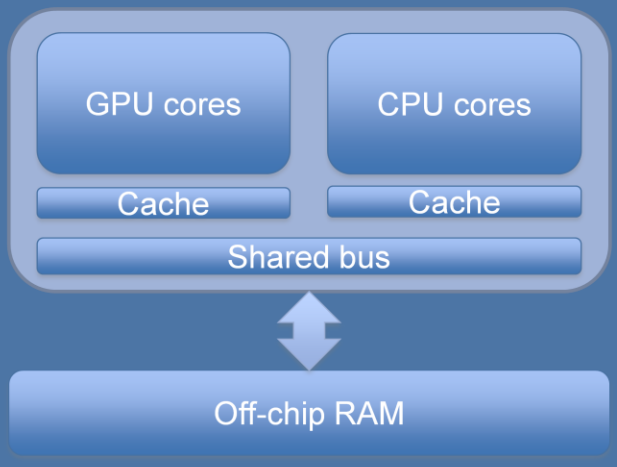
Hi everyone. My name's Andrew Garrard, and I work for Samsung in the UK.
Part of what my team do is help work out why applications may not be getting the best performance from mobile hardware.
Developers often complain that drivers have unpredictable slow-downs. Usually they don't: they have fast paths which can't always be applied. Slow and consistent is easy.
This section of the course is about techniques that software vendors can use to make the most of the hardware optimizations covered in the previous section.

Coding for mobile GPUs

- Limited bandwidth
- Hardware is clever
- Optimizations break
- Mobile design
- Next gen APIs



Andy Gruber has given us a description of some of the ways that implementers of tiled graphics processing units manage the limited bandwidth available in mobile and other embedded systems.

Mobile systems have a low-power, space-limited bus connecting the combined CPU and GPU chip to external memory, so the type of high-bandwidth connection used by desktop GPUs is not available.

Mobile GPUs use a range of approaches to limit the need for off-chip bandwidth.

These work well with most types of rendering, but the wrong inputs can break the optimizations and cause performance to drop.

In this section of the course, we'll discuss the types of content that can cause problems on some architectures, and how developers can tune their application for performance.

We'll also discuss designing software for mobile devices, and the exciting next-generation graphics APIs which are appearing.

A software view: outline

- Vertex/fragment interleaving
- Minimizing the cost of binning
- Working with memory
- Designing for mobile
- Next generation APIs

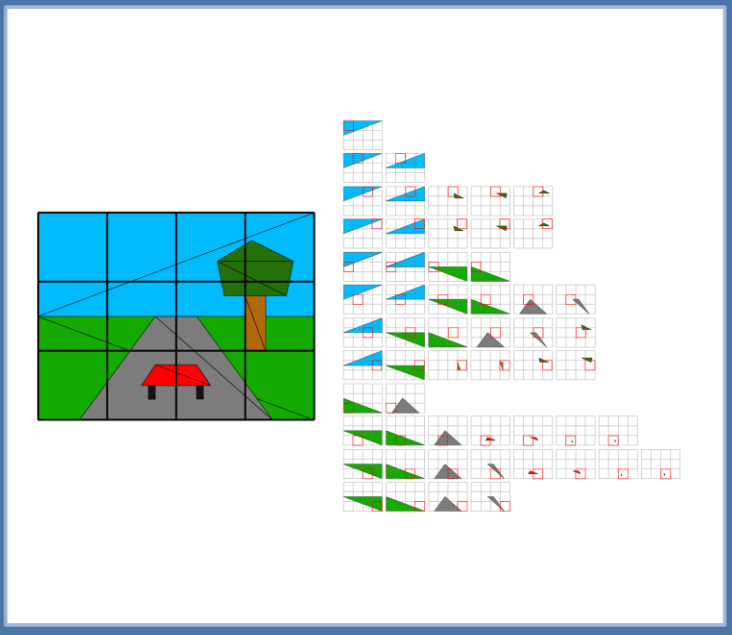
We'll start by talking a bit about how it's possible to trip up the two-stage rendering approach used by tilers.

Binning

Tiled renders first process the scene as geometry is submitted, applying vertex shaders to determine where the geometry appears (without occlusion)

The frame buffer is split into rectangular tiles

Triangles are recorded according to which tiles they intersect (details vary)



As Andy Gruber has described, tiled rendering is a two-stage process. The frame buffer area is divided into tiles.

As the application submits geometry for rendering, the primitives are transformed by the vertex shader (so that we know where they appear) and recorded according to the tiles that contain them.

The binning task is about classifying primitives into the tiles that they intersect.

Each vendor – and often different GPU generations by the same vendor – has its own way to do this.

Here's a simple scene broken down into 4x3 tiles, with a list on the right of each triangle that falls within the tile shown in red.

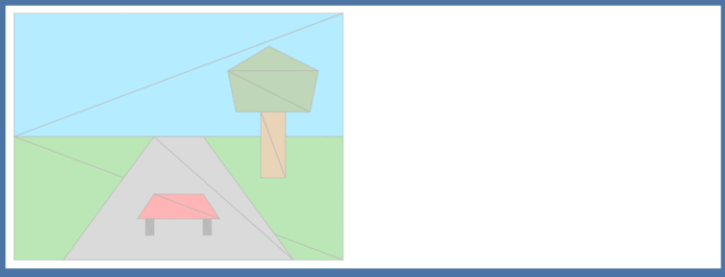
Some vendors, like Qualcomm, record references into the triangle list; others record different triangle lists for each tile.

You can see here that triangles – especially large ones – might touch multiple bins.

Rasterizing

Each tile is drawn in turn by looking at the triangles that the binning phase said intersected that tile

Local frame buffer access is fast, and the final image is written only once



Then each tile is processed (conceptually) in turn, looking at the triangles that are listed as intersecting the tile.

The tile is rendered to fast on-chip memory, so overdraw and access to high sample counts, Z buffer and stencil are cheap.

When the tile is rendered, it is written to the main frame buffer storage, off-chip.

(High resolution images with multiple rendering passes mean that storing the whole frame buffer on the GPU is not practical on current systems, although some older one did it.)

Then the triangles in the next tile are processed, and we continue until the image is complete.

Spilling

- On any tiler, reading the frame buffer causes the entire frame buffer to flush
- Flushing and continuing requires all metadata to flush
  - All samples, Z, stencil…
- Do everything you can to avoid this!

* Andy explained that accessing the frame buffer requires everything rendered up to that point to be processed – which means writing the frame buffer out to memory.

* If rendering then continues, this means preserving all the state needed for rendering to continue – state that can otherwise exist only on-chip.

* Examples are the full-resolution (not downsampled for antialiasing) frame buffer and the depth and stencil buffers.

These data can be much bigger than the frame buffer itself, burning bandwidth and dropping performance.

* So it is always better to do the rendering of a frame buffer in one pass. If you have reflections and shadow maps, render them first, don't swap back and forth.

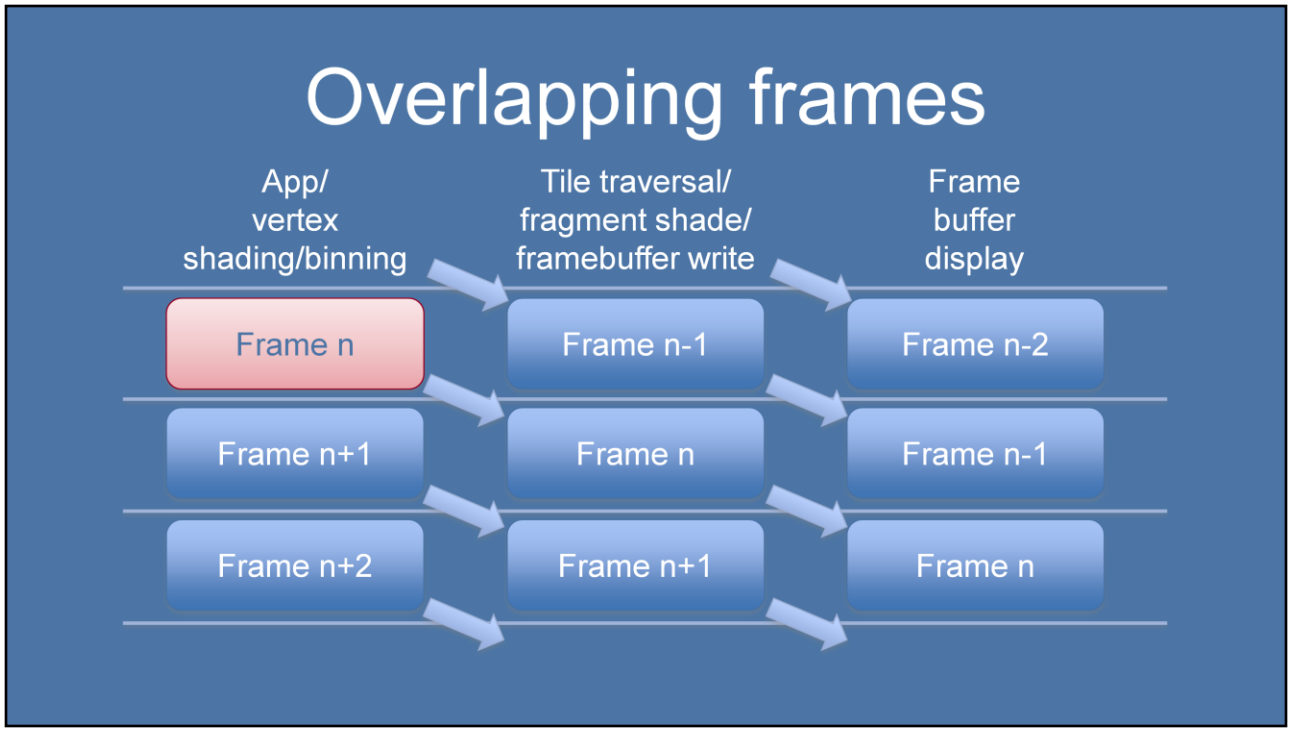
If you're doing an occlusion query to decide whether to do additional rendering, bear in mind the cost of making this decision.

Double buffering

- The binning stage must be complete before rasterization can begin
  - Otherwise you don't know whether there's more to be put in a tile
- *But* binning of frame n can happen in parallel with rasterizing frame n-1

* Although the binning pass must happen before the rasterizing of tiles can begin – * because otherwise you don't know if you've associated all the geometry with the tile – the binning for one frame is independent of the rasterizing of the previous frame.

* Some architectures do the two stages (which have complementary bandwidth and compute loads) in parallel, which can improve throughput.

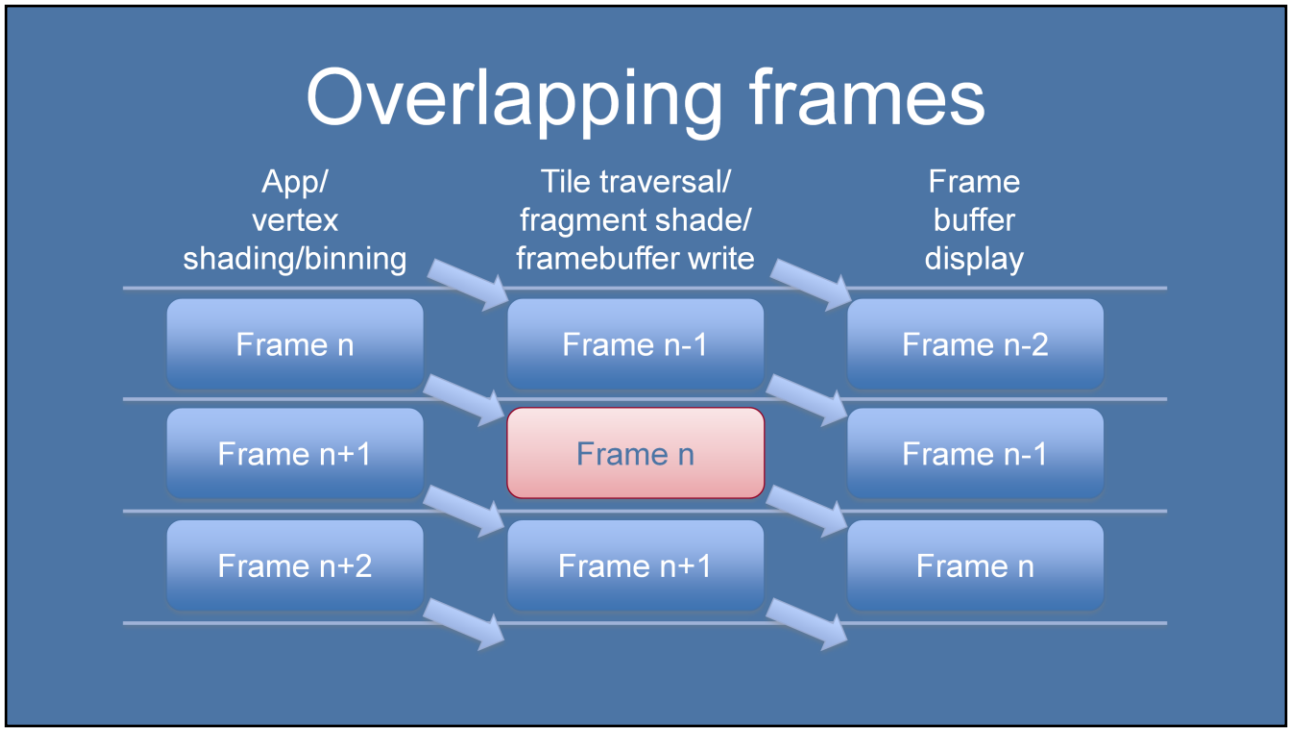


When the rendering pipeline is running efficiently, the application will be submitting geometry, which will be vertex shaded and classified into bins.

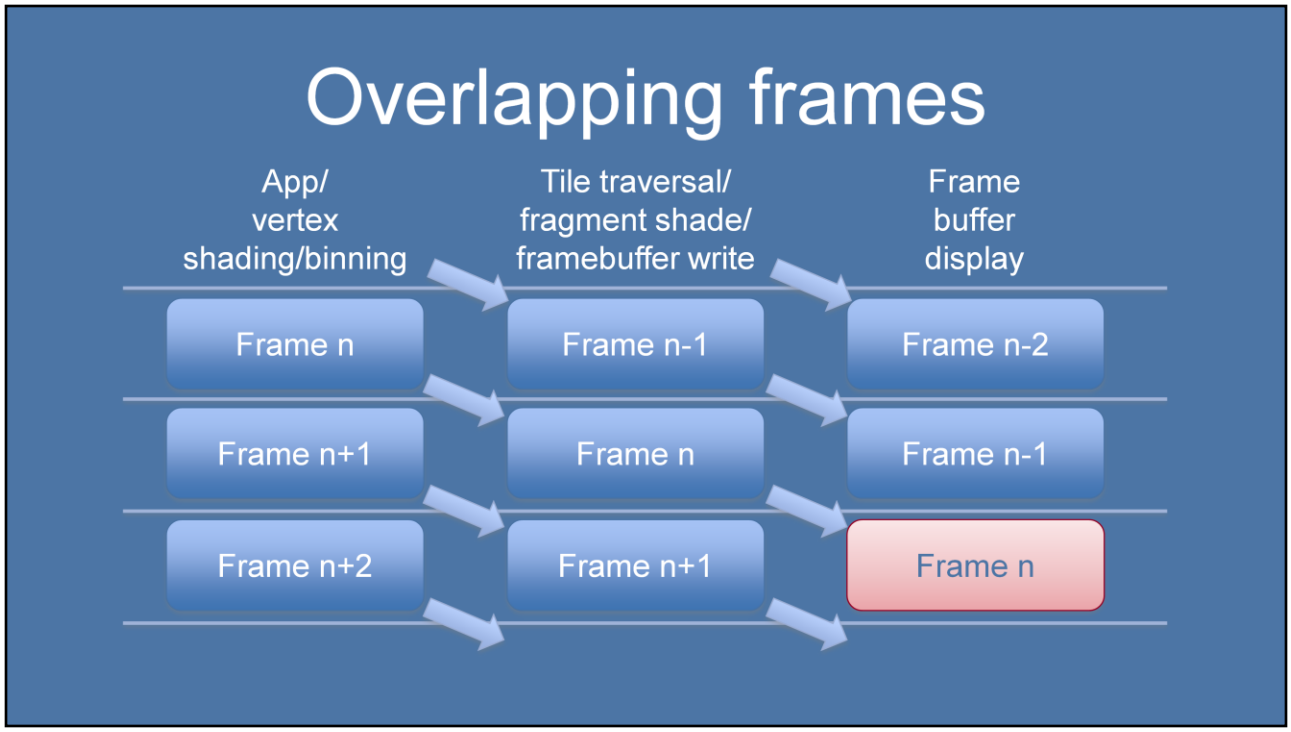
Simultaneously, the GPU may be iterating through the tiles of the previous frame, extracting the triangles, and fragment shading them into an on-chip tile buffer (and eventually resolving them out to memory).

(Note that nothing says only one tile can be processed at once!)

While the rendering is happening, the frame before that is being displayed.



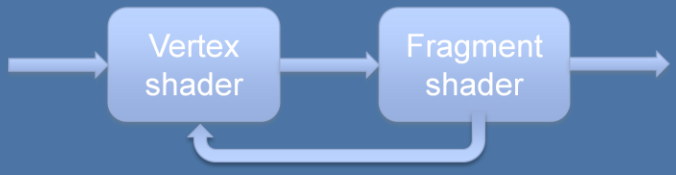
At the end of each frame, the frame buffer is pointed at the most recent rendering result, the rendering pass collects a new set of binned geometry to work on, and the application moves on to the next workload.



With this approach we end up implicitly triple-buffering – increasing throughput, but with multi-frame latency.

Breaking double buffering

- You can't interleave if:
  - Binning depends on the previous frame
  - You change something after you submit it, before it's used in rendering
    - Well, you sort of can…



* This scheme fails if binning depends on the rasterizing of the previous frame.

* This problem is rare, but if you're accessing the render target from a vertex buffer, it can bite you.

* For example, if you're doing a fragment shading pass to a frame buffer object and using the result to generate displacements.

* More commonly, writing to a texture from the CPU, render with it, then update the texture again, then render the next frame…

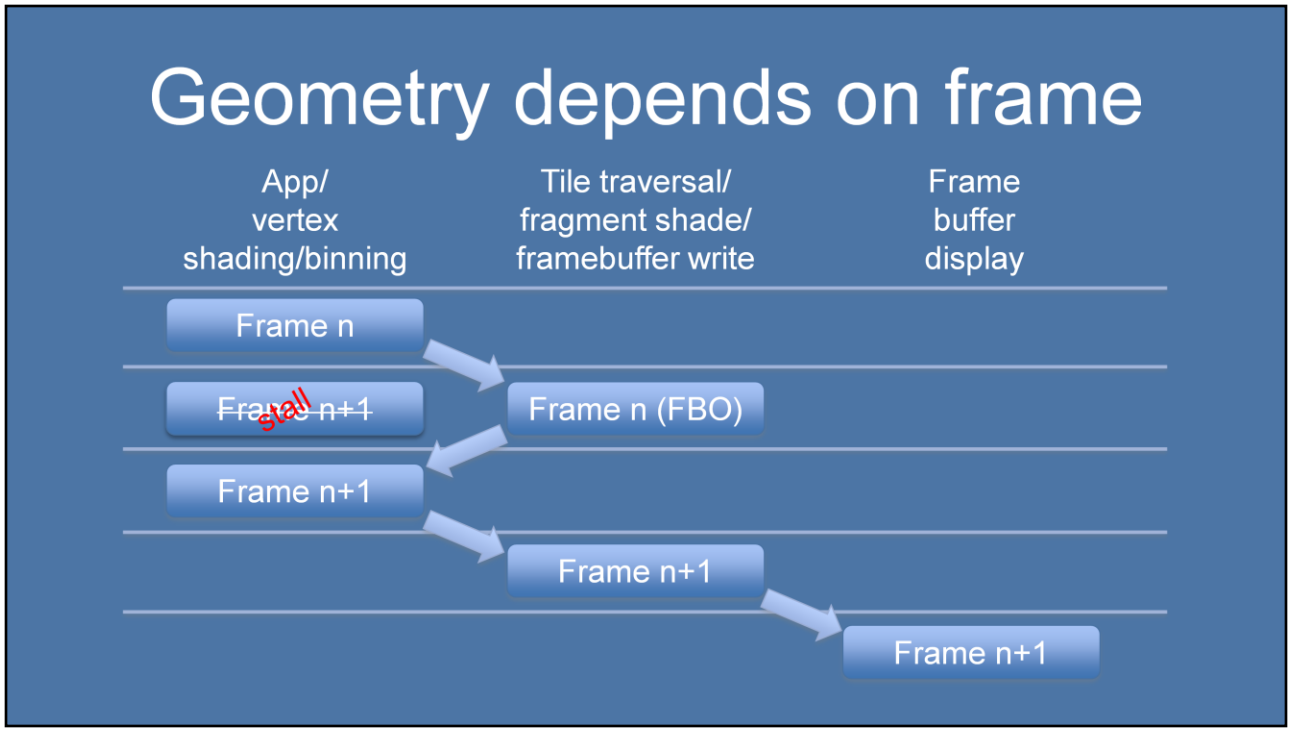
The fragment shader then won't have run before the update. The driver has to make this "work" somehow.

* The usual solution is to make a copy of the texture (before allowing the application to continue).

This must be a copy rather than just double-buffering the texture because it is hard to say what the user has changed.

Double-buffering the textures yourself (alternating between two textures) can avoid this copy.

This isn't a problem on every architecture, but it is on some!



Here we've got frame n+1 being submitted by the application, but something in the binning pass for it – such as the vertex shader – relies on the result of rendering frame n.

By "frame", here, we just mean a target of fragment shading – not necessarily for display – which could be an FBO.

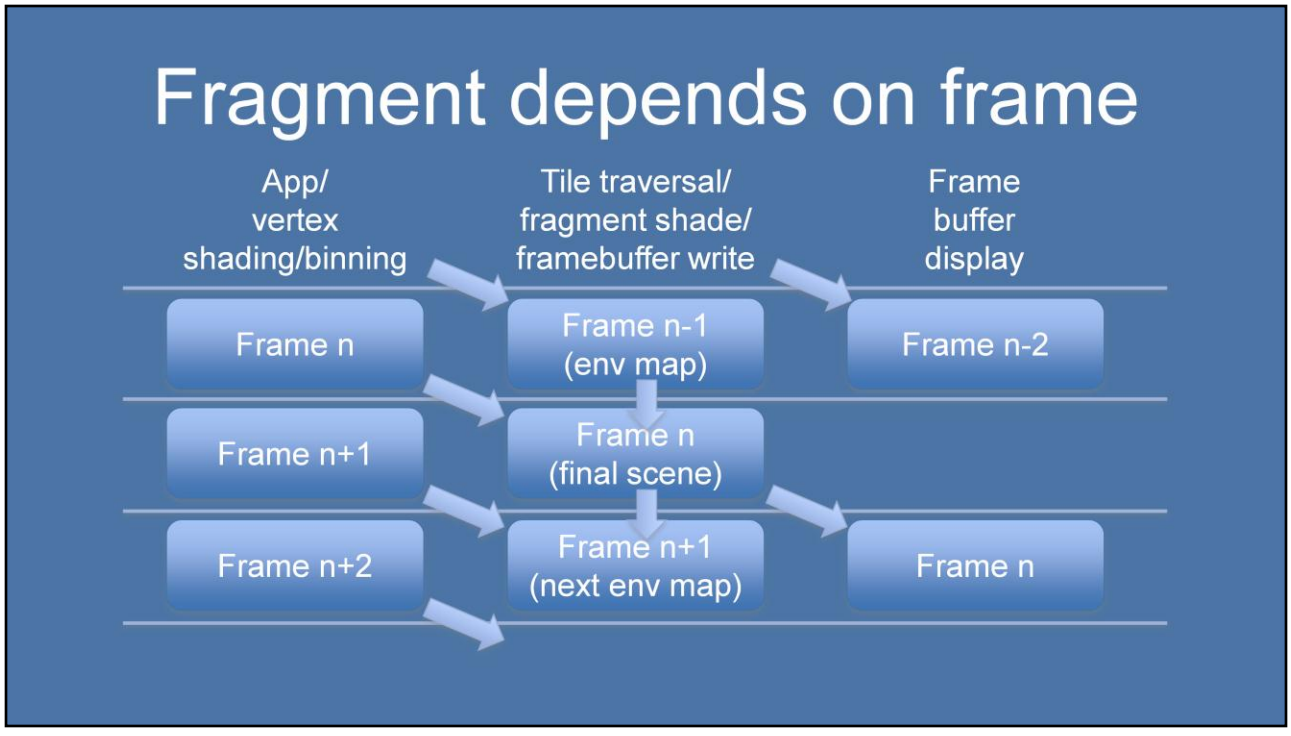
This means we can't pipeline the frames – frame n+1 submission has to wait until frame n has finished rendering, and we get a "bubble" in the pipeline.

Common cases are fine

- Note that using the output of a fragment shader in frame n in the fragment shader of frame n+1 is fine
  - Rendering has completed!

* These problems are only the case if something happening at binning time depends on the result of fragment shading.

* Using the result of fragment shading of one frame in the *fragment* shading of the next frame is fine.



Again, “frames” here refer to the output of fragment shading, not necessarily something for final display.

Here the fragment shading component of each frame depends on the result of fragment shading the frame before.

This might happen if you’re rendering a shadow map and reflection maps, and then using those results in a final render, or if you’re performing post-processing.

Because the vertex shading is still independent of the rasterized results, there’s no stall – the pipeline stays full.

A software view: outline

- Vertex/fragment interleaving
- Minimizing the cost of binning
- Working with memory
- Designing for mobile
- Next generation APIs

Tiling is a really effective way to reduce bandwidth requirements, but the various approaches have limitations that can reduce performance.

We'll talk a little about how the different implementations can be tripped up by the wrong input, and how to avoid this.

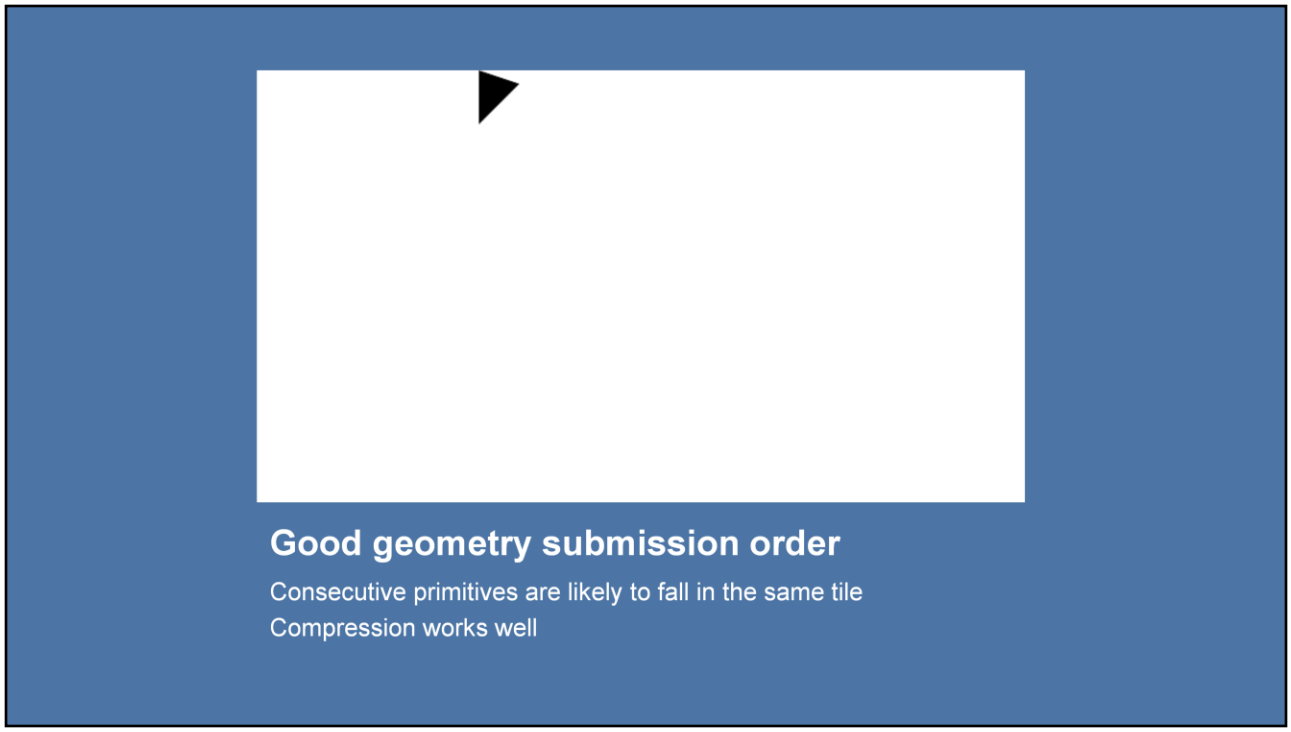
Locality of reference

- Some solutions have a single geometry list that is traversed repeatedly
- For efficiency, geometry that is not in a tile is skipped

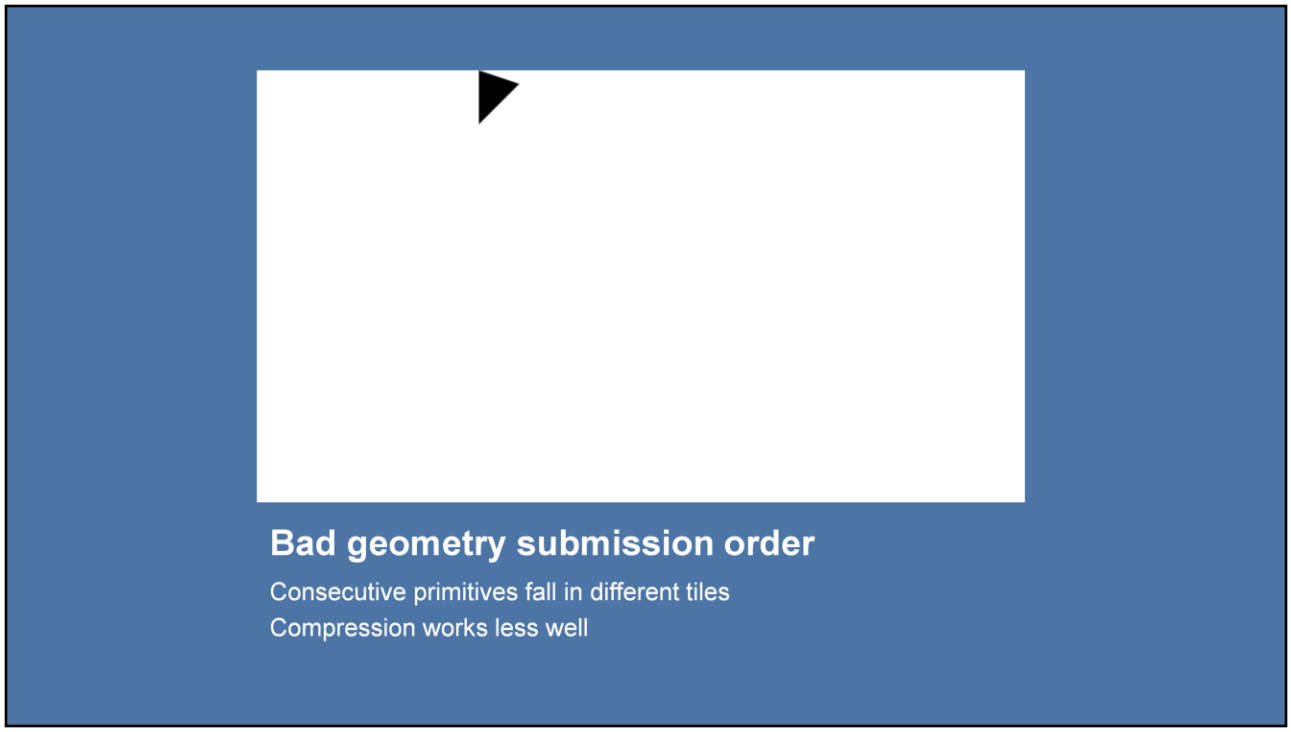
* As Andy has described, some solutions store all the geometry once, and have a mechanism for flagging what geometry belongs in each bin.

* Compression is used to allow sequences of geometry that are not present in a tile to be skipped (so not every primitive is processed for every tile, which would be slow).

The efficiency of this compression depends on geometry that his submitted consecutively having good locality of reference.



With good locality of reference, processing of a tile can skip large sequences of consecutive geometry.



With bad locality of reference, processing of a tile results in touching a lot of the geometry list.

Why would you do this? A geometry shader can generate replicated geometry in this kind of pattern.

More commonly, a particle system could result in primitives appearing in arbitrary order.

It may be worth attempting to keep particles partly sorted to limit this.

Repeated geometry

- Tilers process geometry repeatedly
- Crossing tile boundaries is bad
- Think about 2D grid alignment
  - Powers of two! (mostly)
- Try to keep geometry count limited

* Tiled renderers must process all the geometry associated with each tile on the screen.

* This inherently means that some of the geometry gets processed more than once, where it crosses more than one tile.

If the hardware binning scheme has one master geometry list, each tile requires walking through the whole thing.

If the hardware stores separate lists for each tile, binning can be slower and geometry takes more space.

If there's a mixture, big geometry tends to appear everywhere.

* If you have control over where to place your geometry – as in 2D games –* think about not crossing large powers of two boundaries (but not all architectures have square tiles).

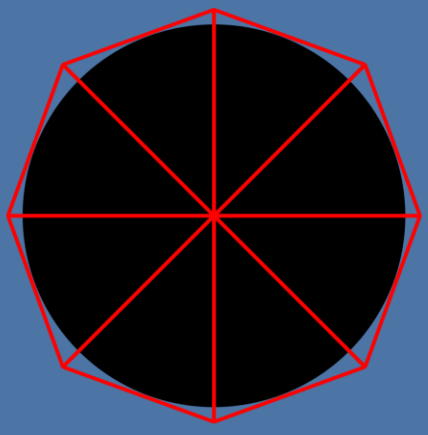
(Different GPUs have different approaches to binning; small tiles may need geometry to be traversed (or replicated) more, but mean that triangles are considered for less of the screen).

Very detailed but small geometry can cause uneven performance when it falls on larger tile boundaries – and watch out for hierarchical tiling schemes that might cause small geometry to be considered for the whole frame buffer. (Suitable use of level-of-detail approaches can reduce this problem.)

* The most obvious way to reduce the overhead of repeated geometry traversals is not to have so much geometry.

Limiting geometry

- Fragment discard
  - But slows shading
- Transparency
  - But order-dependence
- Polygon silhouettes



How do we cut down on the geometry?

Easier said than done.

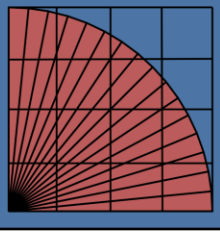
* You can draw the outline of the shape and use fragment discard, * but this typically bypasses the expected fragment pipeline operations, so it may be, at best, no faster than shading the whole primitive (and may be worse). It also won't play nicely with multisampling.

* You can use alpha blending and transparency – * but this approach will fill the Z buffer, so you need to ensure that the approximate Z buffer is sufficient. It may also break deferred shading.

* The additional cost of geometry may make some obscure techniques more useful: think whether you can move the primitives to the silhouette of the shape – e.g. drawing a sphere as a circle, not a polyhedron.

Beware diagonals

- "In a tile" can be approximate
- Big diagonals get processed everywhere
- Consider axis-aligned splitting
  - But beware adding more geometry



8	6	3	0
11	9	6	3
14	12	9	6
16	14	11	8

Binning has to be *fast* – it can be tied to the speed of geometry submission.

(Mobile GPUs can't throw a lot of silicon area at this kind of specialist problem.)

* Therefore not all GPUs do a full conservative rasterization to determine what tiles a triangle intersects: it's common just to use the coordinate limits of the transformed triangle.

* Triangles with a large diagonal component can be listed for processing a lot more tiles than they actually intersect.

The fan of 16 triangles in this diagram shows more triangles are being listed against each tile than actually touch them, because of this bounding box approach.

E.g. the bottom right tile only has four triangles in it, but processes 8.

* If you're tessellating a large shape, think about whether you can axis-align the tessellation – in this example, forming the curve with horizontal triangle strips would probably work better.

* The trick is to do this without increasing the geometry count enough to make things worse. Tessellating down to the tile level is unlikely to help.

With small primitives, this isn't so likely to be a huge problem – they rarely cover many tiles.

The good news is that increasing scene quality tends to mean primitives are getting smaller relative to the screen area. The bad news is that, as we've seen, increasing

geometry can be painful for a tiler.

A software view: outline

- Vertex/fragment interleaving
- Minimizing the cost of binning
- Working with memory
- Designing for mobile
- Next generation APIs

As we've seen, the performance of mobile GPUs is heavily determined by off-chip bandwidth.

This is likely to get worse: Moore's Law adds calculation performance, but bandwidth is limited in space and the requirements of driving external signals.

At some point, you do need to access the memory; let's talk a bit about how to do that.

Texture formats

- Use compression when you can
  - Fit the format to the content
  - Fit the content to the format!
- Don't forget uncompressed formats
- Maybe roll your own scheme

* Andy has already spoken about the merits of texture compression.

* There are a range of different compressed formats, some of which have configuration options; newer APIs are standardizing support.

Each approach has a different size and quality trade-off: see what best fits your source image.

* Think about aligning your texture detail for texel block boundaries (and don't forget ASTC's range of texel blocks sizes).

* Don't forget native formats that aren't compressed if they're a suitable fit.

Some frameworks may convert to a standard, large format – don't assume everything is as small as you'd hope.

* The texture compression schemes are lossy and designed for photographic-style textures; you know your content, so you may be able to do a better job with shader code, especially if you're not filtering.

Sizing and filtering

- Do you *need* a trilinear-filtered mipmap?
- Consider scaling offline
  - But remember external displays!
- Is this the best scaling anyway?
- Filtering in the shader

* If you're doing full 3D rendering, you're generally stuck with conventional filtering.

If you're doing a 2D app, make sure your API isn't doing trilinear filtering just to get the texture to the size you want.

* You could scale in advance, meaning you can get away with low-cost point sampling.

Upscaling does allow smaller textures and, therefore, less bandwidth – but think about prefiltering for the target resolution even then.

* If you're scaling for the device, remember you can attach an external screen – have a fallback path.

* Conventional filtering for 3D may not look best – * better quality may come from shader techniques such as reconstructing edges from signed distance fields.

Frame buffer bandwidth

- Simple content compresses better
  - Don't give artists free reign!
  - Think about tile-aligned detail
    - In textures *and* output
- Work with transaction elimination
  - Don't change more than you have to

We've been told that vendors have some approaches for compressing the frame buffer (losslessly).

* Like most compression schemes, these tend to work best with predictable content. A photorealistic image might take more bandwidth than plain line art, because the line art will compress better.

* Artists may like to include pretty flourishes in user interfaces, but there is a cost to this.

* If there is going to be areas of high detail in the image, think about power of two tiles – * localized detail means that other tiles can still be compressed effectively, and careful positioning can increase the number of tiles with good compression.

* Hardware vendors also use transaction elimination – only updating a tile if it's actually changed (although bear in mind the triple-buffering we discussed earlier).

This isn't the same thing as overdrawing previous content – you still don't want the old framebuffer content to be loaded back onto the GPU.

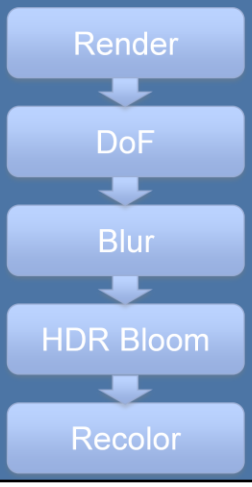
How can you help this? In 3D, short of areas of solid color, not much. * In 2D, think about aligning animation to tiles.

Also remember that rounding errors can introduce changes – a barely-visible subpixel position change can require the tile to be written out.

(That animated background character was causing a lot of unnecessary frame buffer work, especially if it crossed into another tile by a couple of pixels, and especially if it was only "animated" because of rounding.)

Frame buffer processing

- Reading outside a tile requires a full flush
  - Increasingly useful
- Combine passes if you can
  - Kernel fusion, PLS
- Separable filters
- Incorporate into fragment shading



Sometimes you want to do a post-processing pass, and you can't do everything within the tiling scheme.

(For example, blooming on HDR content or depth of field effects.)

* Because you have to read outside the current tile to do this, the whole frame buffer will be written, and there's currently no common implementation for making this fast on a tiler.

* However, filtering operations are common enough, especially in computer vision, that we can expect technology to appear in the future. OpenVX is based on the assumption that hardware can do this.

If you need a downsampled version of the frame buffer, which is common for blurring effects, check whether you need to start with the full resolution version.

For simple scenes, you can render the whole scene again at a smaller scale faster than resizing a larger image – especially if you can use level of detail to simplify the smaller version.

* Where you do need to perform post-processing passes (or when you're using the GPU to perform filtering operations on camera content, for example), try to minimize the number of accesses to memory.

* This means combining passes – such as by generating shaders that load textures from multiple sources and blend internally.

If you need more complex reuse across different primitives, consider pixel local storage (see the next talk).

* If you must talk to main memory, think about the texture cache size. Sometimes it's worth

using separable filters – but, on tiled content, sometimes it won't be.

* Finally, some post-processing passes are only for simple operations such as recoloring.

This can be really costly compared with incorporating the same set of operations into fragment shaders, even if this means the same instructions are being run redundantly for overdrawn content.

A software view: outline

- Vertex/fragment interleaving
- Minimizing the cost of binning
- Working with memory
- Designing for mobile
- Next generation APIs

So far we've been talking about techniques for working with tiled architectures – and we should be clear, not every mobile GPU is tiled (or at least, *always* tiled).

There are other considerations in working with mobile hardware, so let's talk about a few of those.

Mobile limitations

- Small caches
- Vertex ordering
- Portability
- Feature set
- Storage and content distribution

Not everything is about off-chip bandwidth.

* Even allowing for Moore's Law, mobile GPUs have a limited die area. Caches are smaller than desktop GPUs, which makes the bandwidth problem worse.

This is another reason to use texture compression, but also means that managing texture resolution and locality of reference can be important.

* Desktop systems may benefit from cached vertex shader operations when drawing with indexed vertices.

In mobile, this cache is very small, and incoherent access of the vertex buffer can be costly – it can sometimes be better to process the vertices without indexing.

* There are a lot of very different mobile GPU implementations – you can expect to see larger behavioral differences than you would find between typical desktop systems.

Although a basic GLES application is likely to run on everyone's GPU, the performance on some workloads may vary considerably. Each vendor has made different design trade-offs.

This concept runs as deeply as the shader implementations: some ALUs are scalar, some SIMT; some have vector units, some work on one component at a time.

This doesn't make much difference for most graphics work, but if you start running compute-heavy operations, it can do.

* The degree of floating point support can also differ between systems, and the latest graphics extensions are rolling out – and there are a lot of legacy devices running legacy APIs out there.

The latest systems come close to the level of functionality found in desktop GPUs

(tessellation and geometry shaders, for example) – but these aren't universal yet.

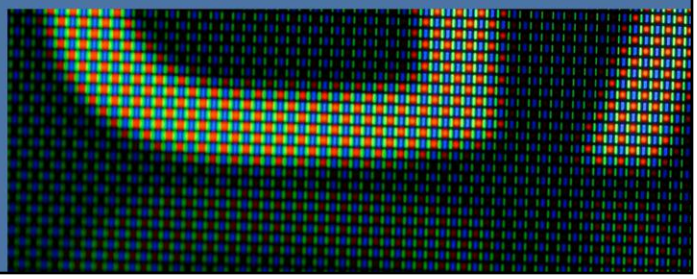
* One concern for the content creator is the size of the application. It's become common to stream content, but mobile games can still be extremely large.

On a PC, storage space is not a major concern – hard drives are cheap. In mobile, not every device has removable storage, and not everyone has an unlimited data contract.

Samsung hasn't made a phone with a blu-ray drive – yet.

Mobile displays

- High pixel density has benefits
- Viewing conditions hide details
- Wide range of resolutions/aspect ratios
- OLEDs
- Consider RGBW
- Work *with* PenTile



* We have some advantages when rendering to a mobile display – most notably that the pixels are tiny.

This can create better immersion than desktop graphics on some content despite the small screen size.

The reverse argument applies to VR – you get immersion from the large field of view, but large pixels are visible!

Not seeing the pixels removes the effect of looking at a computer screen – and also means you can fudge fine detail, or do less antialiasing than you'd need on desktop.

* Mobiles are also rarely used under ideal viewing conditions, unlike a carefully-positioned TV or monitor.

When there's bright sun shining on a small screen that's covered in fingerprint grease, not every detail will be visible. This helps us to cut corners.

* On the down side, where most desktop systems have standardized on a few aspect ratios and have windowing systems, mobile applications need to be able to scale.

Pixel density varies – especially when connecting to an external display; aspect ratio varies; distance to the display varies. Apps need to compensate.

Remember that an emissive display can require a significant power drain on its own.

* On OLEDs in particular, darker colors are more energy efficient – it is worth considering that when making artistic decisions.

* Some displays support RGBW; think about limiting saturation to support these.

* PenTile RGBG displays, * which are common on OLEDs, have higher resolution in the green channel than the red or blue.

This means that shaders and textures can save processing performance on values that can’t be seen.

Mobile advantages

- The nature of casual gaming
- Means of interaction
- Connectivity and cloud rendering

* Mobile devices are often used for casual games – while there is a fair share of moderately photorealistic content out there, many people dip in and out of their content.

This makes simple content and simple graphics more forgivable (and don't forget that spending minutes uploading textures can cut into a lot of play time).

No hand-held device is going to be as immersive as a large television for a first-person shooter, and there's a lot to be said for the popularity of flappy bird.

* Most mobile devices have touch screens – from a rendering perspective, this means your hand must be blocking the screen. You can save some rendering work by allowing for that.

* And, of course, most mobile devices have an internet connection. If you can manage the latency (and have a fall-back for when you're in a tunnel), you can think about offloading the heavy lifting to a render farm.

A software view: outline

- Vertex/fragment interleaving
- Minimizing the cost of binning
- Working with memory
- Designing for mobile
- Next generation APIs

The vast majority of mobile graphics work to this point has used OpenGL ES.

In the last couple of years, people have been working on reducing some of the historical limitations of GLES by using new APIs like Metal, Vulkan, Mantle and DirectX 12. So, what can we gain?

How can we improve GLES?

- Not a lot (kind of) if GPU-limited, but…
  - A lot of CPUs per unit GPU
  - A lot of driver magic (GPUs are different!)
  - GLES drivers handle memory
- Explicit control isn't for all rendering tasks
  - Engines may be your back-door way in

OpenGL ES has served us well for a long time, and it's relatively easy to program in, and newer extensions offer increased performance.

What problems does it actually have?

* Well, so long as you're limited by the GPU performance, not much. There are exceptions, but the new APIs won't do much to make a fragment-bound application go faster, though they can cut driver overhead a lot - you may not be as GPU limited as you think.

* However, mobile devices typically don't devote as much die area as you might expect to the GPU, and there's a lot of CPU performance available.

Having a single-threaded rendering API can be a bottleneck.

* Because the hardware on different mobile GPUs is so different, a lot of driver magic happens to make rendering efficient.

Because application software is often unaware of what the driver has to do, this makes performance unpredictable.

The driver has to apply heuristics to decide how the rendering task should be mapped to the GPU, with limited information from the GL API.

* In GLES, the driver controls graphics memory, which makes it hard for the application to control memory reuse and content streaming.

* A word of warning: the explicit nature of the next-generation APIs means they do need more coding.

If you're optimizing performance, they can *save* you a lot of work compared with reverse-engineering driver behavior.

If you're coding a simple application, especially if you are a novice to graphics, they may not be the best introduction.

* However, game engines are adopting the next-generation APIs, giving you performance without the complication.

Don't be intimidated by these APIs – explicit does not necessarily mean complicated, and extra helper layers and libraries will appear to take care of some of the heavy lifting for you.

Nevertheless, OpenGL ES is by far the most widely-deployed mobile graphics API, and the most portable solution for mobile graphics; don't rule it out yet!

Reducing CPU overhead

- Spread the work onto multiple threads
  - Lots of spare cores on many devices
- Costly operations off the critical path
- Reusing work

What can we do with new APIs?

* The first advantage is moving the work to multiple threads.

* When we have eight cores available, spreading graphics work between them can give a big speed-up on scenes that are highly dynamic.

* Submitting geometry is inherently ordered, at least for a single frame buffer. But the hard bit is actually not the geometry submission: it's operations like converting state and geometry into hardware-specific forms, and compilation steps. These can all happen on independent, separate threads.

* We can also re-use some of the set-up work that has been done.

Increasing predictability

- Explicit state transitions
- Schedule costly operations when needed
- Compilation steps on request
- Application memory management
  - Reduced memory overhead

* Arguably the biggest advantage of the new APIs is that they are explicit: rather than applying heuristics, the driver will do what you tell it to.

You can control passes in a tile and manage bandwidth.

This means you need to listen to the driver when it tells you the steps it needs to perform, which does complicate simple applications. In learning to do this, developers may get better at using older APIs more efficiently!

* However, knowing exactly what's going on and when means that performance is much more predictable, and * heavyweight operations can be scheduled when performance is available.

* Applications also have more direct control over the memory used by the application – streaming is more predictable, * memory can be more easily aliased for different purposes during the rendering stages.

More next gen this SIGGRAPH

- "An Overview of Next Generation APIs"
  - Tuesday
- Khronos BoFs
  - Wednesday

I'm not going to say more about the next generation APIs here, other than to say that they're an exciting opportunity for improving performance.

There are more detailed presentations happening at this SIGGRAPH where you can learn more: * there is a course on Tuesday, and * Khronos, originator of OpenGL ES, will be discussing Vulkan more during the graphics birds-of-a-feather session on Wednesday.

Predictions

- Resolutions will keep rising (but slower)
- GPU performance will increase
- Caches will get bigger
- Software will get more demanding
- Don’t expect the RAM to keep up

What can we expect in the future?

* I thinks resolutions will keep growing – although the rate seems to be dropping off now that the pixels are barely visible.

(I’d note that HD content doesn’t scale properly to 2560x1440, though, so UHD screens have their place.)

There’s always a trade-off between resolution and performance, so don’t expect 8K screens just yet.

* GPUs are going to keep getting faster, and cleverer.

* Caches will grow, partly removing one of the awkward mobile limitations – but the increase in resolution will counter this.

* Software will place higher demands on the GPU, or “better quality rendering” as the software vendors would probably like to market it.

* What I don’t expect is for RAM performance to increase at the same rate as computation – physical demands on space and power are limiting with current technology.

Summary

- Keep the pipeline flowing
- Limit the cost of traversing geometry
- Simplificate – the RAM is lava!
- Design for the target
- Things are getting more explicit

What have we learnt?

* Tiled architectures have two stages, which may or may not be interleaved, and try to keep content on the chip. They're fastest when they can do this.

* Tilers touch geometry more than immediate-mode renderers – so increasing geometry hurts them more, and geometry that touches multiple bins can be painful.

* If you're rendering something simple, it'll probably render fast. Make sure the cost in battery life – or play time – is worth the pretty effect you want to add.

* Mobile devices aren't the same as desktops. You can cheat, and the ideal content is often different.

* And the next generation is coming – it's worth understanding how mobile GPUs work so you can be ready to make the most of them with the new APIs.

Thank you!

a.garrard@samsung.com
http://community.arm.com/moving-mobile-graphics-2015