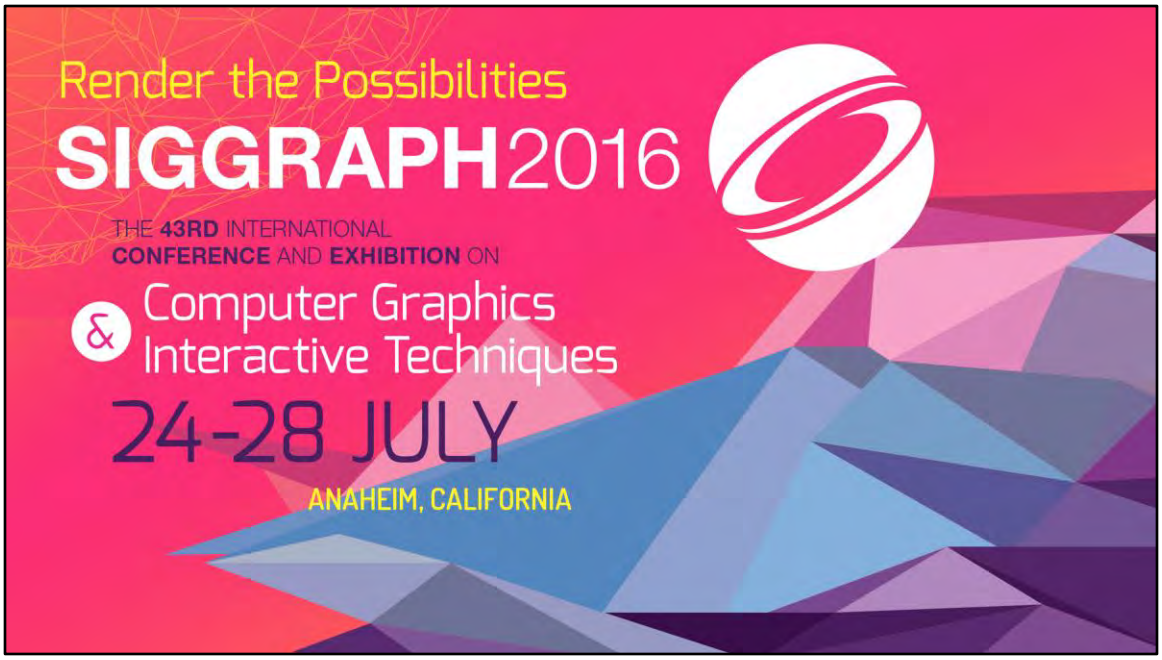


If this doesn't look familiar, you're in the wrong conference

This section of the course is about the ways that mobile graphics hardware works, and how to work with this hardware to get the best performance.

The focus here is on the Vulkan API because the explicit nature exposes the hardware, but the principles apply to other programming models.

There are ways of getting better mobile efficiency by reducing what you're drawing: running at lower frame rate or resolution, only redrawing what and when you need to, etc.; we've addressed those in previous years and you can find some content on the course web site, but this time the focus is on high-performance 3D rendering.

Those other techniques still apply, but this talk is about how to keep the graphics moving and assuming you're already doing what you can to reduce your workload.

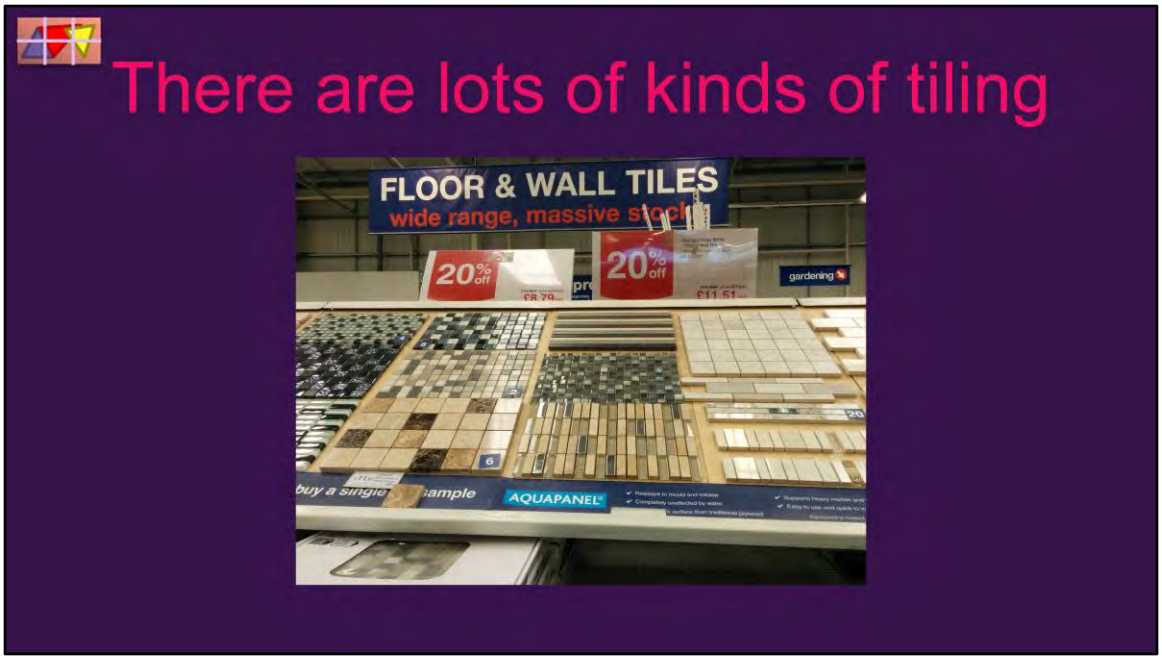
Mobile Architecture Features

- Most mobile chips have:
  - Tiled GPU architectures
  - Unified RAM
  - Several CPU cores
- APIs like Vulkan expose these features
  - Lessons from Vulkan apply in other APIs

Mobile graphics processing units are usually fundamentally different from classical desktop designs.

* Mobile GPUs mostly (but not exclusively) use tiling, desktop GPUs tend to use immediate-mode renderers.
* They use system RAM rather than dedicated local memory and a bus connection to the GPU.
* They have multiple cores (but not so much hyperthreading), some of which may be optimised for low power rather than performance.
* Vulkan and similar APIs make these features more visible to the developer.

Here, we're mostly using Vulkan for examples – but the architectural behaviour applies to other APIs.



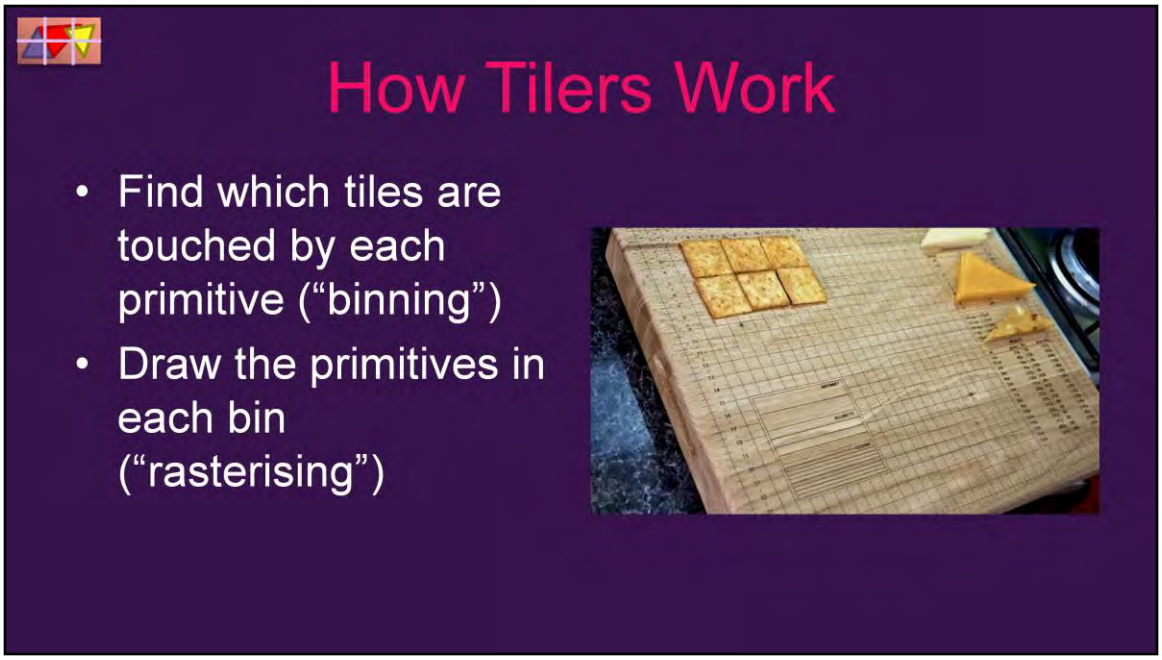
Let’s start with the tiled renderer.

There are many approaches to tiling, even within a company.

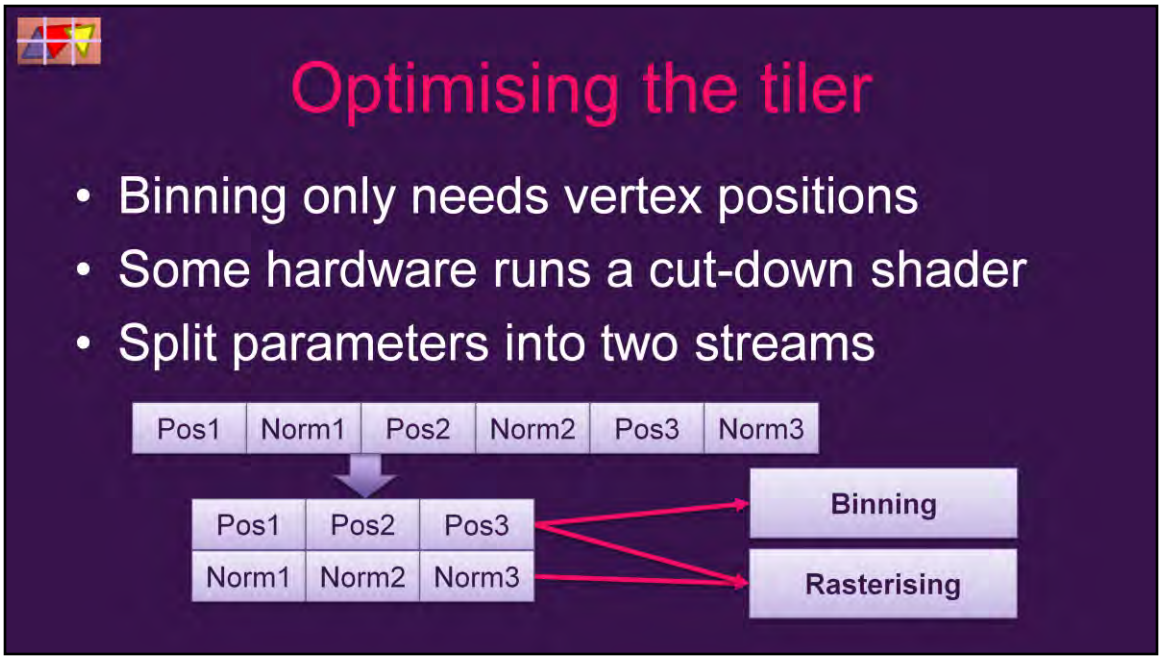
Hence lots of subtle behavioural differences.

This entire course comes with a big disclaimer: everyone’s hardware is different, everyone’s rendering flow is different, and you’ll see different benefits from different platforms.

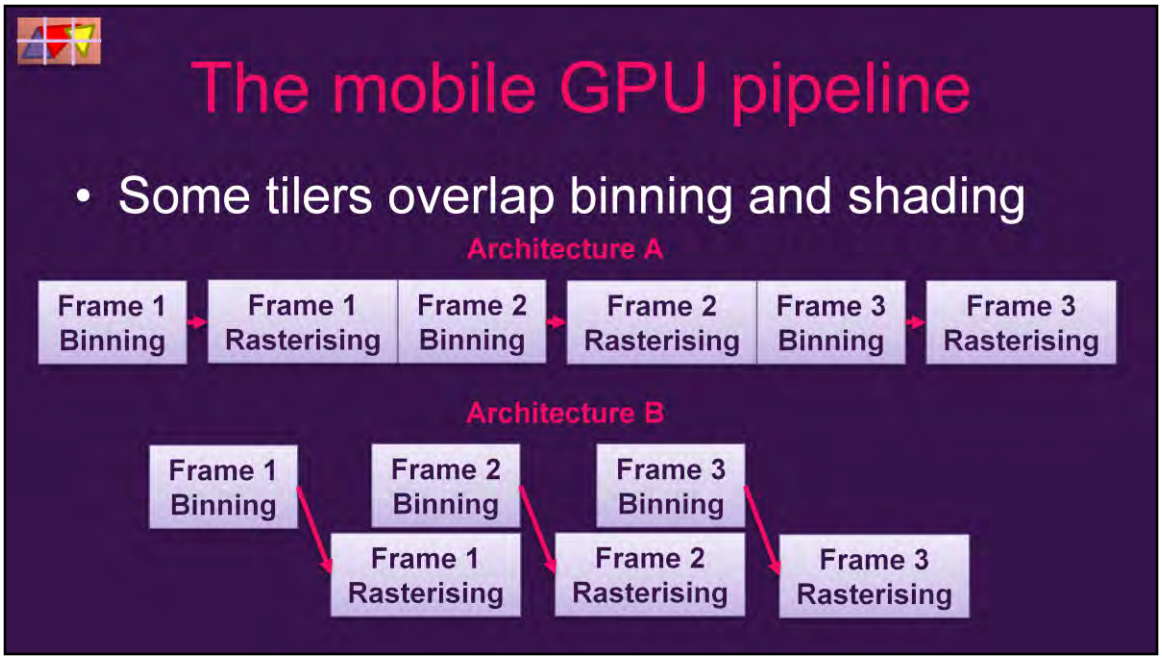
Still, some general approaches work well on most tilers, and that’s where we’ll focus here. Recommendations by individual vendors can get you even better performance if you don’t care so much about portability.



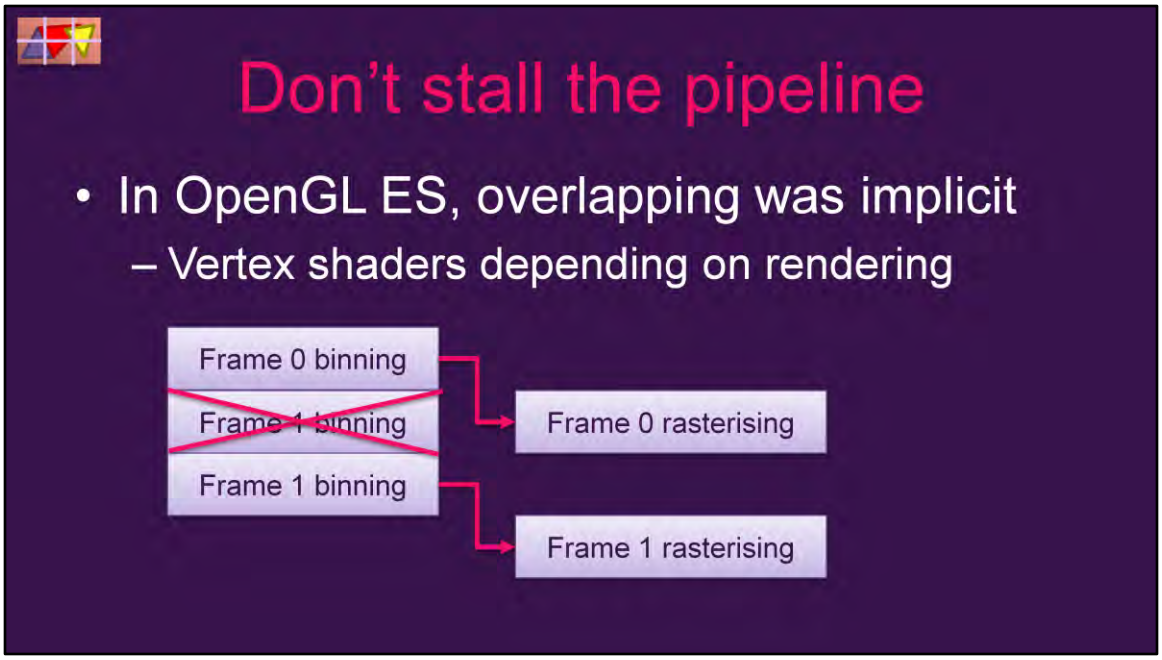
A classic immediate mode renderer just draws the primitives in the order they’re submitted.
Tilers first perform a ”binning pass”, working out which tiles are affected by each primitive. These are the European cheese mountains.
Each triangle is tested in turn, and in some way the triangles associated with each tile are recorded (here we just stack them).
* Then each tile is processed in turn, looking just at the primitives that affect that tile and the area where that tile intersects the primitive.
Rasterising, including overdraw and Z comparison, is handled in fast on-chip memory before writing out the final tile contents to main memory.



We want the tiled rendering to be as efficient as possible.
* One observation is that the binning pass only needs vertex positions, not anything else evaluated in the vertex shader.
* Some hardware uses a cut-down vertex shader in the binning pass that only evaluates this, and doesn't need any other vertex shader parameters.
* Applications can optimise this by splitting vertex shader inputs into those needed for position and everything else.



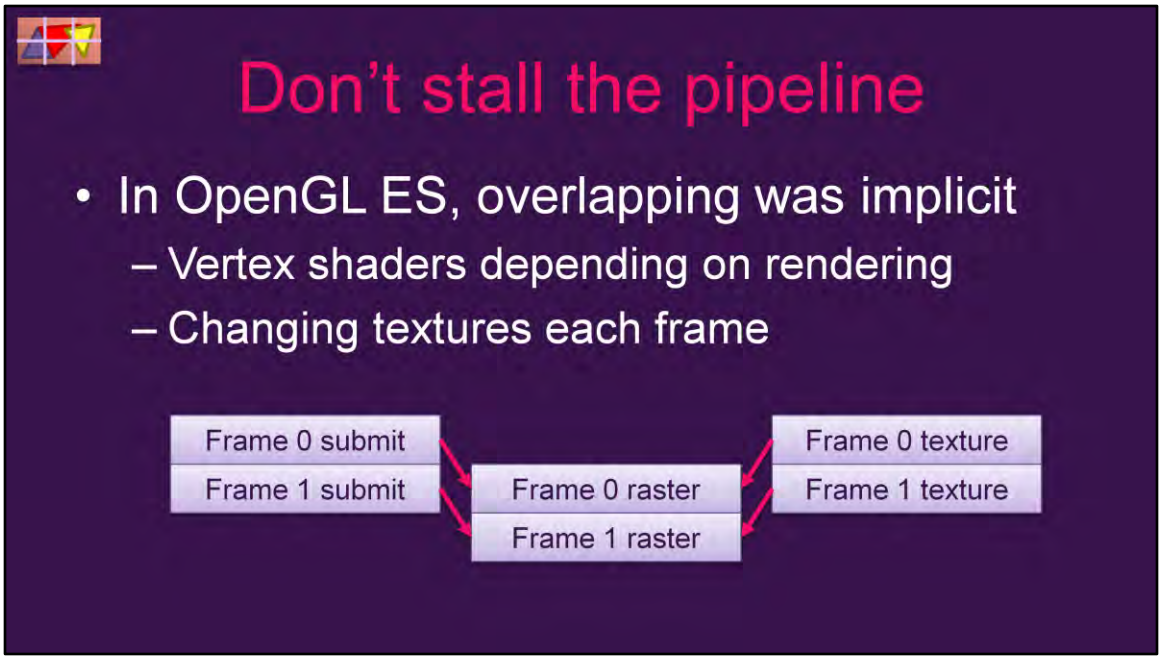
Several mobile GPUs perform the binning pass of one frame while performing the rasterising pass of the previous frame.
This allows typical ALU-heavy vertex operations to overlap with texture-heavy fragment operations, and keeps the GPU busy.
In GLES and similar APIs, dependencies between passes are implicit.
In APIs like Vulkan, the pipeline is more explicit.



This pipelining of the binning and rasterising stages only works if the user avoids dependencies – something that was potentially easy to do in an API like OpenGL ES.

Making the binning pass of one stage dependent on the content of the rasterising stage causes a stall.

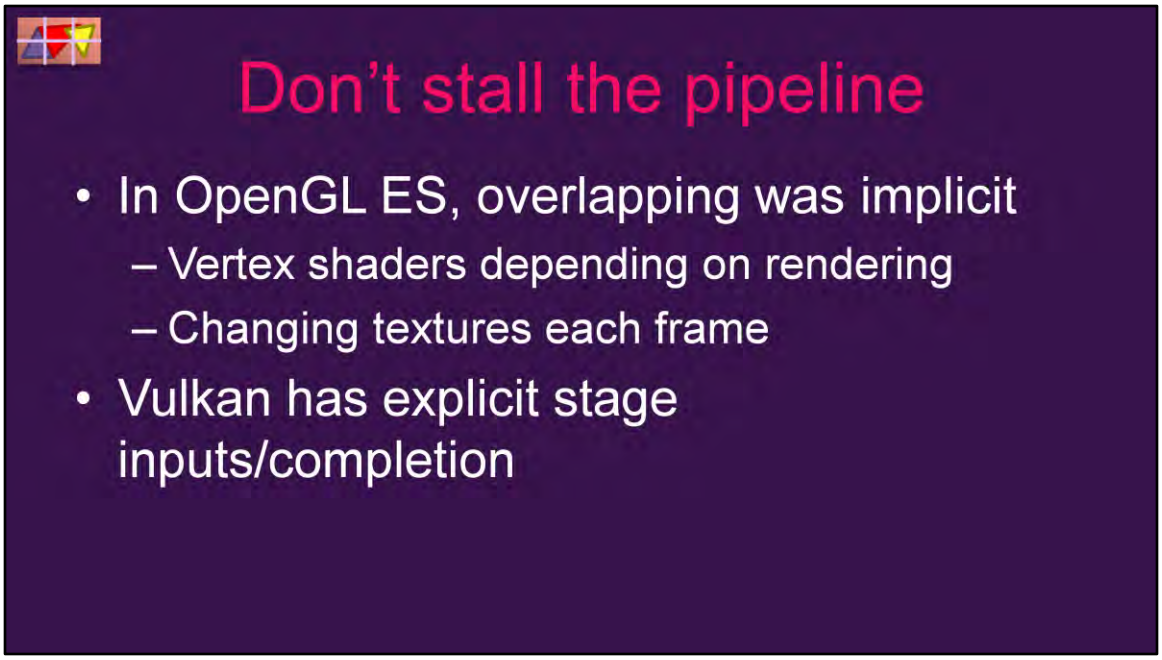
Fortunately, this is rare.



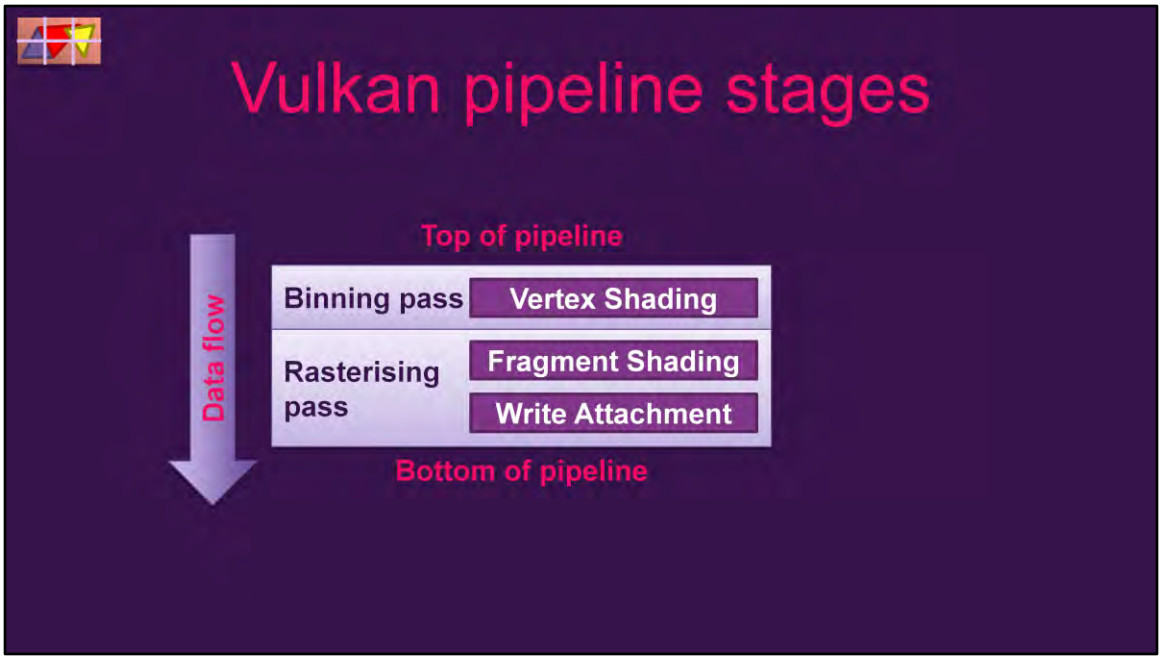
More likely, if you try to update a texture every frame, the rasterising of that texture is happening one frame behind – so the wrong thing would be rendered by a naive implementation.

If you do this, in order to get correct results, the driver must either stall or – more likely – make a copy of the texture as it was before your modification.

This can be costly, and the preferred solution is for the application to double-buffer texture use.



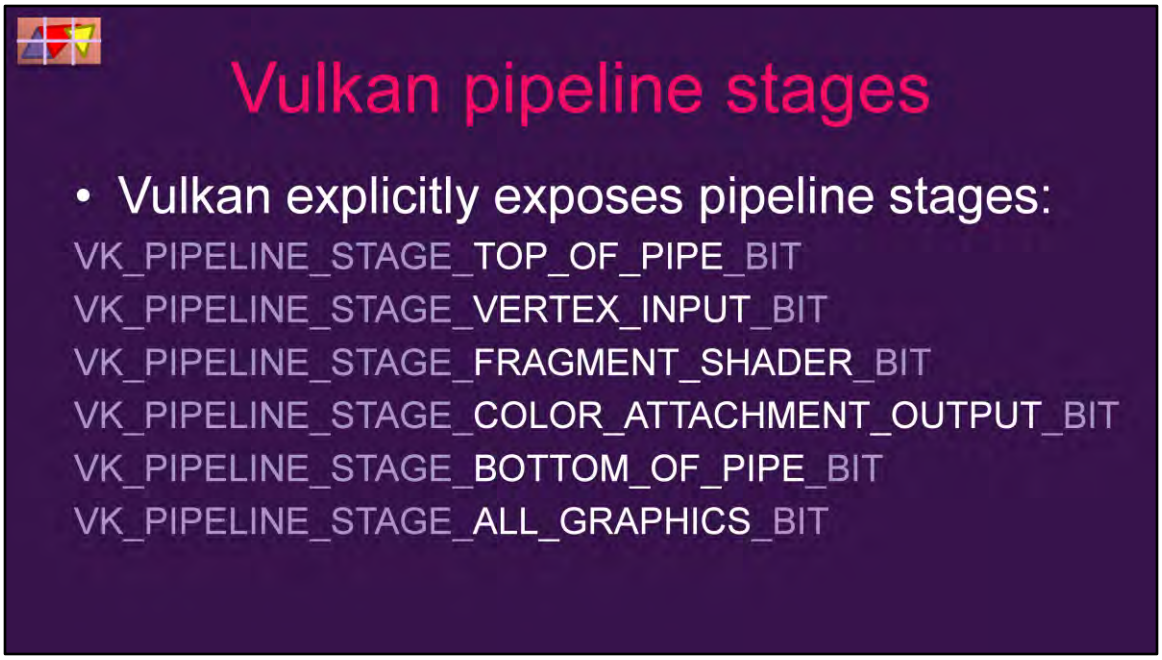
In Vulkan, the dependencies between stages are more explicit, and you can tell when a stage is done with its inputs.



On a conventional tiler, we can think of vertex shading needing to happen in the binning pass.

In the rasterising pass, we have fragment shading happening, and then the data in the tile is written out to memory.

Vulkan lets an application stall the operation of one of these stages until its resources are ready, or wait on the output of a stage.



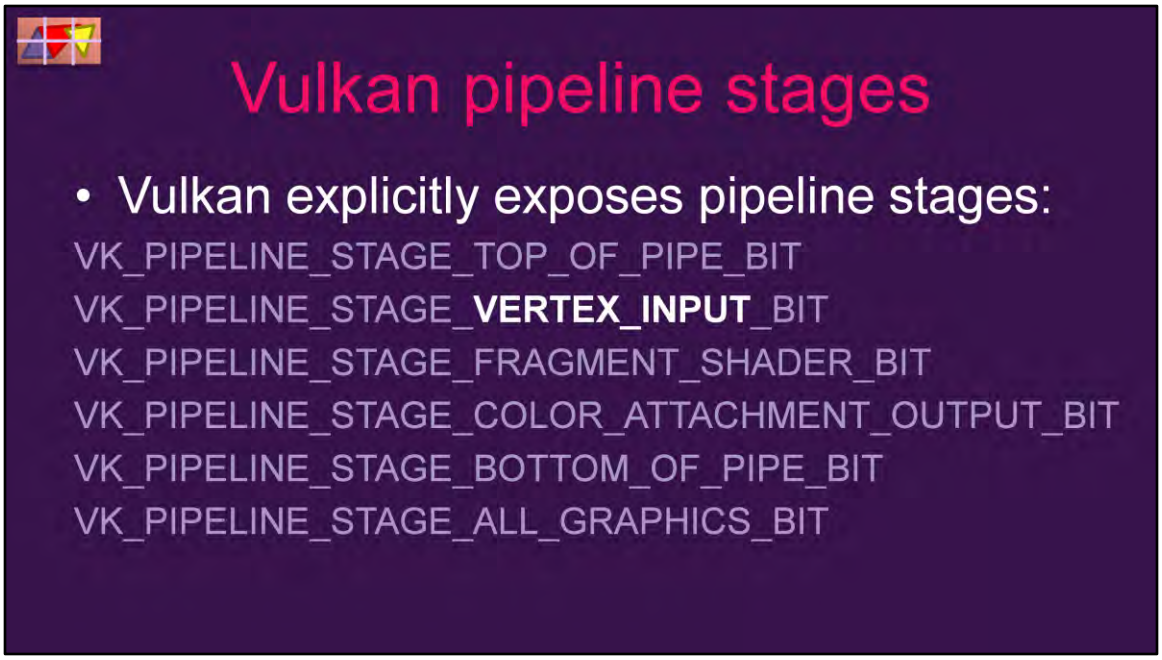
Vulkan makes the concept of pipeline stages explicit.

How each of these are implemented may be radically different, especially between a tiler and a rasteriser.

These stages can be used to control the flow of rendering to know when a pipeline stage can begin and when it has completed.

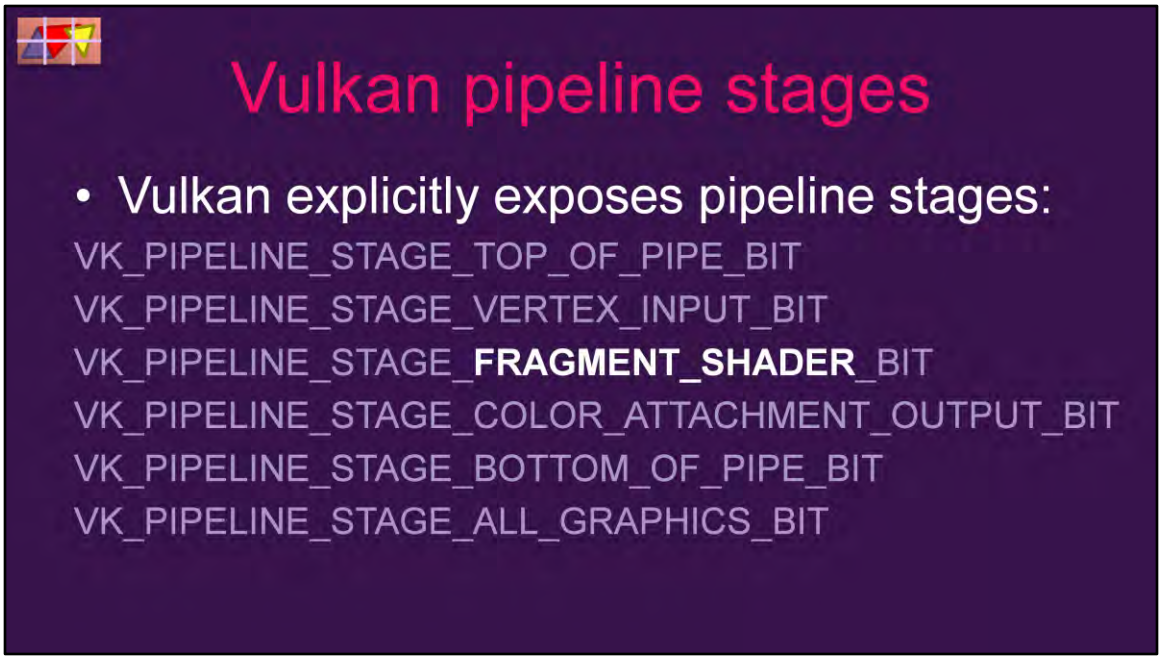
One a tiler, this allows you to distinguish the binning and rasterising passes.

There are also other stages for compute, fragment discard, resolving and, potentially, for tessellation and geometry shading which I've not included for this discussion.

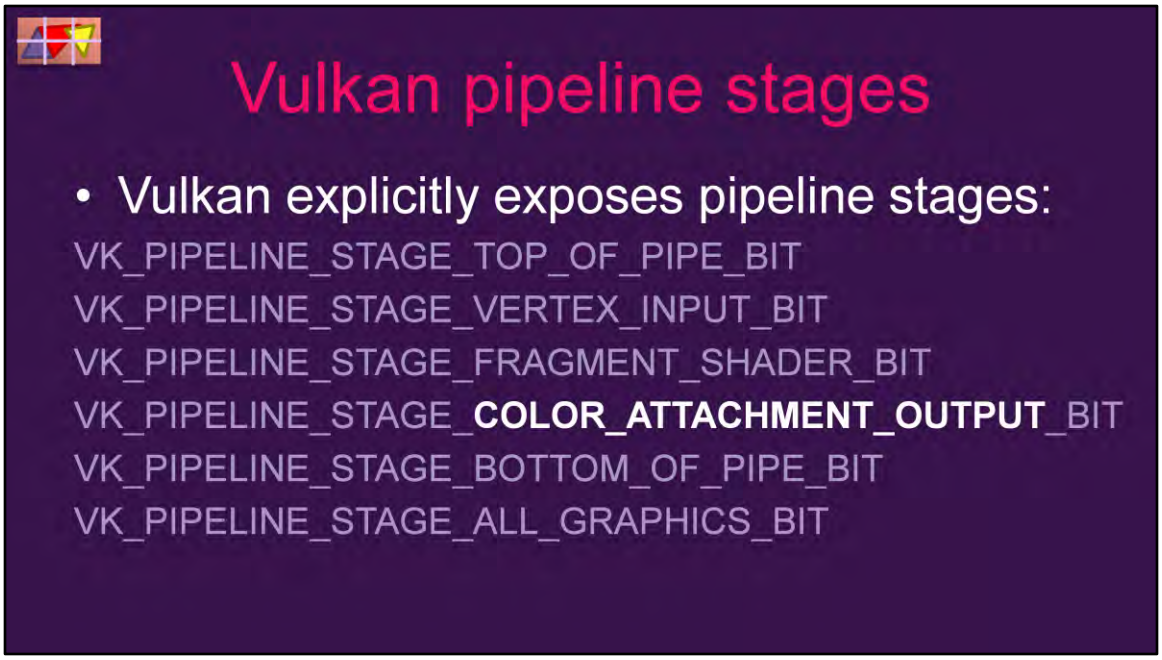


The interesting ones here are:

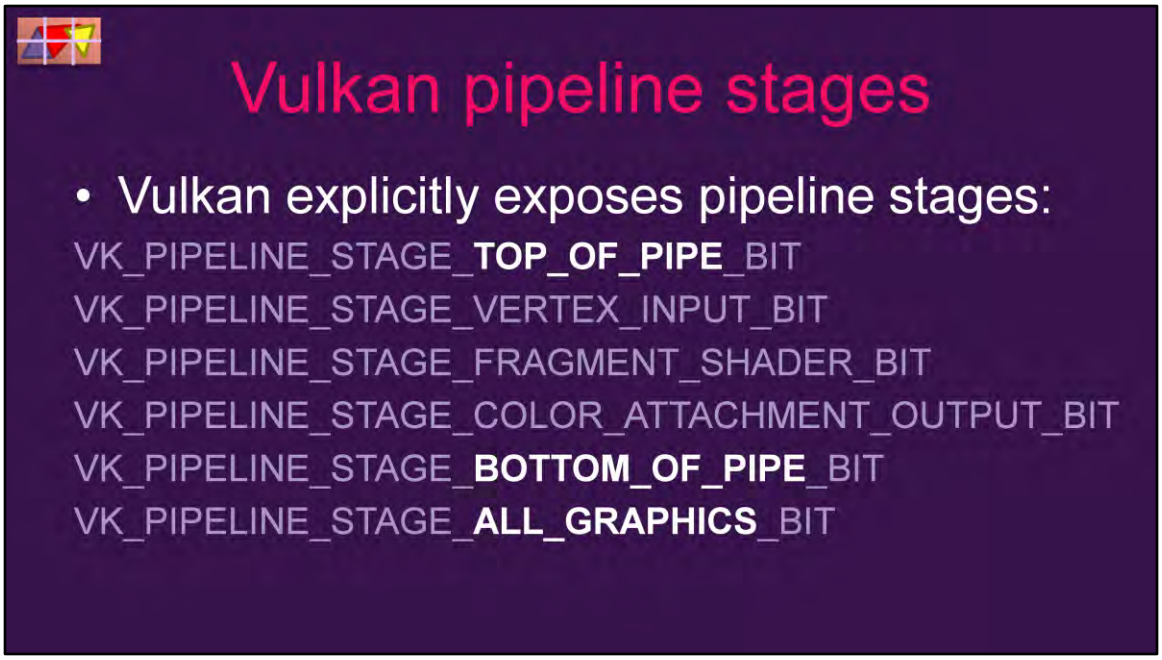
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT (you might want to delay the beginning of vertex shading/binning).



VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT (you might want to delay fragment shading until the results of input textures have been calculated).

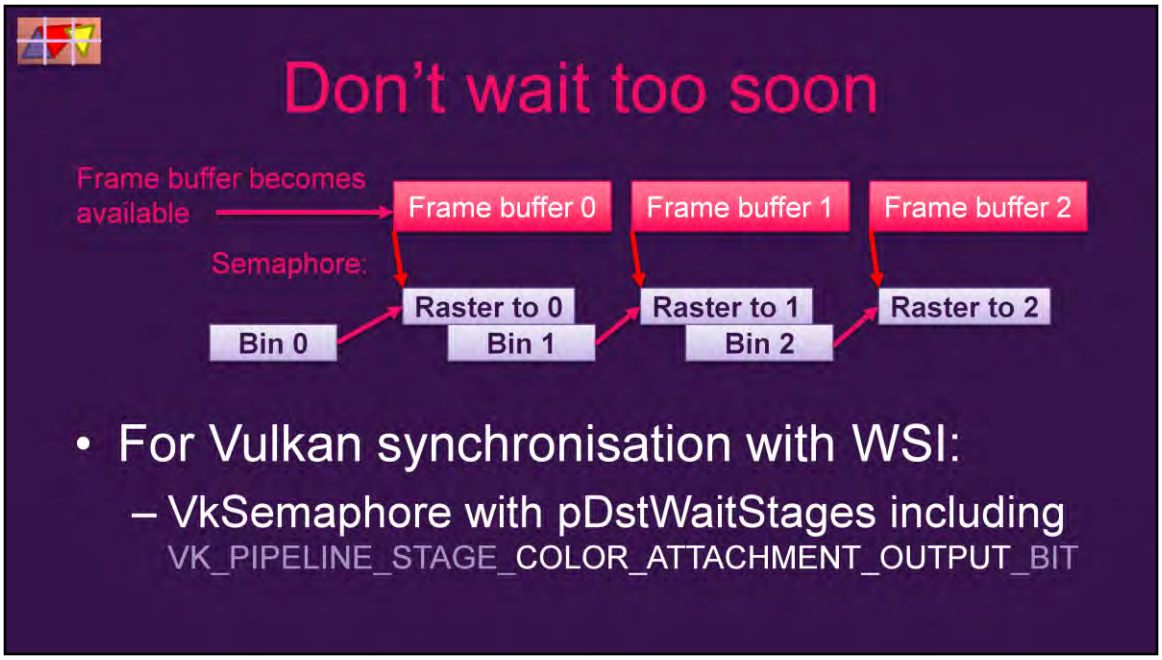


VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT (you might want to delay writing to the frame buffer until it is available, or know when rasterising is complete)



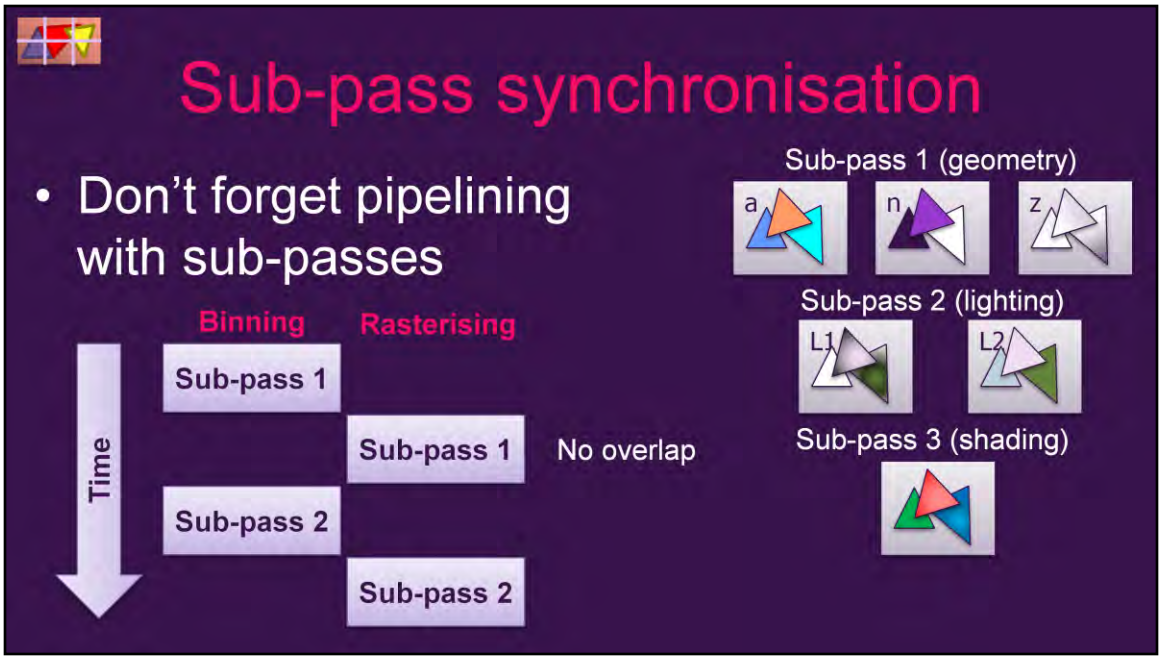
"TOP_OF_PIPE", "BOTTOM_OF_PIPE" and "ALL_GRAPHICS" are catch-all flags where the details of the pipeline don't need to be known.

While they're often equivalent to other explicit stages, it's better to control the flow at exactly the level you need.

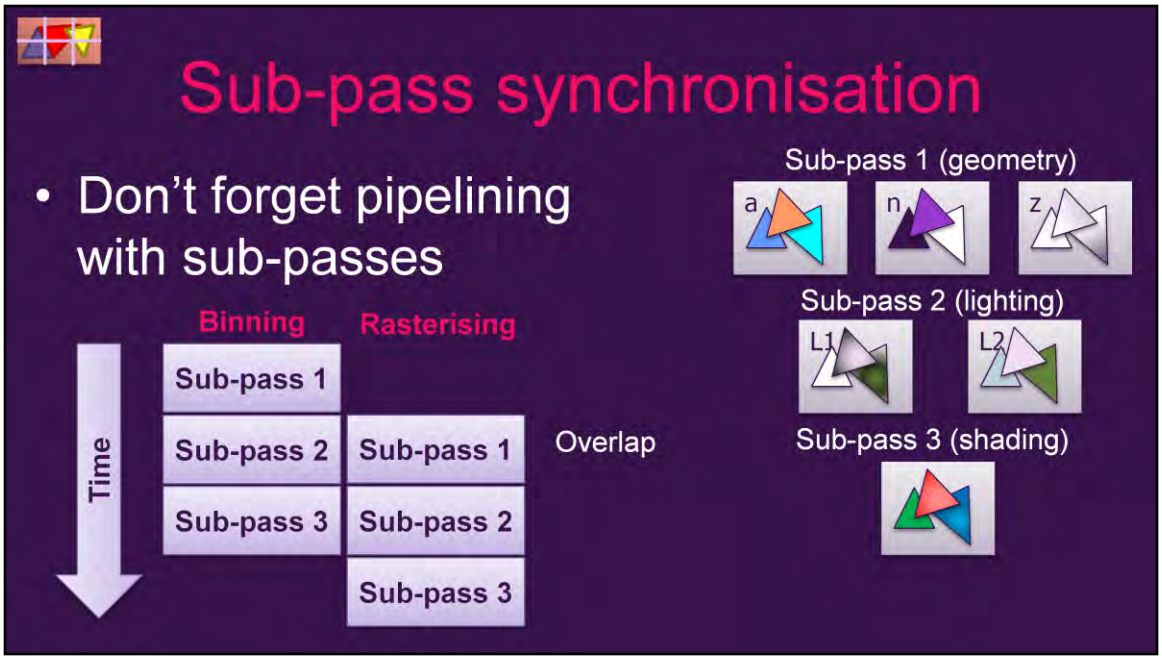


The speed of rendering in a typical application is gated on a new render target becoming available.
The binning pass does not need the frame buffer to be available – so binning can be complete before the render target is free, and only rasterising needs to wait.
vkQueueSubmit associates semaphores with each command buffer that is submitted.
You can make pipeline stages wait on a semaphore, not just the entire command buffer – and this is how you can get rendering to begin, but wait at the rasterising stage.

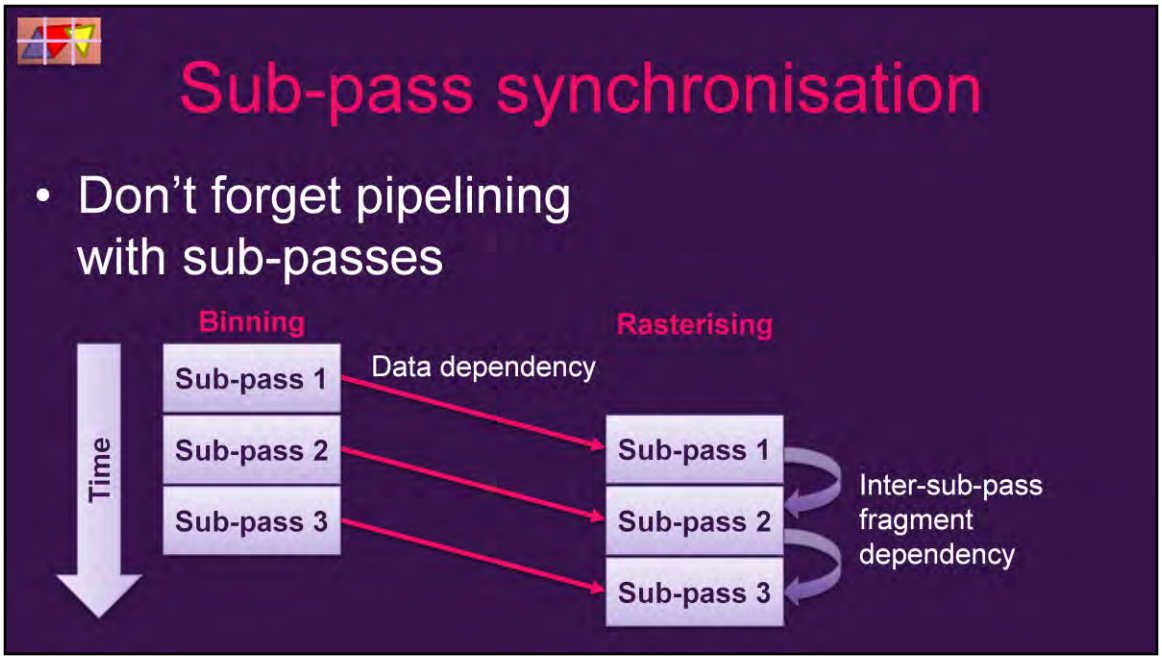
For example, the application may be acquiring render targets from the window system integration layer’s swap chain – which provides a vkSemaphore when a new render target is available.
Maximum throughput comes from queueing up a lot of work – but this needs to be balanced against latency.



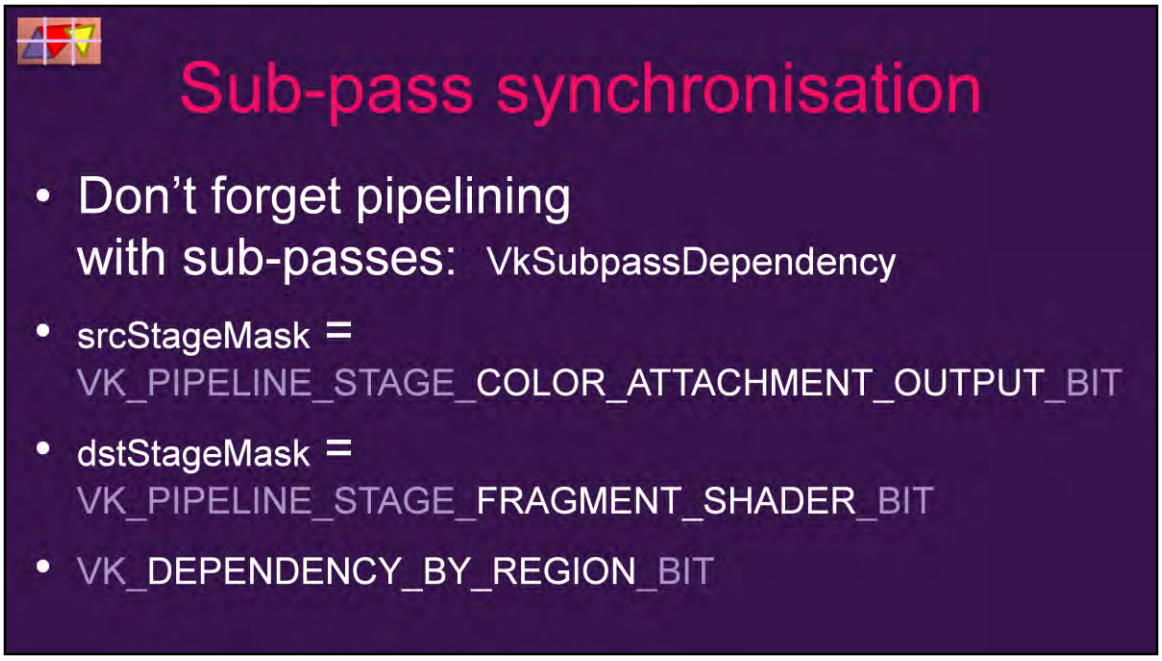
Sub-passes (which we'll revisit later) can also overlap binning and rasterising, on GPUs which use this approach – but we need to tell the pipeline stages how to do this or we may get sequential ordering.



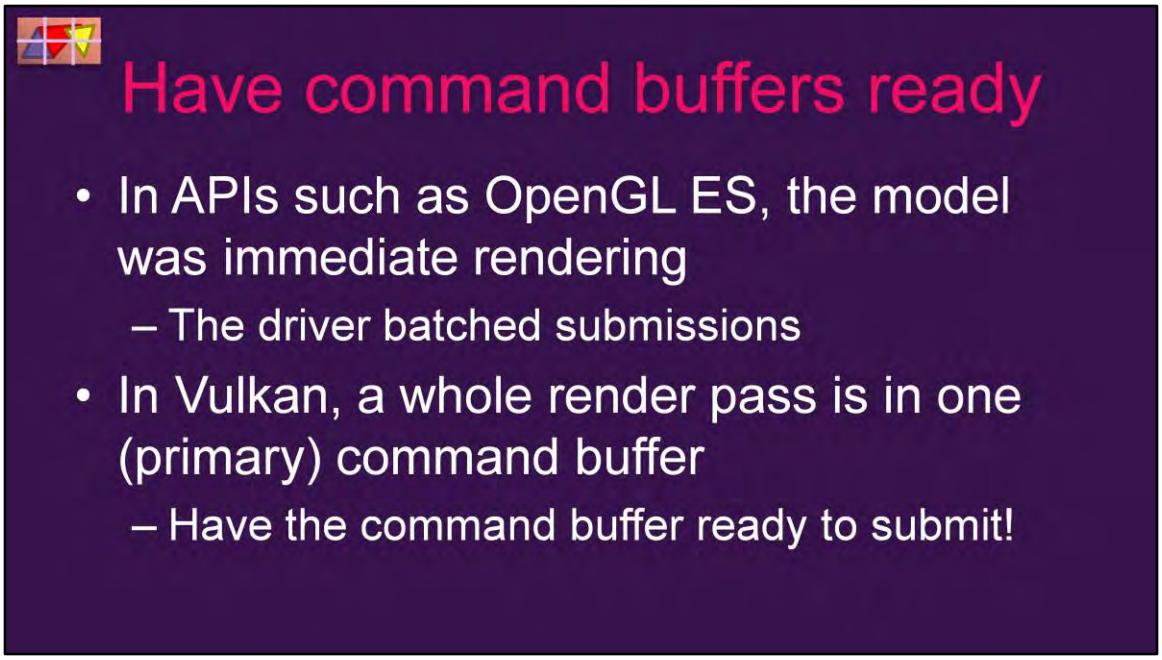
For good performance, overlap binning and rasterising, on GPUs which use this approach.



We need the binning pass to complete before the rasterising pass which depends on it (this is implicit in the pipeline behaviour).
We also need to ensure that the previous sub-pass has finished its rasterisation before a subsequent pass which depends on it can begin.
Dependent sub-passes require pixel data to be available (or they wouldn't be dependent).



For this reason it’s important for performance that dependencies between sub-passes only wait at the rasterisation stage.
In this case, we don’t allow fragment shading to begin until the previous sub-pass has written its intermediate data to the attachment, which ensures it can be accessed.
The dependency is like the semaphore on command buffer submission:
The source is where this semaphore is triggered (the pipeline stage whose completion we’re waiting for).
The destination is what is waiting on this semaphore (the pipeline stage which cannot begin until the dependency is satisfied).
Also, use VK_DEPENDENCY_BY_REGION_BIT in the subpass dependency flags where possible, since you’re worried about dependencies within the tile.



OpenGL and similar APIs had the model that when you ask for a primitive to be rendered, it will happen automatically.

Practically, this means the driver batches draw calls heuristically.

The GL driver doesn't know when you're going to be done with submitting work, so these heuristics can go wrong.

The binning pass can be processing as batches of work are submitted.

Rasterising then begins when you call glSwapBuffers, glReadPixels, or equivalent (not always on glFlush).

Command buffers are recorded explicitly in Vulkan, since the application should know how much work is still to come.
It is up to the application to work out how much rendering work to build into a command buffer and how many command buffers to use – this is heavily dependent on the application.
Command buffer recording is potentially slow, so where possible have command buffers built before binning begins.

Vulkan command buffers are submitted explicitly by the application – they're not

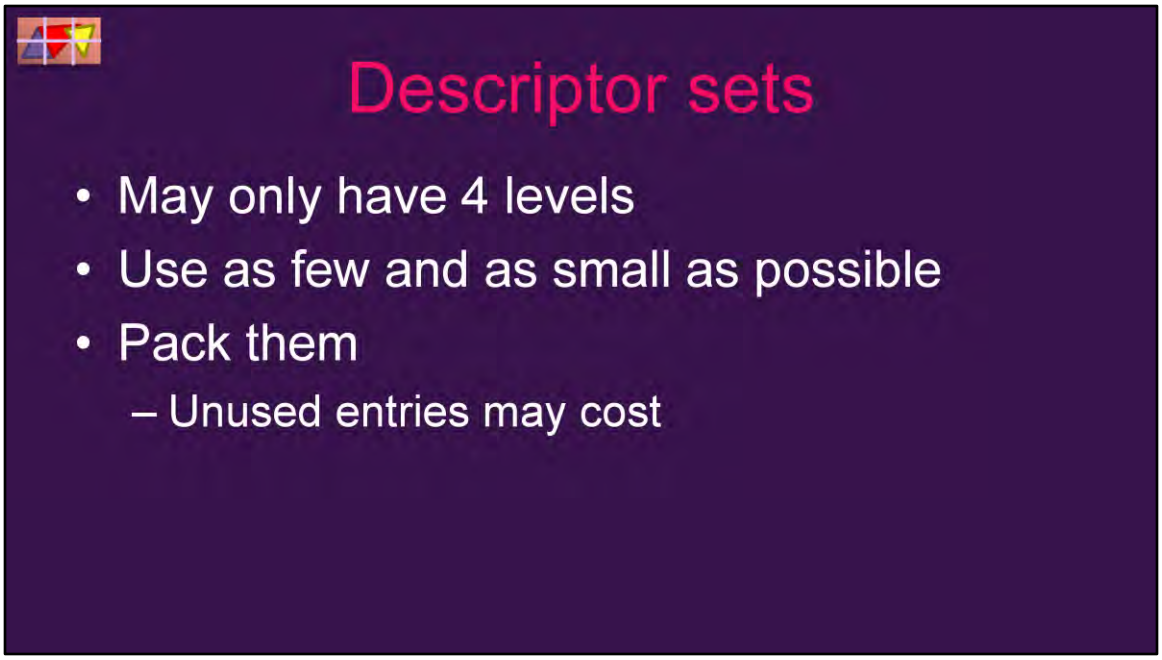
submitted incrementally.

Command buffer recording has the potential to be costly, so where possible the application should begin recording early.

Binning can’t happen until the command buffer is submitted.

On a tiler, a render pass might be the unit that ties together binning and rasterising – and a render pass has to be within a single primary command buffer

To get the binning working early (and all the vertex shading work that this involves), build the command buffer while you’re waiting for the next render target, so you can vkQueueSubmit early.



Vulkan, like some other recent APIs, has a hierarchicy of descriptor sets, with the intent that the most frequently-changing resources can be swapped out without needing to change the full configuration.

Vulkan only guarantees the presence of four descriptor set levels, and some mobile architectures only implement this minimum.

Using more descriptor sets than necessary carries an overhead; large descriptor sets equally can have an overhead.

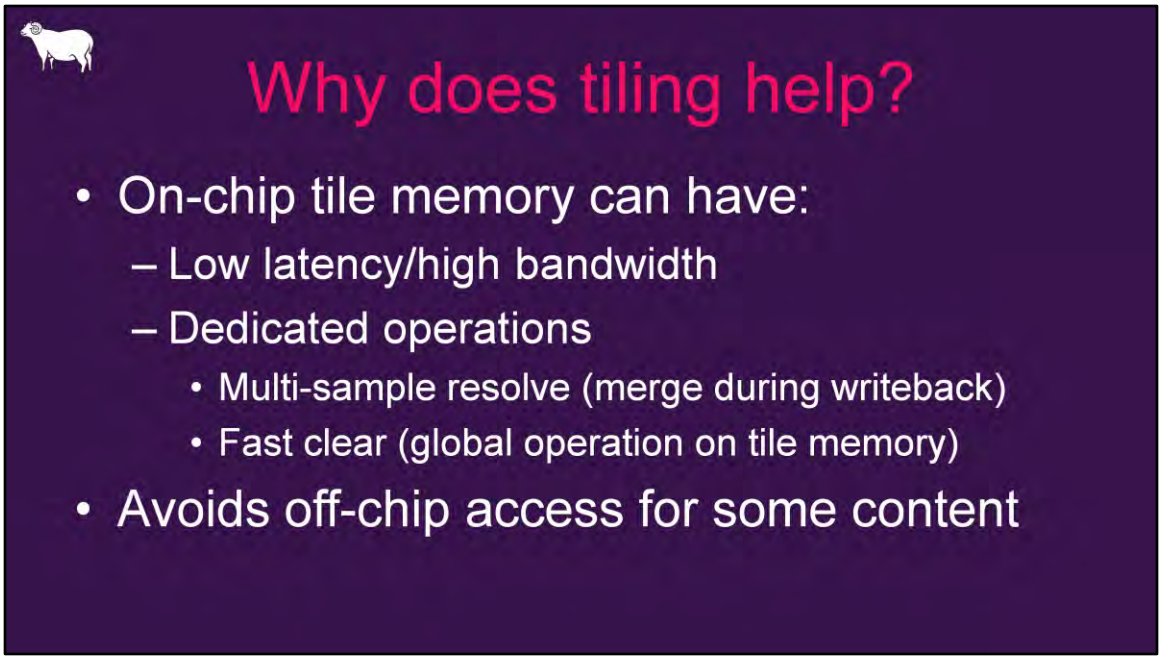
While some architectures, particularly on desktop, don't see much disadvantage to unused entries in descriptor sets, on mobile systems it is often beneficial to keep descriptor sets tightly packed and remove unused entries.
As ever, the details of each mobile architecture is different.
Specifics of behaviour – for example, the merits of push constants may vary depending on the target system.

This is, of course, a massive generalisation. (Why a massive generalisation? Just big horse.)
Actual hardware varies even within a manufacturer, and performance effects can be heavily masked by other costs.
There's no substitute for profiling on as many kinds of hardware as you can.

RAM

On to RAM...

The GPU has a lot of bandwidth to internal tile memory.
The moment you access external memory, that high speed data flow slow down.
Mobile GPUs have much less bandwidth than desktop systems.
Partly this is down to space for the connection, but it's largely down to power – the amount of bandwidth relative to calculation is not so different from desktop, but there is more power cost to using memory in mobile.
One of the Samsung team leads quoted a few years ago that it takes tens of times more power to toggle an external connection than an on-chip one, so external bandwidth has serious effects on power consumption.
The reduced resources available to a mobile device might also reduce the ability to hide latency if RAM was being accessed excessively.

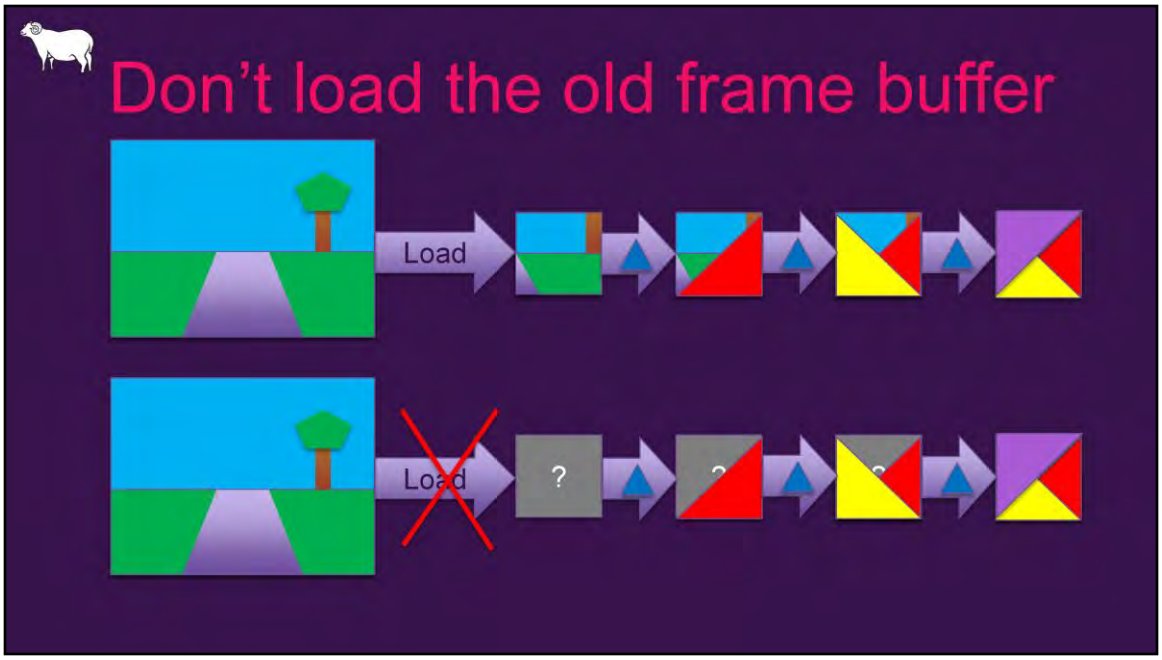


Tiling architectures make as much use as possible of on-chip dedicated frame buffer memory.

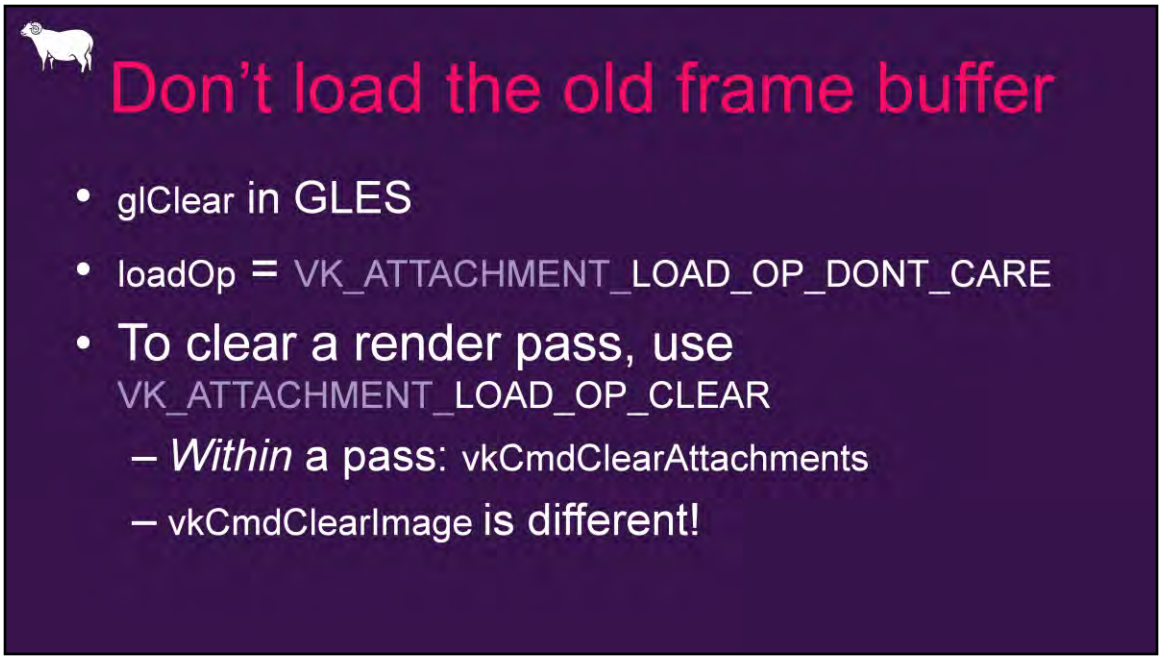
By working on a small section of the frame buffer at a time, access to the off-chip main frame buffer (which still needs to exist – space is limited!) is greatly reduced.

Once you have dedicated tile memory, you can give it dedicated hardware for operations such as multi-sample resolving and clearing

Sometimes content is not needed except as temporary scratch space during the rendering operations, and bandwidth for this content can be removed.

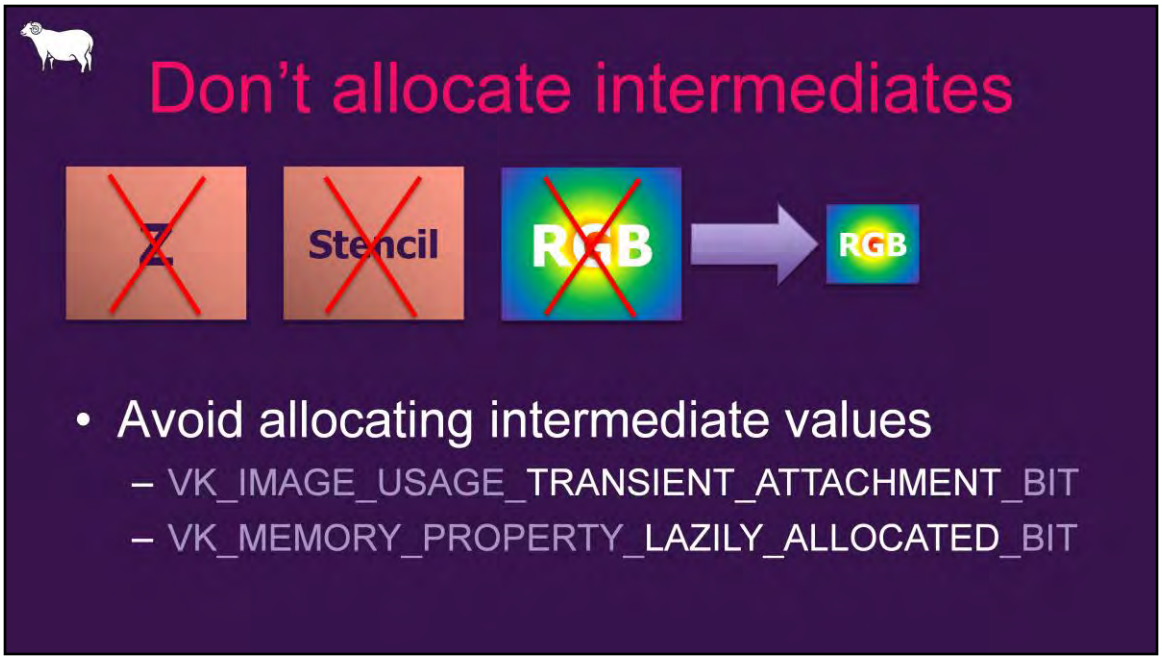


It’s important to minimise the access to RAM, particularly of the frame buffer. Unless you really need to, make sure you don’t load the prior frame buffer contents out of memory, because this uses a lot of bandwidth.

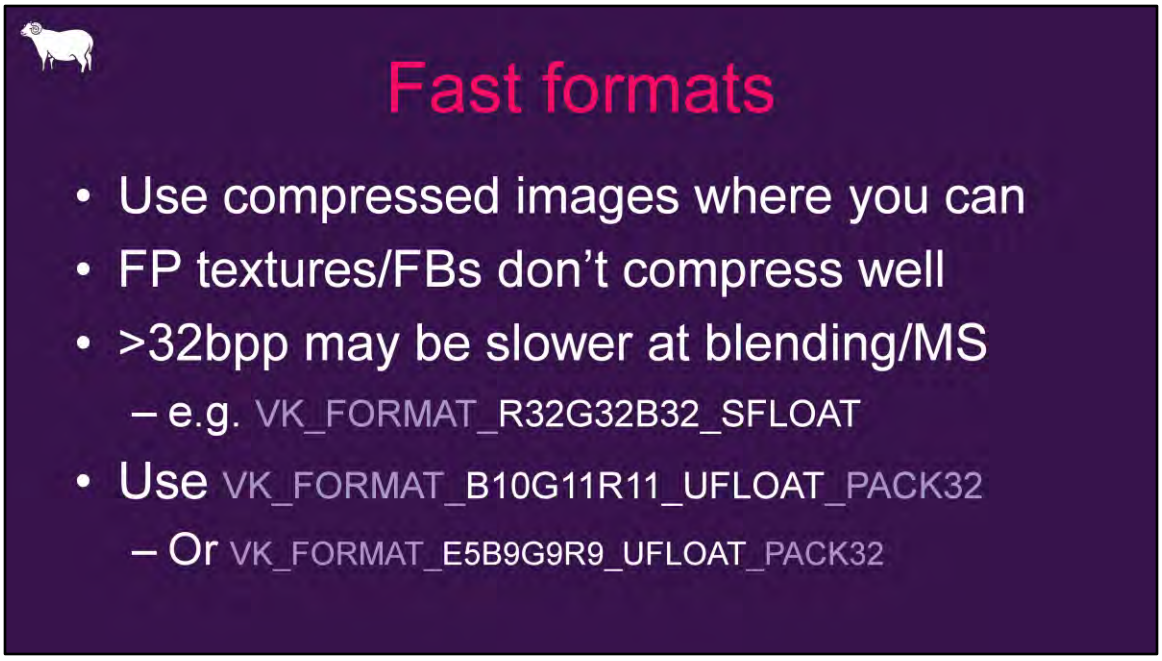


In OpenGL ES, the way to do this is with a glClear at the start of the frame.
In a Vulkan render pass, specify a clear (if you care about the colour) or DONT_CARE (if you don't) load op for the attachment to stop it being read from memory. This clear is usually very efficient. Don't load the old value unless you really need to.
If you're dealing with combined depth/stencil, remember that there is a separate stencilLoadOp.
If you want to clear an attachment part way through a render pass, use vkCmdClearAttachments (which should still be fast).
In Vulkan, don't use vkCmdClearImage (which explicitly clears an image) for this purpose – this will write to the entire image separately. (Don't confuse this with glClear despite the name!)

Similarly, if you don’t need data after rendering – for example depth and stencil – make sure you discard it at the end of the render pass.
Remember this is full-resolution (not down-sampled) data, and can be big.
Extensions make this possible in OpenGL ES, and can significantly help performance there.



To avoid wasting memory, mark intermediate images as transient and allocate them with “lazily-allocated”.
The transient attachment bit indicates that the *image* will be used for on-chip rendering and does not need backing store in off-chip RAM.
Lazily allocated indicates that the *memory allocation* will be used for transient attachments, so the entire allocation may not need to happen on a tiler.



Texture compression is an obvious way to reduce memory bandwidth – and trying to optimise locality of reference can also be important (for example transitioning an image to an optimal state for texturing).

Some architectures use compressed frame buffer formats, which can further reduce bandwidth.

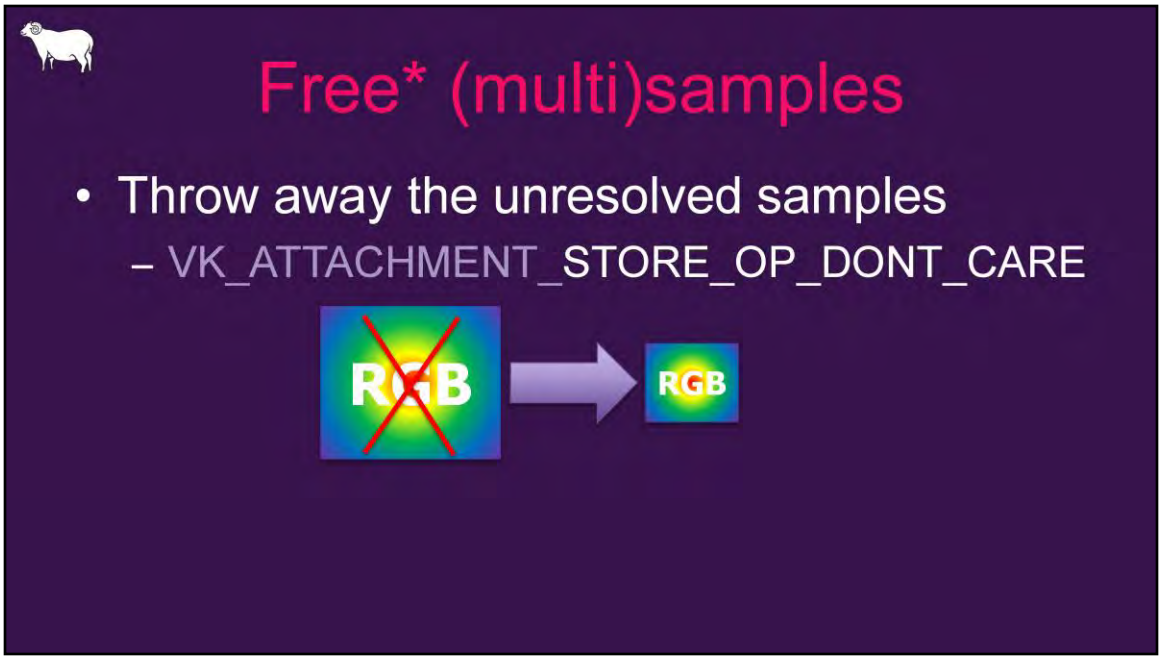
Be wary that in HDR rendering, floating point frame buffers do not compress well, and may carry a higher cost.

Larger channels may also run into hardware limitations, especially when blending or multisampling – avoiding formats of more than 32 bits can help.

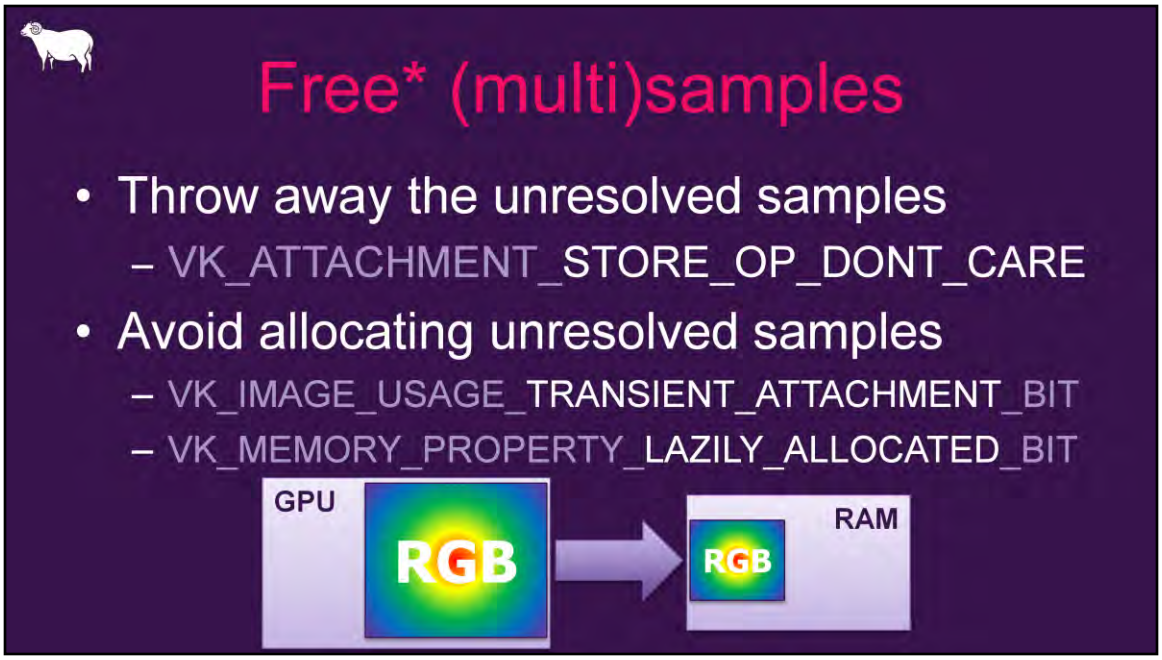
For HDR formats, consider using B10G11R11 as a way to reduce bandwidth.

For textures, the E5B9G9R9 shared-exponent format can also be useful, but most mobile architectures cannot render to this format.

Tilers typically implement multisampling by rasterising (but not necessarily shading) at a higher resolution on-chip, then ”resolving” the samples (squashing them together) to an anti-aliased image when the tile is written to memory.
This means that rendering at higher resolution for multisampling may not result in more frame buffer bandwidth – though it can still affect shader costs (so, as in many cases, “free” is a marketing message).
In Vulkan, the “resolve” attachments hold the antialiased image, and you have to configure them explicitly.

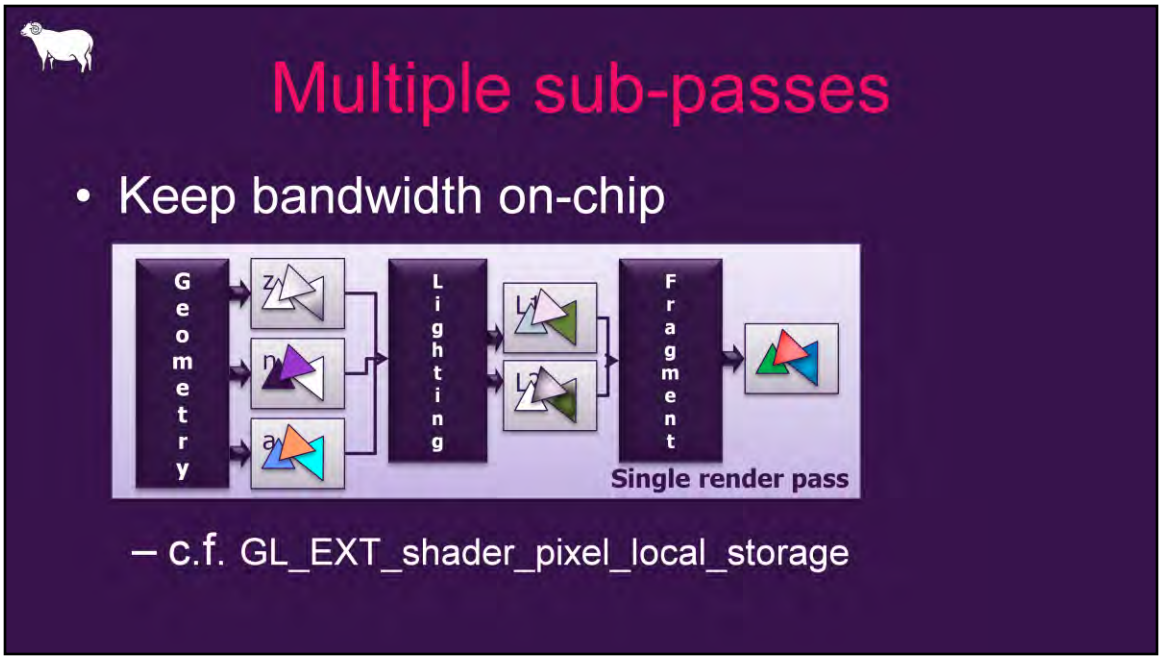


To avoid the bandwidth cost of writing the un-resolved samples to memory, mark the attachment store as ”don’t care”.
(It’s not “don’t store” because immediate-mode renderers have to write anyway.)

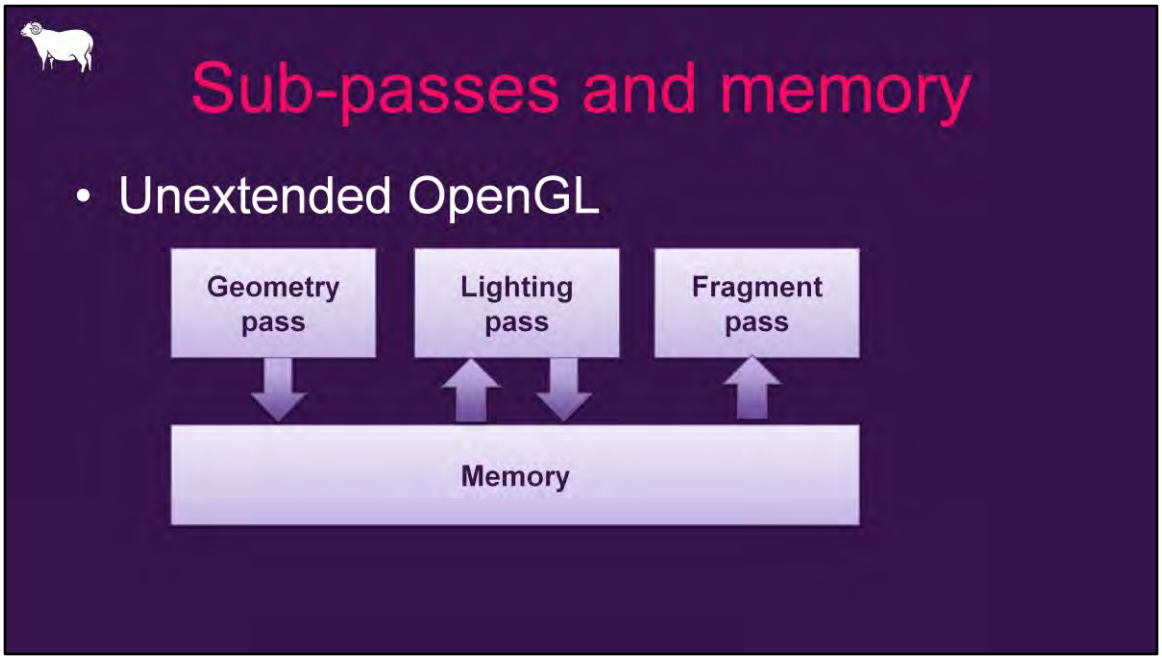


If you're never using the un-resolved samples, tilers can avoid even allocating this (large!) memory.
GLES tries to work out when it can do all this for you, but needs memory allocation just in case you access the frame buffer later.

vkCmdResolveImage reads an unresolved image from memory and resolves it to produce the antialiased version – don't mistake it for the tiler's automatic resolve!



Sub-passes allow tilers to work on more than one set of frame buffer operations within a tile without having to go out to memory.
Because only the current tile is available, each pixel can only access its own coordinates (otherwise you might go off-tile).
This is still useful for deferred shading, HDR, etc. which all rely on being able to write a value, then use that as an input to a later calculation (or series of calculations).
Sub-passes are like a more flexible version of GLES's pixel local storage extension, but non-tilers can implement it effectively (with direct memory writes) so your desktop debugger works.



On an API without a way to describe render pass dependencies, such as OpenGL if you don't make use of extensions that were introduced for tilers, each pass through an algorithm like deferred lighting – writing to the geometry buffer, performing a lighting pass and doing a final shading pass – would have to write the frame buffer out to memory, then read it back.

This is really costly on most mobile architectures.

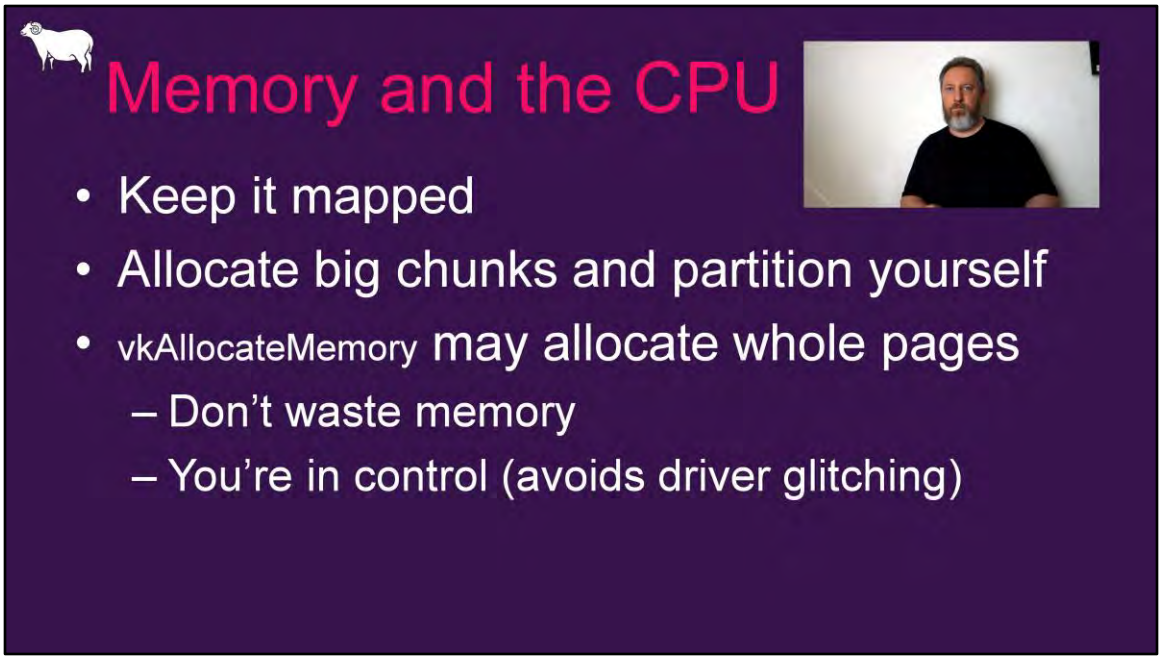
On a tiled GPU using Vulkan sub-passes, each pass can be processed without going off-chip.

As with multisampling, remember to avoid writing and allocating intermediate results unnecessarily.

Implementations may have a limit to how much you can keep in flight and (currently) supported formats – the more exotic you try to be the less likely it is to work. Expect systems to degrade gracefully, and drivers are improving.

As time goes by, expect to be able to keep more intermediate results live in an on-chip optimal manner - the dependencies can be a tree, and don't have to be a simple chain. (This is the topic of a bigger talk.)

Details of how to set this up were covered in a UK Chapter meeting at ARM.



Mapping and unmapping can be expensive on some operations.

When accessing GPU memory from the GPU, there is a cost to changing the memory mapping tables.

Where possible, keep memory mapped and rely on memory barriers to transfer data.

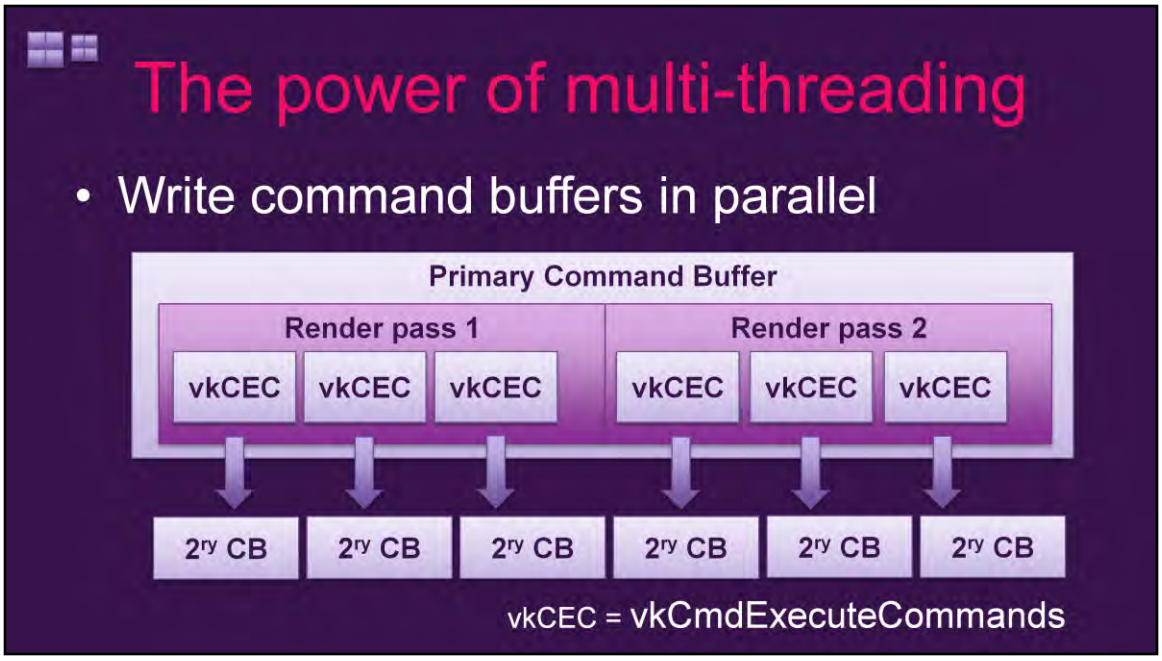
The Vulkan model is to provide control over relatively large memory allocations, with the expectation that the user will partition this into smaller regions for small textures and buffers.

Because of this expected behaviour, vkAllocateMemory might allocate on page boundaries – using it for individual small allocations is inefficient both in driver time and in memory usage.

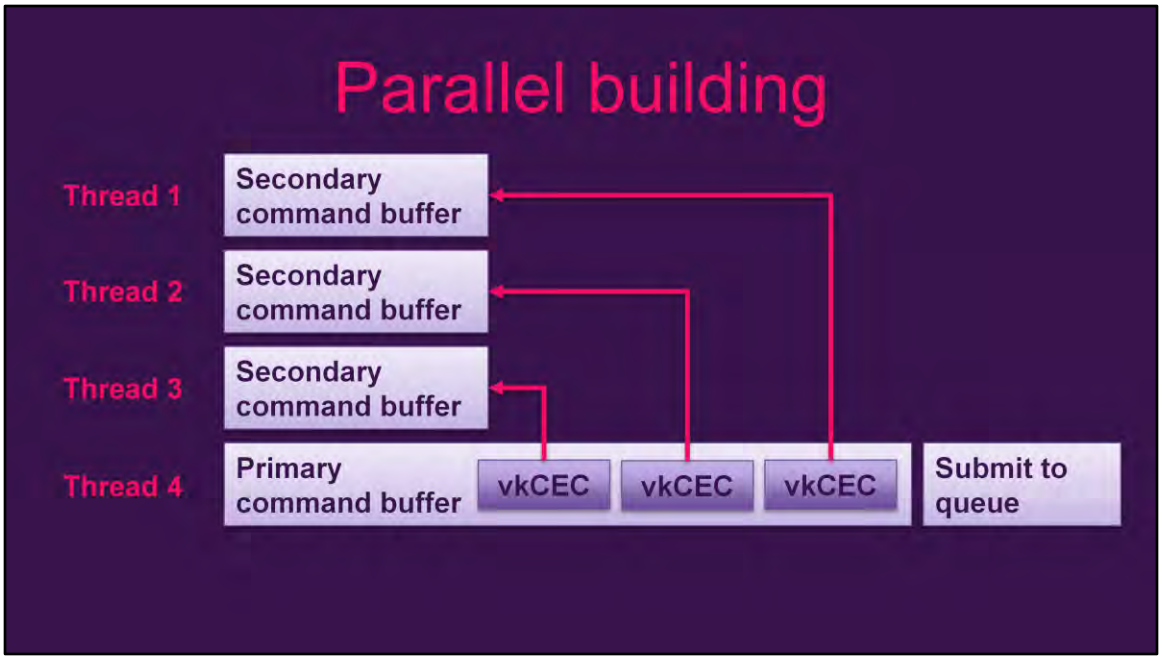
In Vulkan, you have direct control over memory use – you are responsible for allocating and deallocating, meaning that you must keep track of when resources are in use, but you can also ensure that a driver will not delay memory management to an unpredictable time.

Speeding up the CPU

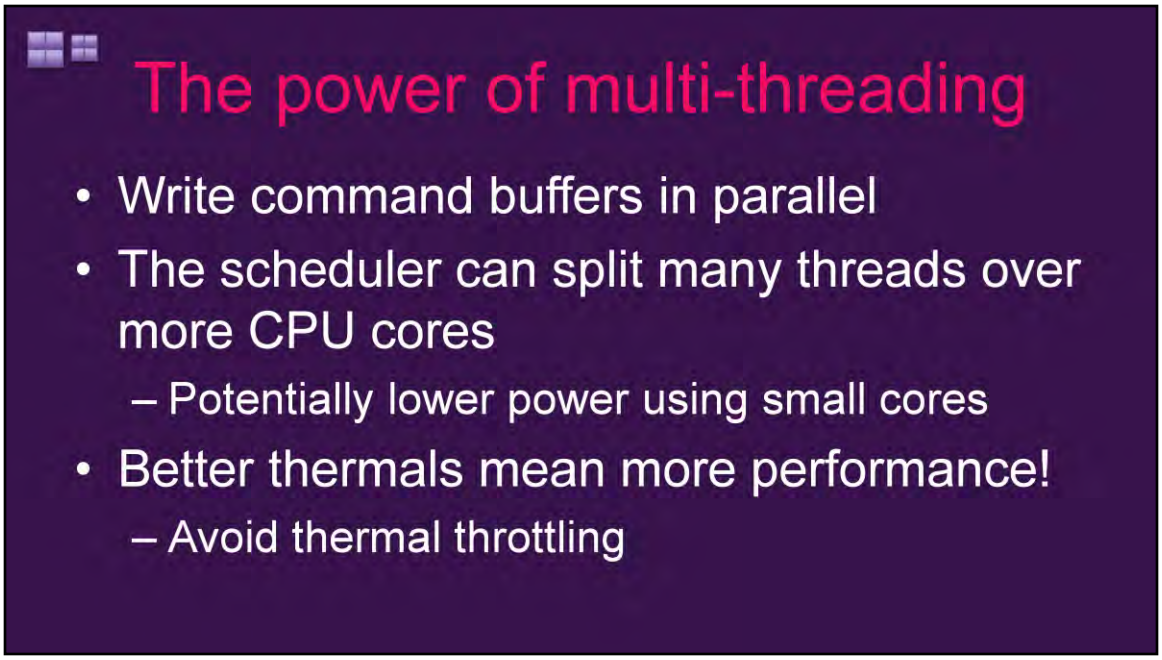
Since we've been talking about the CPU, on to the biggest improvement in next-generation APIs: reducing the CPU bottleneck.



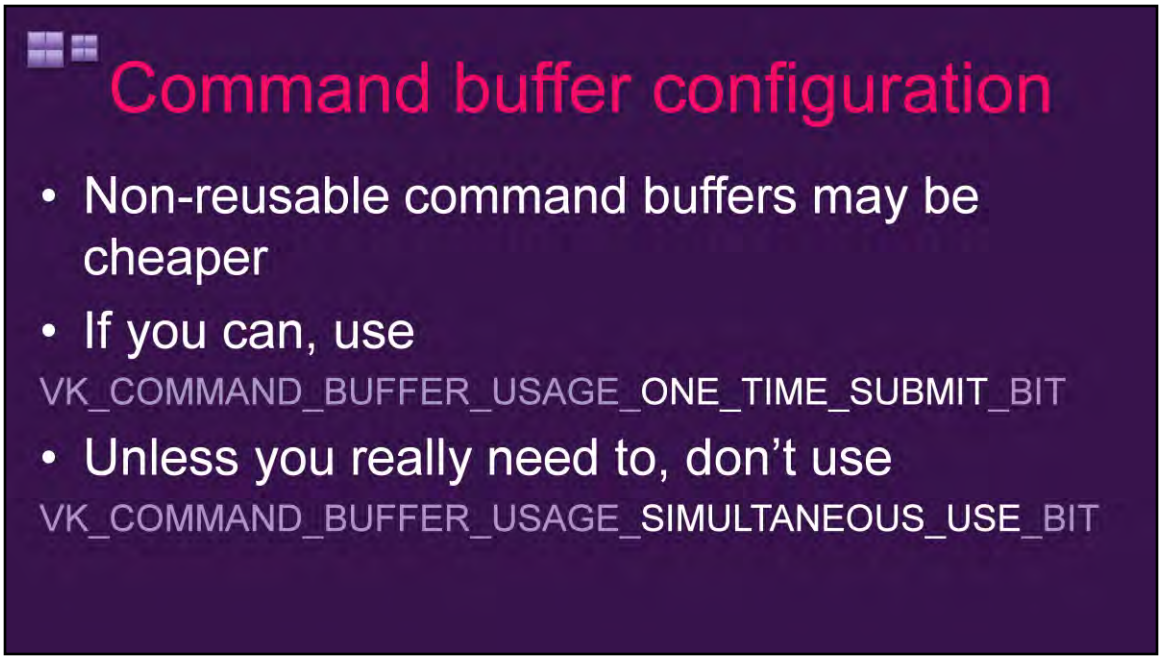
Another important feature of explicit APIs like Vulkan is that they support generating command buffers in parallel, not tied to a thread (like GLES).
Remember that a Vulkan render pass has to live inside a primary command buffer, but you can put secondary command buffers inside a render pass.
You can build the secondary command buffers in parallel and then invoke them with **vkCmdExecuteCommands** in the primary command buffer.
Or of course you can build primary command buffers in parallel if you can spread work across passes.



Don’t go mad creating lots of command buffers, but don’t be afraid to create enough to expose the parallelism in your code.
Don’t forget you can submit many primary command buffers in a single vkQueueSubmit.
Using multiple threads to generate command buffers, both primary and secondary, is a major way of distributing the driver costs in APIs like Vulkan.
While vkQueueSubmit cannot be parallelised, it should be a very lightweight operation compared with command buffer generation.
Likewise, referring to secondary command buffers in a primary command buffer is intended to be cheap.

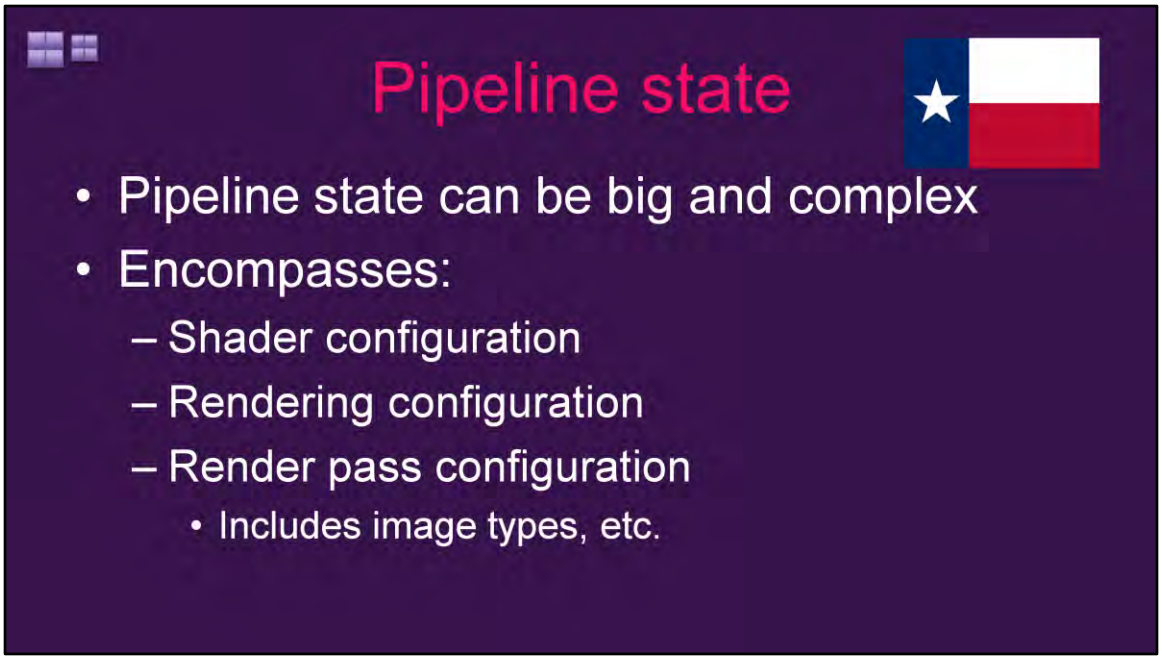


There's an obvious performance benefit to using lots of threads generating graphics commands, but on mobile that's not the only advantage:
If you have lots of threads, the CPU scheduler can spread out the same amount of work over more threads.
Ideally, what would have pegged a high-power CPU can run on low-power cores instead.
One vendor has a test that has demonstrated up to 4x improvement in power consumption by running the same work across four low-power cores.
Just doing this gives you better performance, because it stops the chip from overheating and therefore needing to reduce its clock speeds.
(Cell phones don't have fans, except for the Apple Store queue.)



Vulkan lets you reuse command buffers, if you tell it you're going to do so.
Without the ONE_TIME_SUBMIT bit set, a command buffer can be reused once it has completed executing.
With the SIMULTANEOUS_USE bit set, a command buffer can be resubmitted while it is pending execution, rather than requiring execution of that command buffer to complete first.
Some architectures benefit considerably from the ability to reuse command buffers if your rendering workload is suitably structured; for other architectures it may provide little benefit over generating a new command buffer, and it may not always be worth rearchitecting an application specifically to make use of this feature.
Using a command buffer a second time may involve the driver patching memory addresses (for example as VkImageViews change), keeping track of where this patching needs to take place, and potentially copying the command buffer – so there may be costs in the driver on some implementations.
Marking the command buffer as single-use reduces these overheads – but obviously you can't then reuse the command buffer.
Marking a command buffer for simultaneous use may, on some architectures, be implemented by making multiple copies of the command buffer (which may be needed if submission of the command buffer requires patching its contents). Again,

to avoid a potential overhead like this, avoid using the simultaneous use bit unless you need it.
It may still be quicker to reuse a command buffer if very little has changed between uses, but if there are several changes, consider re-recording the command buffer.
If you're going to reuse command buffers, try double-buffering to minimise changes and avoid the need to reuse a command buffer that's in flight.
Bear in mind that you can queue up multiple command buffers which cover different render passes in a single vkQueueSubmit.
You can still reuse the *storage* of a command buffer even in these cases, once the command buffer has completed executing.



Rendering behaviour depends on a pipeline configuration.

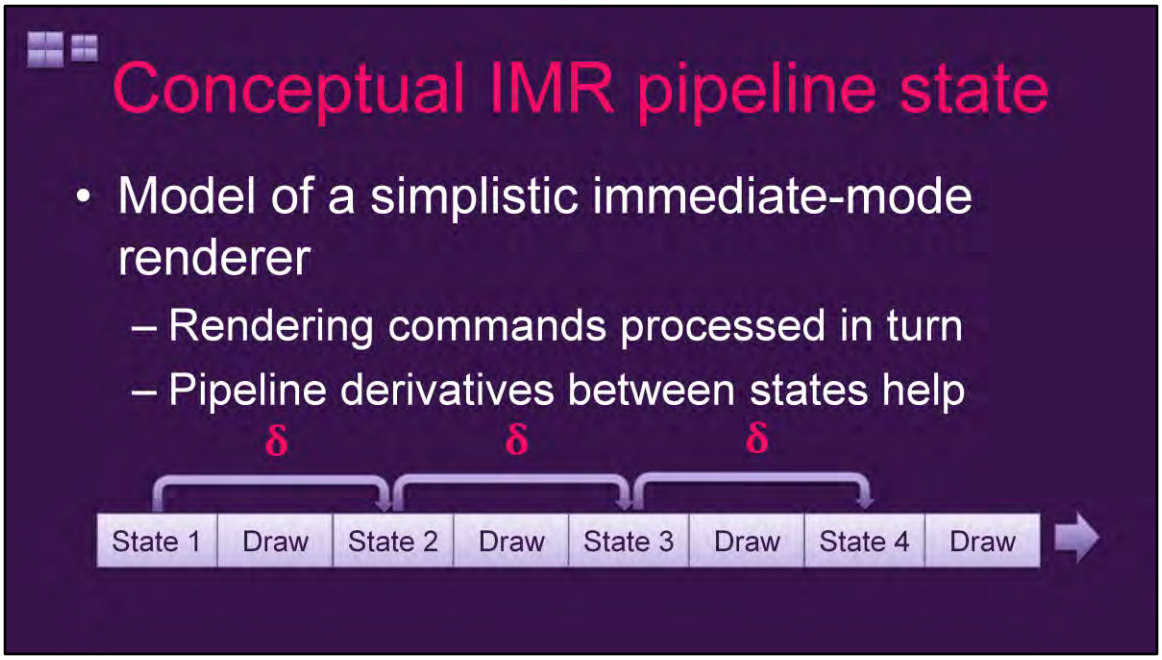
Pipeline state (apologies to Texans, it's the most "pipeline state" I could think of) may be quite different in tilers and immediate-mode renderers.

Pipeline state control how rendering occurs. Logically it controls configuration for rendering options; practically it typically involves quite heavyweight operations such as shader compilation.

State embodies rendering configuration, details of the images and buffers being operated on, etc.

Vulkan has two mechanisms for improving the use of pipeline state – derivatives and pipeline caches.

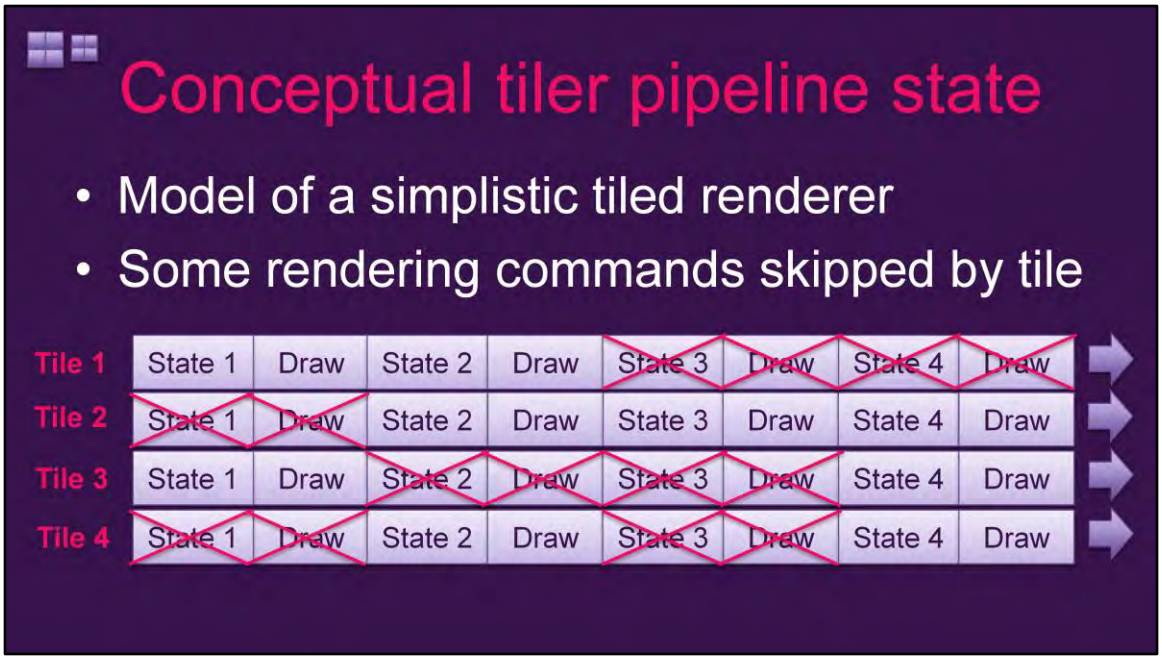
Why do we have two?



Immediate-mode renderers logically traverse the command buffer sequentially.

The driver may need to track what has changed between each state in order to produce an efficient change on the hardware.

Pipeline derivatives can offer a good way to improve the effectiveness of the driver at deducing state changes.

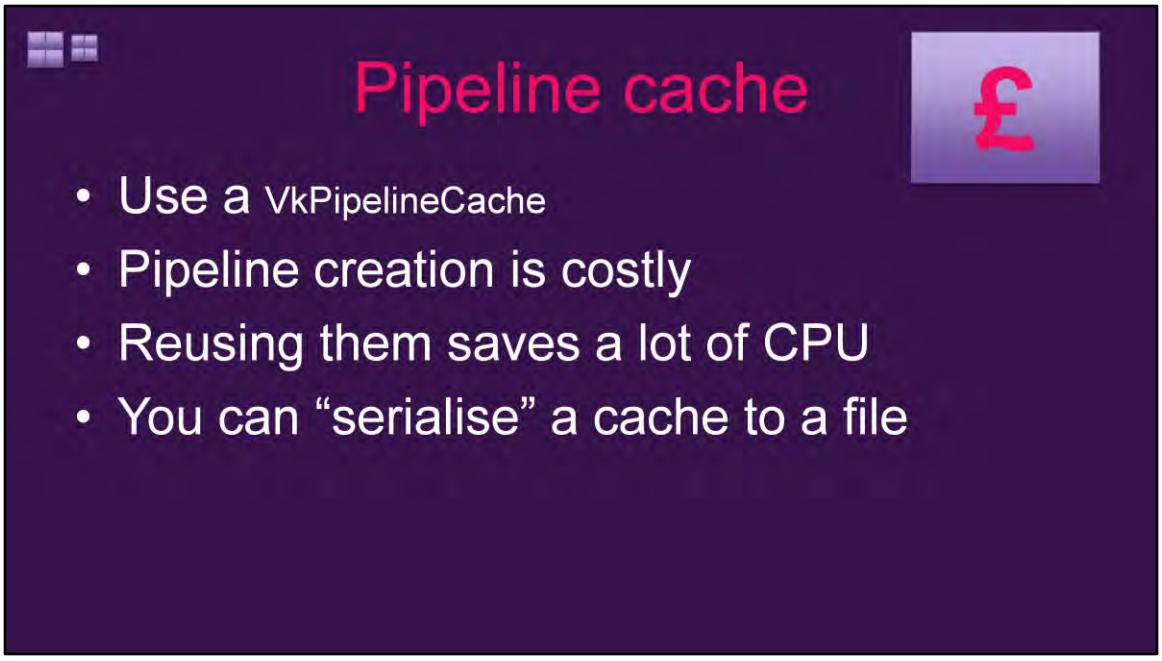


Tilers may not have this linear ordering through the command buffer because of the way that geometry which misses a tile can be skipped.

In this case, it is less likely to be useful to try to record differences between consecutive states in the rendering command scheme.

For this reason, the priority may be on using a pipeline cache instead (although both can be used).

Actual hardware doesn't lie on the extremes of these simplistic models – they are described just to show why different approaches aren't always useful.
Generally, both pipeline derivatives and the pipeline cache are valuable – just don't expect the same benefits on each type of hardware.
Again, the details of the driver behaviour are highly architecture-specific, so check the recommendations of each vendor.
The down-side of the explicit control provided by the Vulkan API is that the application author can no longer rely on the driver to make vendor-specific optimisations; the advantage is that there is less reliance on heuristics to determine the application behaviour.



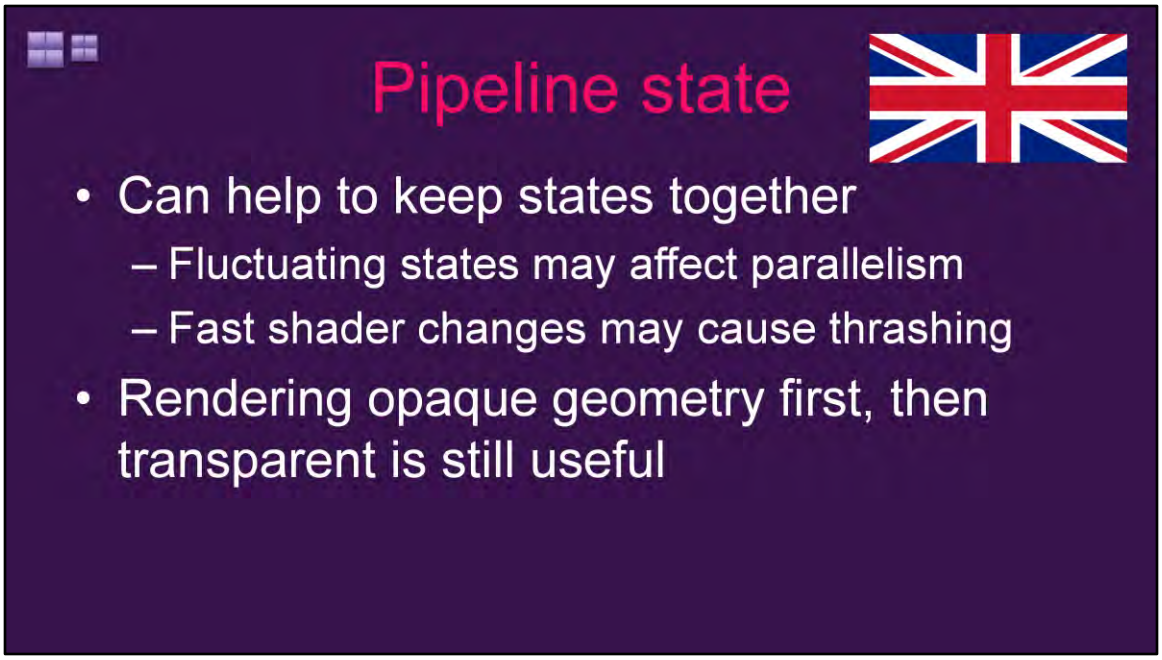
A pipeline cache, however, is still very valuable for recording state during command buffer generation, and between command buffers.

The importance of the pipeline cache is my theory for why people in the Vulkan working group keep saying ”£define”.

Using one is a substantial driver optimisation on some architectures, to keep pipeline creation costs under control.

You can also ”serialise” the pipeline cache to store the computed pipeline states in a way that can be retrieved the next time your application runs.

The stored cache can be reused, at least until the driver updates incompatibly.



Changing configuration options and the executing program at least affect the available parallelism on the GPU and can carry a measurable overhead on some architectures.

Where possible, it is useful to render primitives in an order which minimises the number of state changes – but see your hardware vendor's advice for details.

It is still valuable to sort opaque geometry before transparent, which helps some achitectures.

Summary

In summary... (this is a Brit's idea of summery)

Summary

- Wait in the right places
  - Don't queue unnecessarily

To keep things moving, do your waiting in the right place.

Summary

- Wait in the right places
- RAM: a necessary evil
  - Don't touch it more than you have to

Don't go near the RAM more than you can possibly avoid. Ram is baaad.

Summary

- Wait in the right places
- RAM: a necessary evil
- Use cache and threads
  - (properly)

Make proper use of threads.

Summary

- Wait in the right places
- RAM: a necessary evil
- Use cache and threads
- The driver won't do it for you

With APIs such as Vulkan, you can't rely on the driver hiding the hardware architecture from you in the way that GLES and similar APIs did – so you need to work with it.
Bear in mind that just because an optimisation doesn't help on one type of hardware, that doesn't mean it won't help on other hardware – as much as possible, it's worth testing on all the hardware you can.

Free Vulkan T-shirts
after the course!

Thank you

- https://community.arm.com/moving-mobile-graphics
- www.khronos.org/vulkan
- Khronos graphics BoF (Vulkan and GLES)
  - Wednesday 2:30-6pm Hilton Ballroom A
- a.garrard@samsung.com

Notes from this course are available on the course website.
Advice from hardware and software vendors is linked from the Khronos page on Vulkan (or go to your preferred partners directly!)
Khronos Birds of a Feather on graphics is in the Hilton, Ballroom A, on Wednesday.
There's a status update, but a Vulkan panel at 4pm and a Vulkan Tools panel at 5. Plus a party, after.
There are some free Vulkan T-shirts available after we stop talking.
(At least you all didn't dream about that guy with the cheese.)