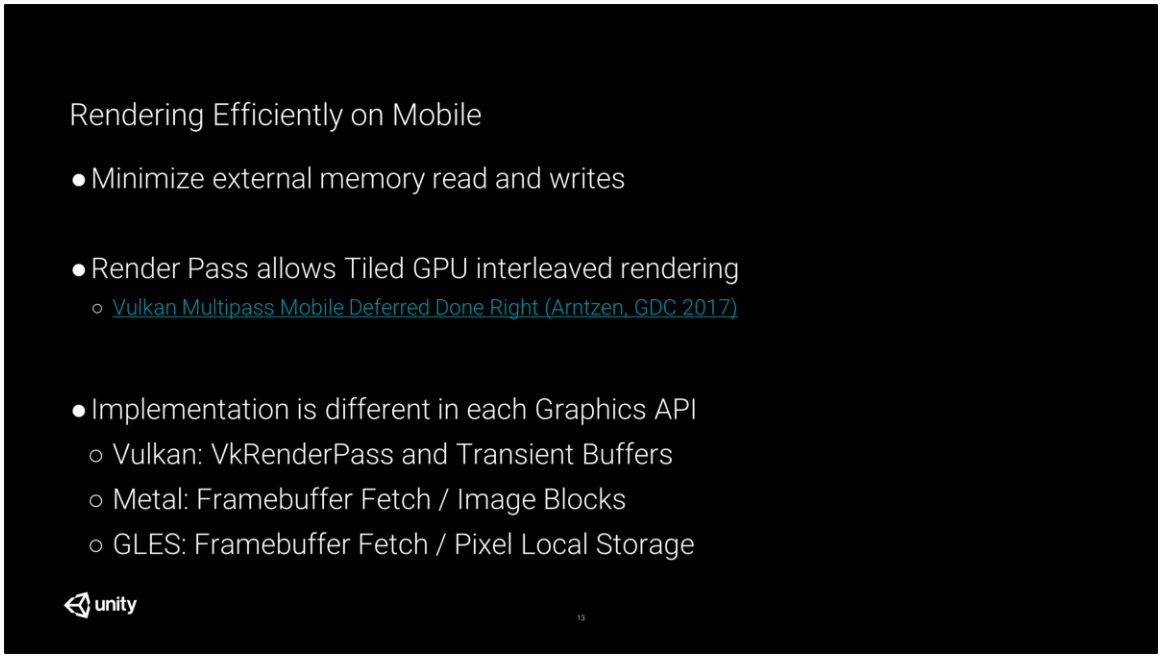
We start by rendering GBuffer. In our example we have a GBuffer layout with 4 textures. (Diffuse, Specular, Normals and LightAccumulation that at this point we render Global Illumination + Emissive).

Images for each target are allocated and uploaded to GPU main memory. We set our framebuffer to render depth + multiple render targets.
For each tile we set loading operation to clear. We don’t pay any cost of initializing them at this point. Upon finishing each tile writes back to main memory.



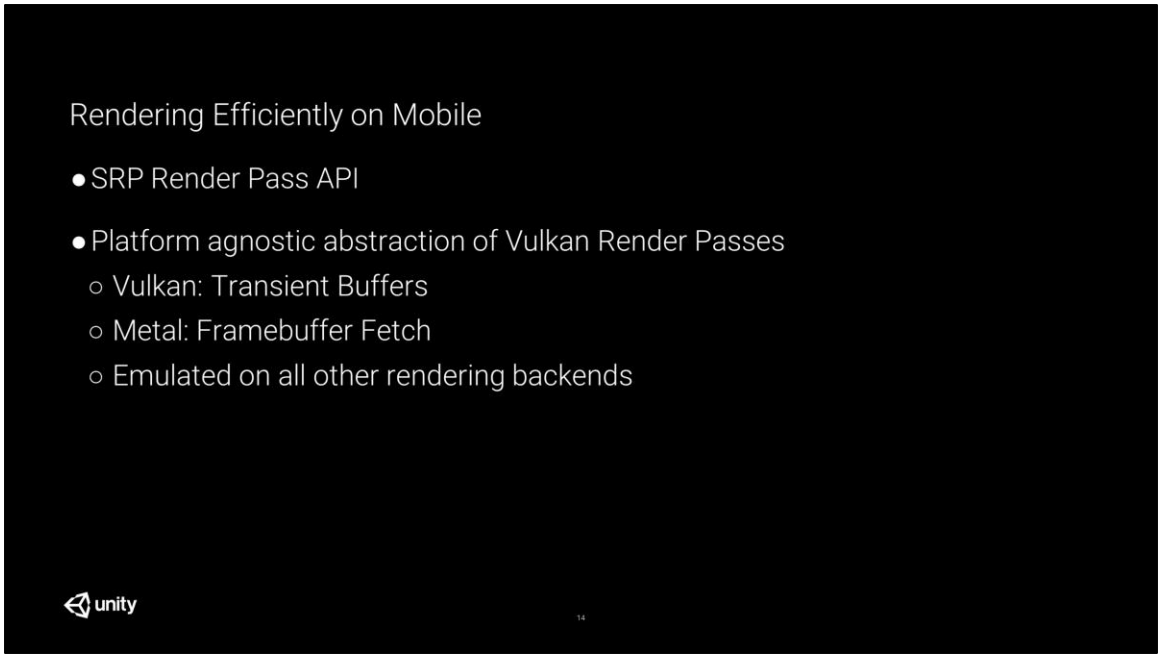
Now we render lighting pass using GBuffer as input textures. We set our render target to Light Accumulation. We set set tile loading option to load as we need the contents of the light accumulation texture.
When rendering we sample GBuffer textures and depth. This will incur in reads from main memory. When finishing we write tile contents back to main memory.

As you can see, there's a lot of reads and writes from and to external memory. This consumes a lot of power, which in turns overheat the device and triggers passive cooling by throttling down CPU and GPU clock, affecting performance.



This is clearly bad for mobile. So how to render efficiently on mobile. For a deferred renderer we need to minimize the external memory read and writes.

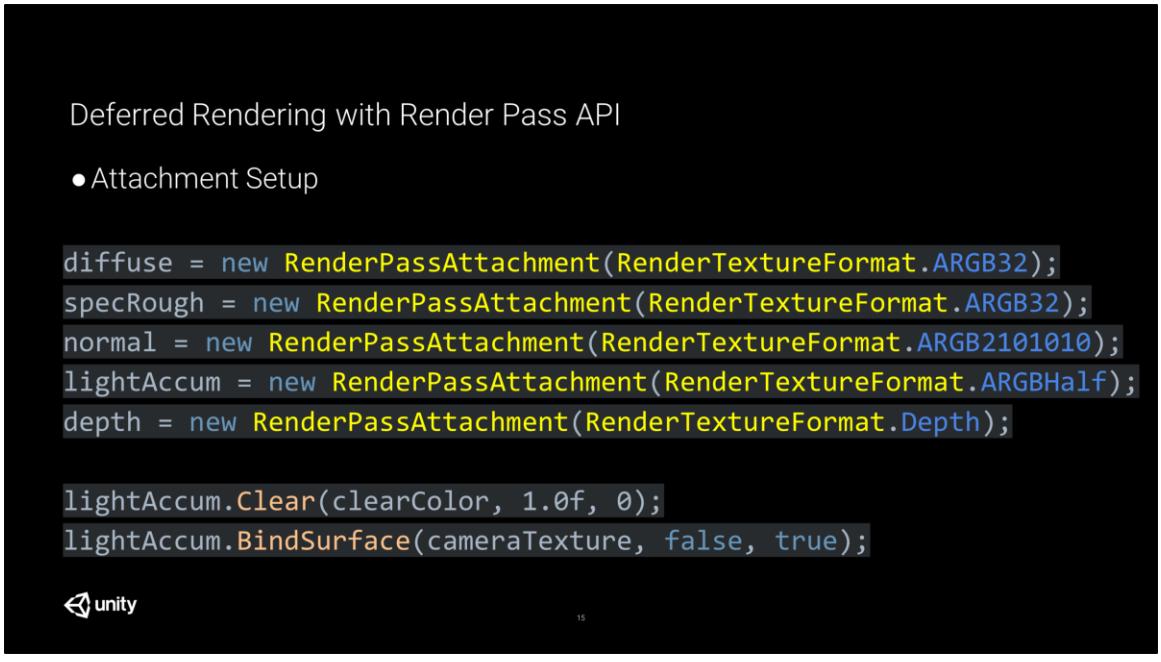
Vulkan introduces the concept of RenderPasses. It allows to stay on tile memory by doing interleaved supass rendering.
The same concept can be applied in other graphics APIs. In metal for instance it can be achieveved with framebuffer featch of image blocks. In OpenGL ES with framebuffer fetch or pixel local storage.



Along with SRP we expose a RenderPass API.

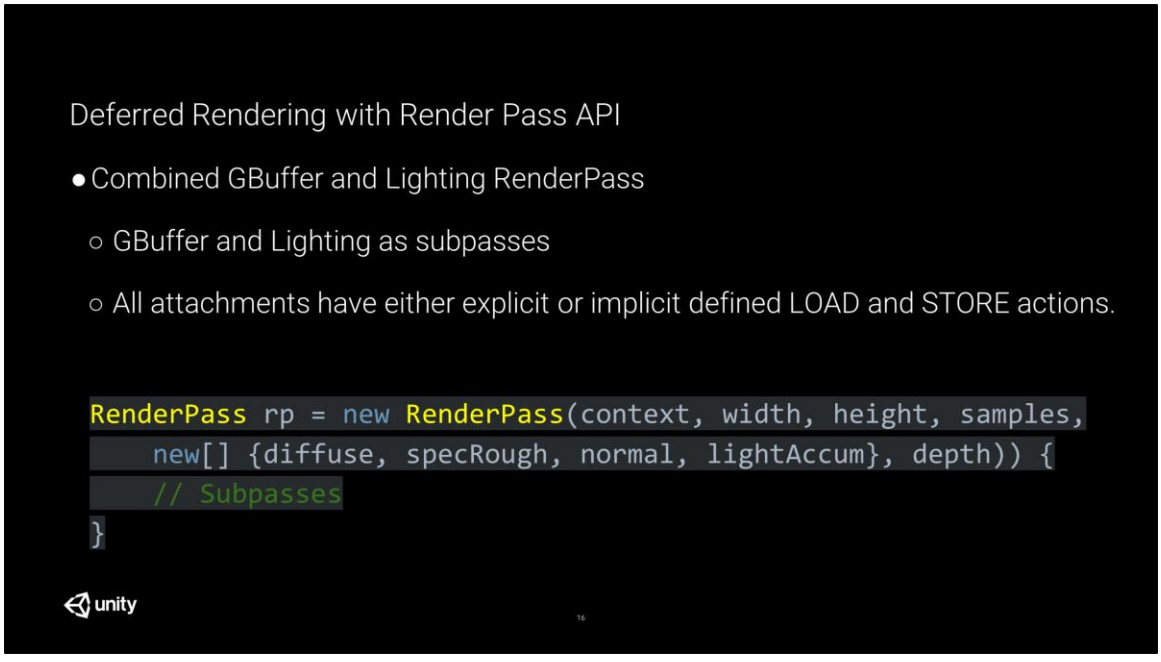
Our api provides an abstraction of the concept of Vulkan Render Passes. It’s currently implemented in Vulkan and Metal and emulated on all other platforms.

The advantage here is that the same code will allow to run efficiently on all mobile platforms but on non mobile as well.



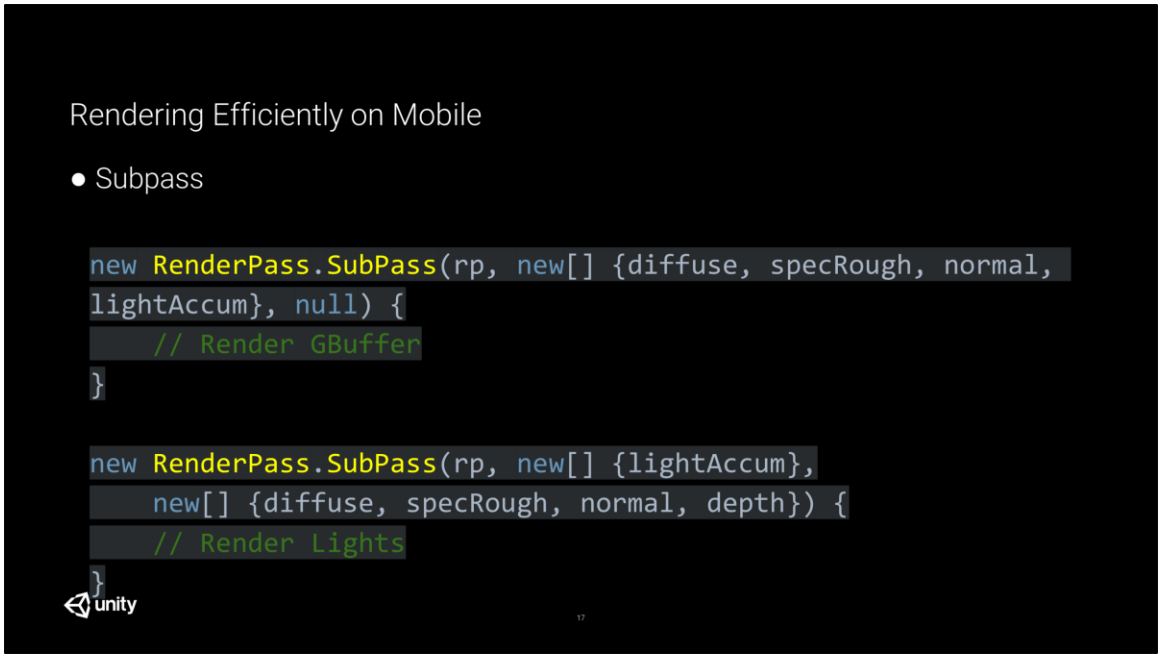
So, let’s take a brief look at how we would implement an efficient deferred renderer using the RenderPass API.

We first create the attachments we will need for our GBuffer. We can bind attachments to physical surfaces and declare how the LOAD and STORE operations for that attachments. That is done with a simplified set of boolean values in the BindSurface function. We just tell the API if we care about loading and writing from and to tile memory. We figure LOAD operation between clear and don’t care depending if the attachment configured a clear state.



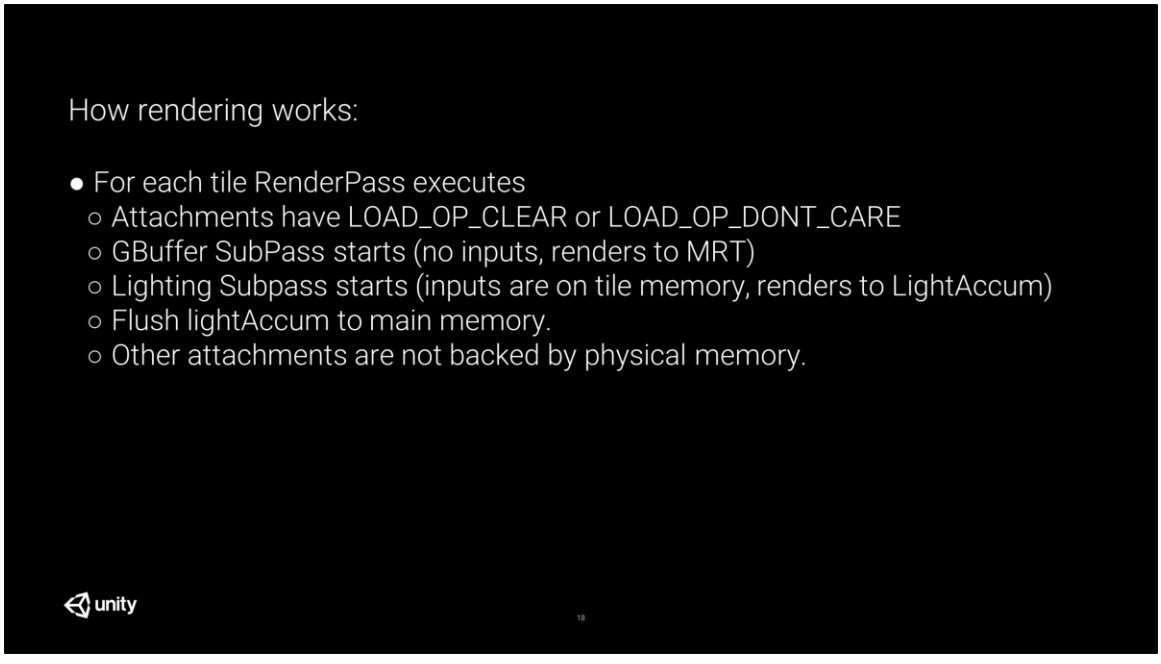
Then we declare a RenderPass. In our case we will declare a combined GBuffer and Lighting render pass. In the render pass we declare all the render target size, samples and all attachments it might write or read from.

Attachments will either have an explicit load and store actions or we will figure out them implicitly based on the dependencies of the subpasses.



Then we declare two subpasses. GBuffer writes to MRT and takes no input. The render target setup is handled by the subpass here and any drawing commands inside the subpass will render to the framebuffer setup we currently have.

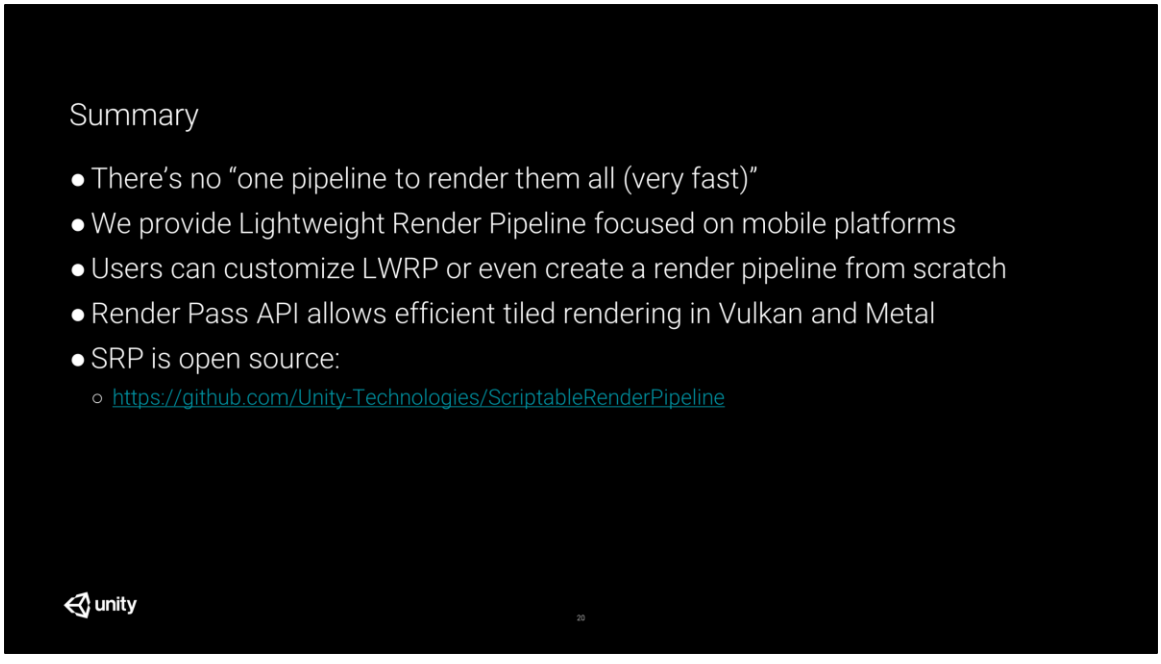
We also declare a Lighting subpass that writes to light accumulation and takes GBuffer as input. Input attachments have some restrictions.
They can only be results of a previous subpass and we can only sample pixels that correspond to the same fragment we are shading.



So, how does rendering work now?
For each tile RenderPass executes.
All attachments have load and store actions explicitly or implicitly defined.
GBuffer subpass starts.
Tile memory is not initialized as in our case all attachments are set to either clear or don’t care loading operation.
GBuffer finished rendering. We don’t flush tiles to main memory. Instead we call next subpass.
At this point we have the results of GBuffer pass in tile memory.
Lighting subpass starts rendering. All inputs that we need to read from are already in tile memory. That means when sampling GBuffer we don’t read from external memory.
Lighting suppass finishes rendering and we call next subpass. All subpasses have completed.
We store contents of lightAccumulation back to main memory.

All other attachments don’t even need to exist outside of tile memory. So, we don’t even create physical memory for them.

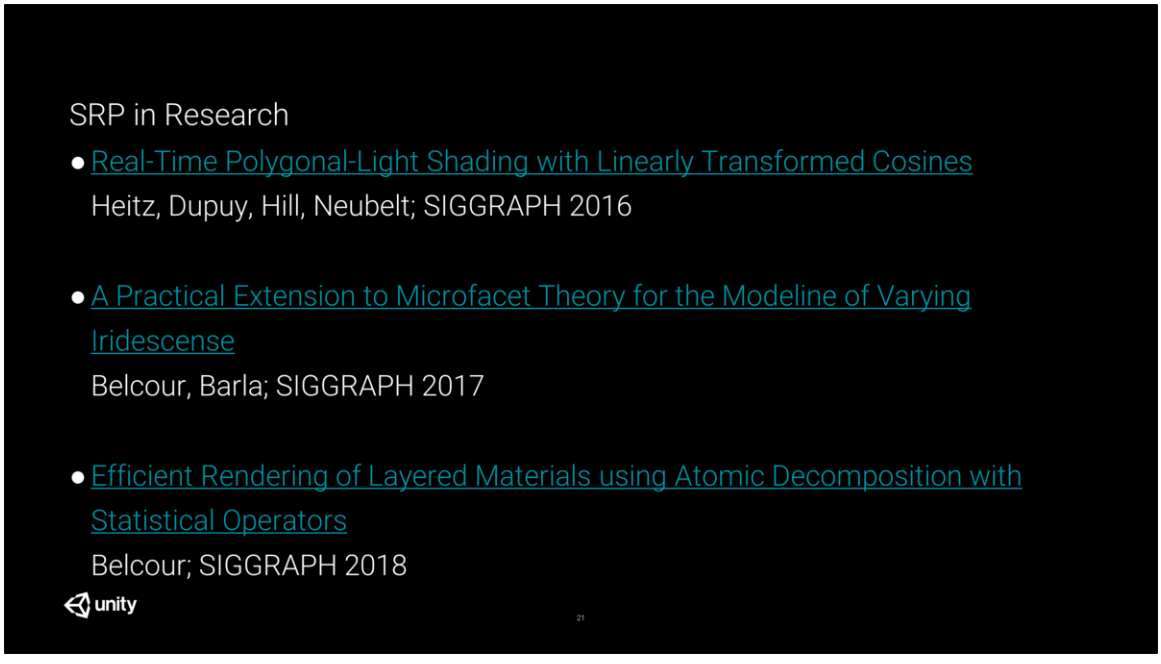
In the shader side we also provide macros for declaring and reading framebuffer inputs. Our compiler backend handles shader code generation for each graphics API.



In summary. Cross platforms / Cross gaming optimization is hard. There's not a silver bullet render pipeline to render everything fast. We provide a Scriptable render pipeline that is optimized for mobile and API that allow developers to either customize it or write a render pipeline from scratch.

Along with the SRP api we provide the RenderPass api to allow anyone to write efficient rendering code for multiple platforms.

SRP is also open source. So, take a look, experiment with it, and develop efficient mobile rendering pipelines for multiple mobile phones.



SRP is an experimental and evolving API but it is already being used in research. :D

Das Questions?

Thank you!