The Revolution in Mobile Game Graphics

Sam Martin, Geomerics
Marius Bjørge, ARM

Hi!

I'm Sam Martin from Geomerics, these days a wholly owned subsidiary of ARM. I'll be talking with Marius Bjorge from ARM. Marius does a lot of graphics r&d out of ARM's Trondheim office.

The last half-dozen years have seen a revolution in mobile graphics. The pace of innovation has been (and continues to be) furious so today we take stock of where we are as well as looking at what's coming down the road.

It was announced at this year's GDC that Enlighten, a high-end real-time lighting solution typically seen in large budget titles, will be available as the lighting solution in Unity 5. We look at what this enables and what the standard for game graphics on mobile is today.

In the second half of the talk we will look to the future and discuss several recently released OpenGL ES extensions that enable very efficient advanced rendering on tile-based devices. This includes "pixel local storage", a rather unique extension that has cross-vendor support and will enable console rendering architectures on mobile GPUs.

Mobile graphics in a nutshell

- 2008 - now
  - Significant die area for mobile GPUs
  - Advanced mobile CPUs
    - ARM® NEON™ vector instructions
    - ARM Cortex®-A9, A15/A7 (32-bit), A57/A53 (64-bit)
    - Dual/Quad/Octa core and ARM big.LITTLE™ configurations
- Massive increase in mobile capability
- Market still catching up

A lot has happened between 2008 (roughly the time of first iPhone) and now.

Over this time period there been a huge rise in the amount of die area mobile processors devote to the GPU. For manufacturers, die area means money. The significant rise in die area therefore reflects the importance placed on mobile graphics. But it's also been a very quick rise and in many ways we are still catching up with opportunities this investment offers.

Over the same time period, CPUs have become more complex but also more specialised to mobile. Multi-core is arguably more important to mobile than desktop. Mobile devices must work within very tight thermal limits that put a cap on raw single threaded performance. Going wide over several CPUs is a key to unlocking more potential while remaining within thermal limits.

Mobile workloads are also diverse. For a long while A9's have powered much of the smartphone market. More recently we've seen pairs of cores emerge, such as the A15's (a "big" core) and the A7 (a "little" core). Devices now ship with a mix of both CPUs in the same chip. The big cores are powerful but hungry – they are designed for fast but short lived operations such as a webpage load. The little cores are somewhere between the A9 and A15 in terms of performance but are considerably more power efficient than the big core. For sustained workloads, such as the ones we have in graphics and gaming, the little cores are the most important. In short, you should run all your code on little where ever possible.

Much of this is still new to the market.

AAA mobile graphics?

- Talk of AAA on mobile
  - Will happen, but won't be the same as AAA on console
- Mobiles != desktops
  - Memory bandwidth differences
  - API differences
  - Incorrect assumptions of tile-based renderers
- A lot of potential in what exists today
  - Unity 5 with Enlighten
  - Physically-based rendering
- Many novel innovations on the horizon
  - Deferred rendering and programmable blending are both possible

There's a lot of talk about AAA graphics coming to mobile. Where there's a will there's a way, so I'm sure we'll see it happen, but if we are to use mobile hardware effectively we will need to change a few things first.

Mobiles are not just a smaller version of a desktop. The architectures are fundamentally different in a few important respects, notably:
- capability differences, particularly memory bandwidth
- api differences (e.g. OpenGL ES vs OpenGL/DirectX)
- majority of mobile uses tile-based GPUs

In order to get the best out of mobile we therefore need to approach it with these differences in mind. Bridging this gap is what drives a lot of our R&D, and we will return to this theme later when we discuss some of the extensions that are becoming available.

There is still considerable potential in what exists right now, so first we'll examine what we can do with Unity 5 and Enlighten on current devices. We'll cover some of the techniques used in our recent Transporter demo, including a few tricks from physically-based shading.

Agenda

- The "Now"
  - Real-time global illumination in Unity 5 with Enlighten
  - Physically-based shading on mobile
- The "Next"
  - Bandwidth-efficient graphics
  - Programmable blending
  - Pixel local deferred rendering

Enlighten in Unity 5

Enlighten

- Practical real-time global illumination
- Enlighten at run-time
  - Dynamically light your game
  - Unique, high quality lighting
  - PC/Mac/Linux/Consoles/Mobile
- Enlighten during development
  - Fast iteration of lighting design
  - High quality lightmap baking
- Real-time, baked and everything in between

Enlighten is a practical real time GI solution that runs on everything from mobile to next-gen consoles and desktops.

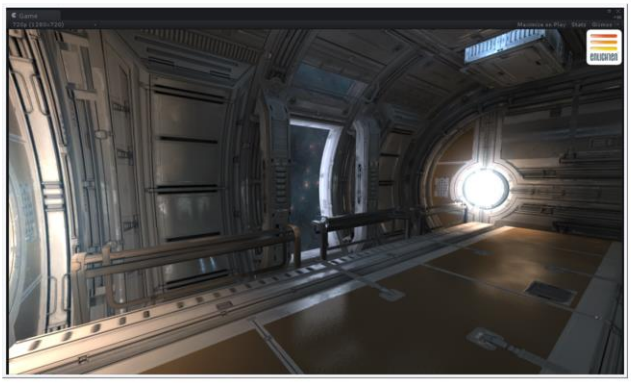
There are two main uses for real time GI:

- Add high quality dynamic lighting to you game/application
- Use Enlighten offline to iterate quickly on lighting design. Artists with better tools make better art, so even when lights are static real time GI helps improve lighting quality through improved iteration times.

Enlighten's core tech is a really fast radiosity solver, but the product is a suite of tools to cover all aspects of game lighting. It includes a static lightmap baker that produces output equivalent or better than other popular lightmap generators, with the added advantage that you can iterate on the results in real-time. It therefore covers the full spectrum of baked to dynamic light types and allows you to mix these as you wish.

Enlighten in Unity 5 – key features

- Real-time GI lighting
  - Dynamic lightmaps
  - Dynamic cubemaps
  - Dynamic light probes
- High quality lightmap baking
  - Non-blocking operation
- Real-time lighting workflow
  - Baked and/or dynamic lighting
- Cross-platform support
  - PC/Mac/Linux/Consoles/Mobile



Enlighten will be the default lighting solution in Unity 5, replacing the existing Beast workflow. It will still support baked lightmaps as before, but adds additional dynamic options.

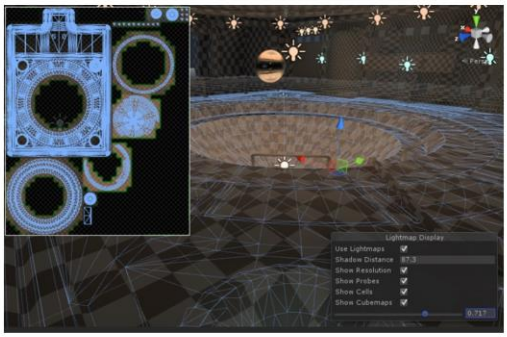
For want of a better term for it, a “normal” baked lightmap will contain both direct and indirect lighting as before. For dynamic content, Enlighten relies on the renderer to produce the direct portion (such as the effect of a spotlight or point light) and focuses on solving the indirect part. Enlighten generates three types of output dynamically:

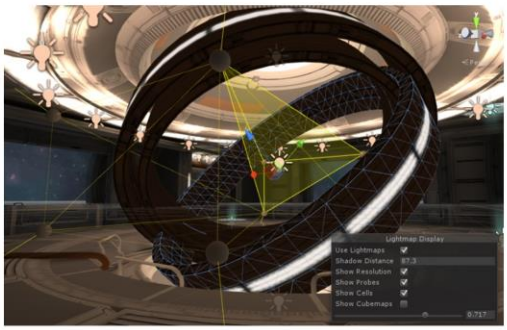
- Indirect-only lightmaps. These are much smaller than “normal” lightmaps , typicall y around 64x64 pixels.
- Light probes. These are for lighting dynamic objects and fit into the existing lightprobe workflow in Unity.
- Indirect-only cubemaps. These provide lighting information to reconstruct a compelling indirect specular effect, and can fit into a physically-based shading system.

Unity put a considerable amount of effort into updating the “normal” lightmap baking workflow as streamlined as possible. Enlighten can bake lightmaps more quickly and works incrementally, so updating an existing result from a small scene change happens very quickly. This, and a very carefully crafted integration of Enlighten into Unity’s job scheduling system, enabled baking to be run as a non-blocking background task. So in Unity 5 you just move lights around, and if they are baked the lighting updates a moment later. You are free to continue editing the scene and the lighting in the meanwhile, solving one of the main issues with lightmap baking blocking your development time.

Enlighten 101

- Hybrid CPU/GPU system
  - CPU handles all indirect lighting
  - GPU renders direct lighting (as usual)
- Offline generation of:
  - High quality baked lightmaps
  - Ambient occlusion
  - Interactive lightmap compositing
- Mature tech used in AAA games
- Transporter demo





(Screenshots in this talk taken from our GDC demo – Transporter)

At runtime we deliberately split the direct and indirect light. All Enlighten work (the indirect lighting) happens asynchronously on the CPU, not the GPU. As a result, Enlighten does not take any significant GPU resource and so does not affect your frame rate. GPUs are highly contented resources, so this approach allows us to add to the graphical output but without trading-off some other GPU-based effect.

Offline, we are free to run Enlighten real-time in the editor, but can also generate additional baked lighting data, including ambient occlusion for inclusion in the baked lightmaps. Enlighten lightmap baking is more modular than before. Internally, Enlighten bakes visibility for each light separately, and outputs separate lightmaps for direct, indirect and ambient occlusion. By keeping the individual lightmap components around we can quickly re-composite the final lightmap. This means that minor, but global changes such as increasing the overall level of bounce lighting or ambient occlusion contribution can be handled interactively. Changes to individual lights only cause minimal re-bakes.

In our Transporter demo we exercised as much of this functionality as possible. It uses a mix of baked and dynamic lighting, and uses all of the dynamic outputs of Enlighten on a Samsung Note 10.1 (an octa-core device with 4 big and 4 little CPUs, and a Mali T628MP6 GPU).

Enlighten principles

- Pay for what you change
  - No change = no cost
- Dynamic lighting must be cheap
  - Enlighten lightmaps are small and cheap to update
  - Highly optimised runtime on all platforms
  - CPU workload, no GPU overhead.
- Do not sacrifice lighting quality
  - Some precomputation is better than bad lighting
- Provide artists with controls
  - Default settings → physics

There are some guiding principles behind Enlighten that make it practical and scalable.

The first is that runtime cost should scale with the amount of dynamic content. This is important because not all games need dynamic lighting.

Where dynamic lighting is used, we've put a lot of effort into making that cost as small as possible. Cheap enough that it runs on a phone, but capable of scaling up to the quality and quantity demanded of next-gen titles. By running Enlighten asynchronously on the CPU we also ensure we degrade well. It is possible to give Enlighten a CPU time budget and force it to remain within it. If the demands exceed that time then updates to the indirect lighting will lag behind slightly, but this is often hard to see. Slow moving lights are particularly cheap as their cost can be amortized over many frames, making time-of-day transitions particularly efficient.

We also believe that it's better to have some precomputation time than to suffer from bad lighting. Offline, we pre-process static geometry to capture visibility information we can use while doing lighting at runtime. This allows us to generate lighting comparable in quality to a offline render but with dynamic lights.

Finally, lighting is based on physics but may not be what is required. We expose many controls to allow artists to deviate from the physically-correct interpretation.

Enlighten in Unity 5

- Works with existing content
  - Re-import geometry with Enlighten AutoUVs
  - Tweak resolution settings
  - Lights, skybox, self-emissive materials all work
- New dynamic content
  - Dynamic lightmaps and cubemaps
  - Dynamic albedo and emissive surfaces
  - Enlighten script bindings
- Updated content
  - Baked lightmaps and light probes
- Real-time lighting workflow
  - Heavy lifting done incrementally in the background
  - Scene edits automatically trigger re-builds

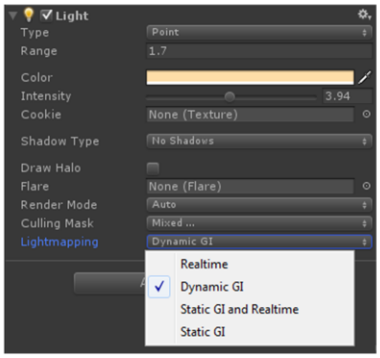
Content authored in Unity 4 will work straightforwardly with Enlighten in Unity 5. In particular, all the existing light types, skybox, and emissive materials will work in Enlighten as expected. We put in a lot of time with Unity ensuring that this will be as smooth a transition as possible.

The new dynamic lightmap output from Enlighten needs an additional set of UVs to work. These UVs are not the same as the existing lightmap UVs as they reference the smaller, simpler indirect-only lightmap, but are generated automatically by Enlighten as part of the mesh import options.

The dynamic lightmap and lightprobe output are folded into Unity's existing workflows so existing materials and custom surface shaders need no special treatment to work with them. However, dynamic cubemaps are a new resource and because there is no single workflow that would suit everyone, they are exposed as script bindings to be incorporated as you wish. For Transporter we built a simple C# cubemap binding system and surface shader that added an indirect-specular lighting term.

Enlighten in Unity 5 - lighting

- Select baked or dynamic per-light
  - Lighting matches in either
  - Use dynamic GI to light in real-time
  - Bake where necessary for performance
- Soft shadows
  - Baked lights support a wider range of soft shadowing
- "Enlighten-only" lights
  - Easy to create hidden bounce lights
- Artistic controls
  - Per-light and global GI controls
  - Freedom to go beyond the physical model



The decision of whether a light is static or dynamic is made per-light. The UI could change before the official release of Unity 5, but at present there's a simple dropdown menu on each light that allows you to toggle between "Realtime" (no GI), "Dynamic GI", and statically baked.

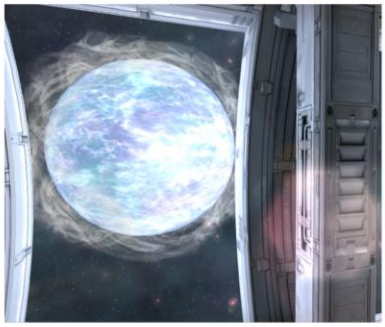
The overall illumination matches between the dynamic and baked versions, so lighting can be done in-editor using dynamic lights, and then individual lights can be baked for performance. The main difference between the two is that shadowing from static lights is ray-traced and so can capture high quality soft shadowing effects.

By changing the culling flags it is also possible to put lights directly into Enlighten, effectively creating indirect-only lights. This can be handy for creating artificial sources of illumination, or simply tweaking the indirect lighting.

Other artistic controls include the ability to modify how much bounce lighting any particularly light has. The default is 1.0, and can be boosted or cut as you wish. There are additional global scales that have the same effect but apply to all lights. It is also possible to adjust the degree to which light is allowed to continue to bounce around the scene. A "bounce scale" value of 1.0 will allow lighting to bounce around until it is fully absorbed, while a value of 0.0 will only allow it to bounce once. This can help control the amount of contrast in a scene and adjust how far the influence of a light will travel.

Materials

- Enlighten supports dynamic materials
- Make surfaces glow
  - Change them in-game in real-time
  - Really cheap / "free"
- Example: soft illuminaton in Transporter
  - Enlighten includes real-time illumination from the skybox for environment light
  - But we also had a big procedural planet courtesy of Allegorithmic…

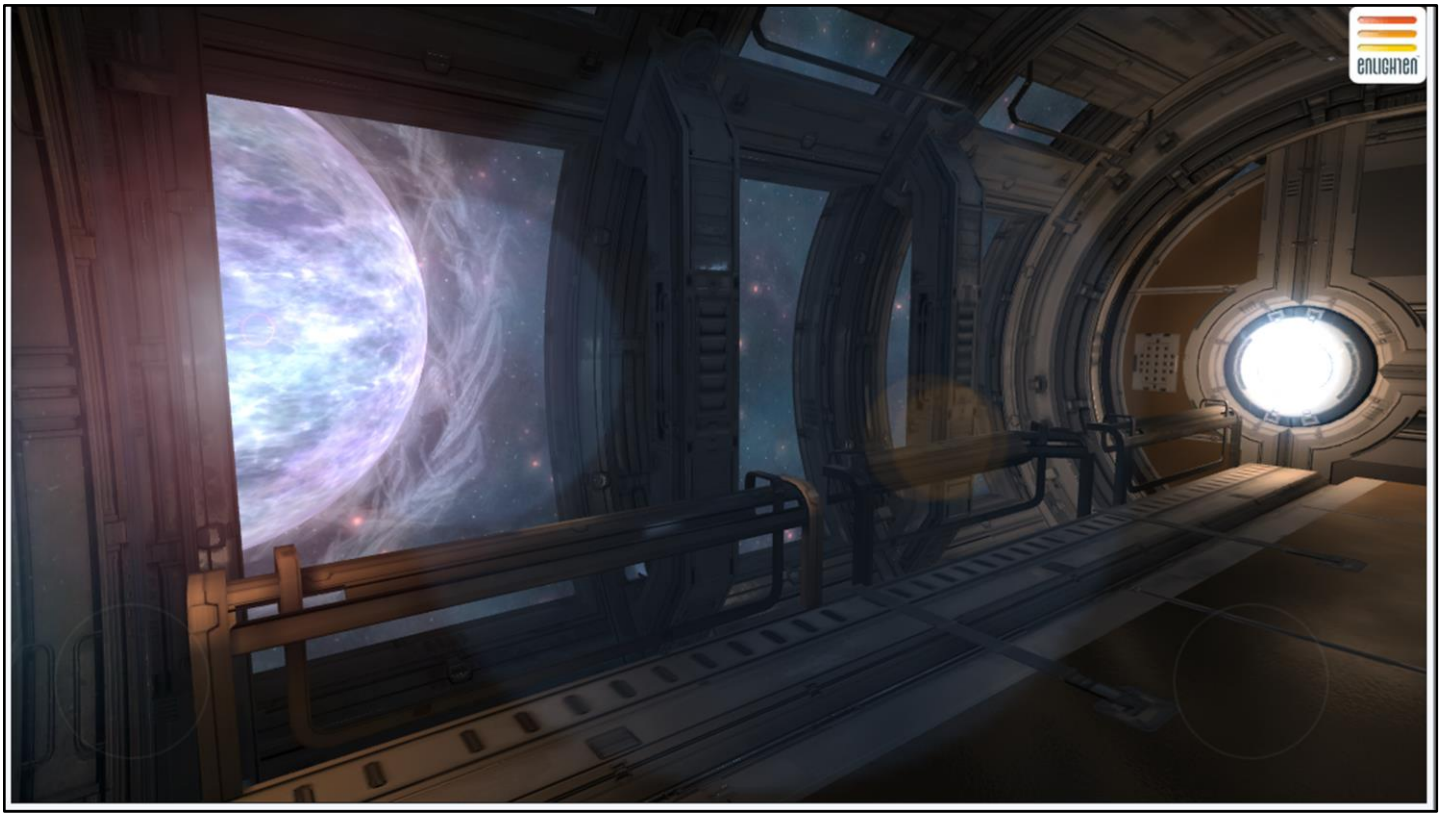


allegorithmic

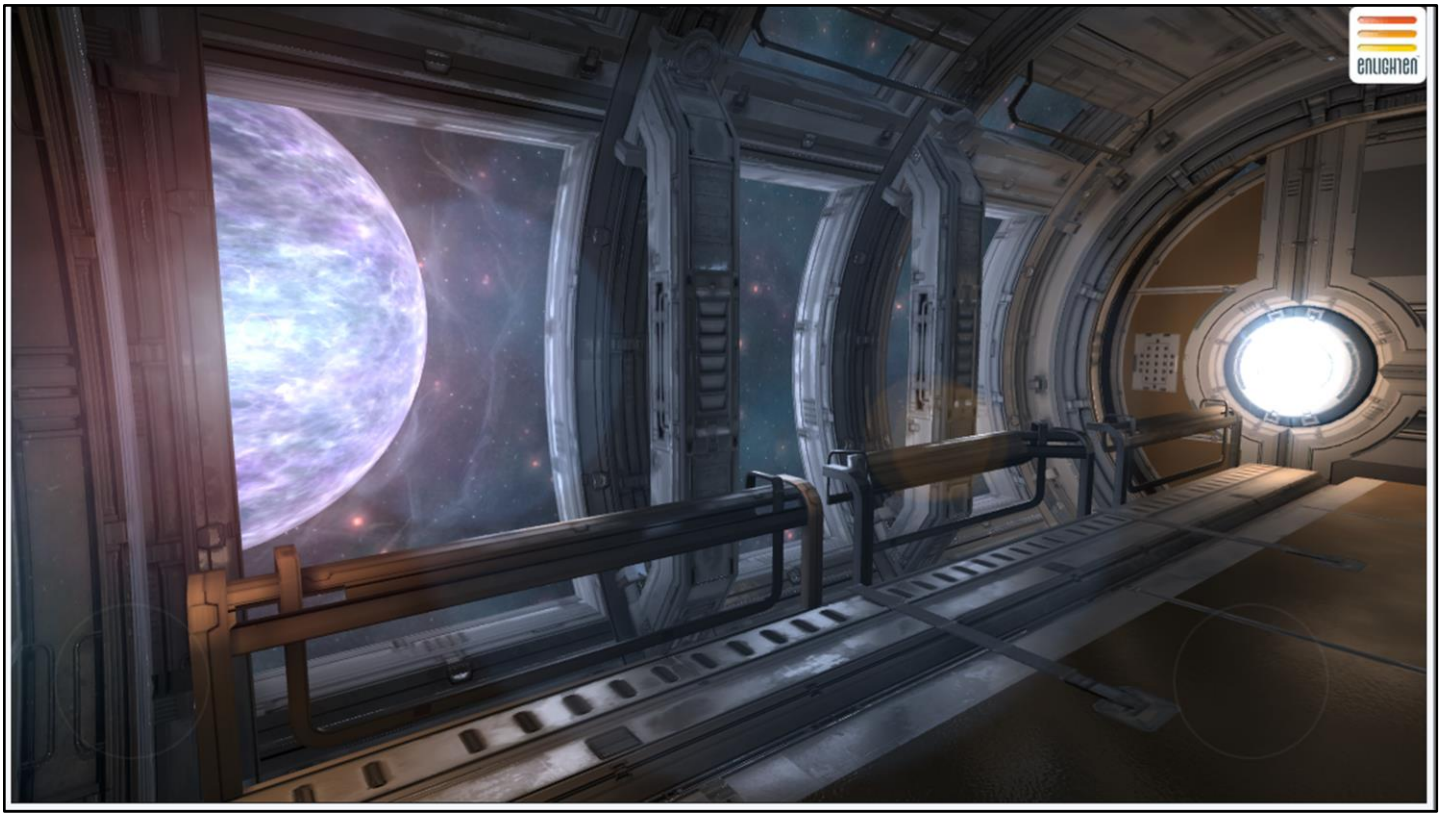
Enlighten doesn't bake in any assumptions about materials, so you are free to change the surface albedo (diffuse colour) and emissive values dynamically.

If you are updating Enlighten regularly anyway, because you have some dynamic lighting in your scene, it turns out that changing emissive material values are effectively "free".

Unity 5 includes simple script bindings to change the emissive surface colour per mesh renderer-material. We used this to create dynamic area lights in the Transporter demo.



A few days before shipping the demo, this was the view of the main corridor. We had a nice glowing planet from Allegorithmic, but it didn't have any corresponding illumination. We could have added a point/spot light but this wouldn't have had the same soft lighting effect as an area light.



Instead, we placed a large white quad outside the window and made it emissive. This provides the soft diffuse illumination you can see around the window frames and in the interior, as well as the indirect specular reflection on the floor.

Hidden geometry

- Added hidden emissive quad
- Trivial to animate in a script

- Technique previously used in Quantum Conundrum

In the top image you can see the quad in the editor. During the game it is hidden but it's illumination remains.

This kind of hidden geometry is a helpful source of illumination and something we have used many times before. Emissive surfaces were used by Quantum Conundrum to great effect where they were the source of most of the illumination. As well as being "free" in terms of Enlighten (as long as you are already updating it), they are also truly free in terms of GPU time, as they have no direct component. All lighting comes through the dynamic outputs. This can be a helpful dynamic addition to "normal" baked lightmaps on platforms that don't have much GPU resource.

Physically-based rendering

Physically-based rendering (PBR)

- Consider different sources of illumination
  1. Lightmaps, cubemaps, light probes
     - Cheap to use in PBR
     - Good approximations exist
  2. Direct lights
     - More expensive in PBR
     - Focus of much next-gen console R&D
     - Must approximate more aggressively on mobile

Physically-based shading is more of a discipline than a technique. It also has a reputation for being computationally expensive, and this is certainly true of an accurate reference implementation of a micro-facet model. In practice, approximations are employed and benefit from the advantage that they can be directly compared to a reference model.

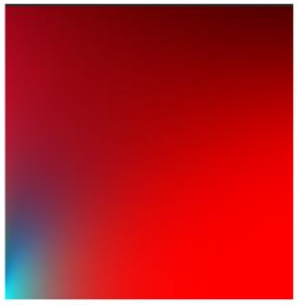
We loosely group illumination into two different sources: Those that are represented by lightmaps, cubemaps or lightprobes, and those that are computed directly. The common theme with the first set is that they represent the illumination (direct or indirect) from multiple sources of light. Conveniently, they are also easy to handle in a physically-based manner without having to abuse the term physically-based too much ☺.

Handling of direct lights typically involves some form of analytical integration and is currently just out of reach for practical purposes. In our demo we did not attempt to do anything special here. The dynamic cubemaps provide much of the specular term, so the illumination is consistent, but this is something that we should expect to improve upon as mobile compute capabilities improve.

A great reference for all things physically-based is the SIGGRAPH Physically-Based Shading course, particularly Brian Karis' talk.
Brian Karis, **Real Shading in Unreal Engine 4**
http://blog.selfshadow.com/publications/s2013-shading-course/
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

Physically-based rendering (PBR)

- Transporter demo
  - Physically-based indirect and environment terms
  - Direct lights are simplified
  - A more efficient lighting culling approach would help significantly
- Precomputed material response
  - VoN x Roughness
  - See [Karis 13]



Next-gen secret sauce

- Cubemaps!
  - 1986 all over again [Greene 86]
  - Analytical models of specular don't really cut it
- On next-gen:
  - Mix of screen space reflections and cubemap fallbacks
- On mobile:
  - No screen space reflections, more emphasis on cubemaps
- Cubemap mip-maps
  - Input to physically-based shading system
  - Supported in OpenGL® ES 3.0 or through EXT_shader_texture_lod

Another trick coming out of next-gen rendering is the resurgence of cubemaps. Invented many moons ago they have made a bit of a comeback.

It turns out that simple analytical models of your lighting environment don't really cut it. They may model direct lighting well (if your analytical model is complex enough) but indirect illumination needs some kind of texture data.

On next-gen there's a common pattern of:
- If you can get a decent screen space reflection, use that.
- Else, fallback to local cubemaps, else fallback to some environment cubemap.

On mobile we can do something similar, but skip the screen space reflection part as it's too expensive. Enlighten's dynamic cubemaps come with mipmaps and if you have a GPU that supports explicit lod lookup (which on mobile means es 3.0 or es 2.0 with an extension) then you can use these as the lighting environment for your indirect specular term. Again, see Brian Karis' talk for how to combine this with a precomputed physically-based material response.

Refs:
Ned Greene, "Environment Mapping and Other Applications of World Projections". Ideas predate this (Jim Blinn, Martin Newell).

Cubemap parallax

- Cubemaps lack depth information
  - How far away is this illumination?
- Provide additional input to approximate depth
  - We used a box for each cubemap in Transporter
  - Can build any solution on top of Unity 5
- Could also
  - Sphere mapping, any convex geometry
  - Ray-march through cube depth map
  - Special-case floors
  - Seb Lagarde, 'behc' on gamedev.net – see slide notes for references

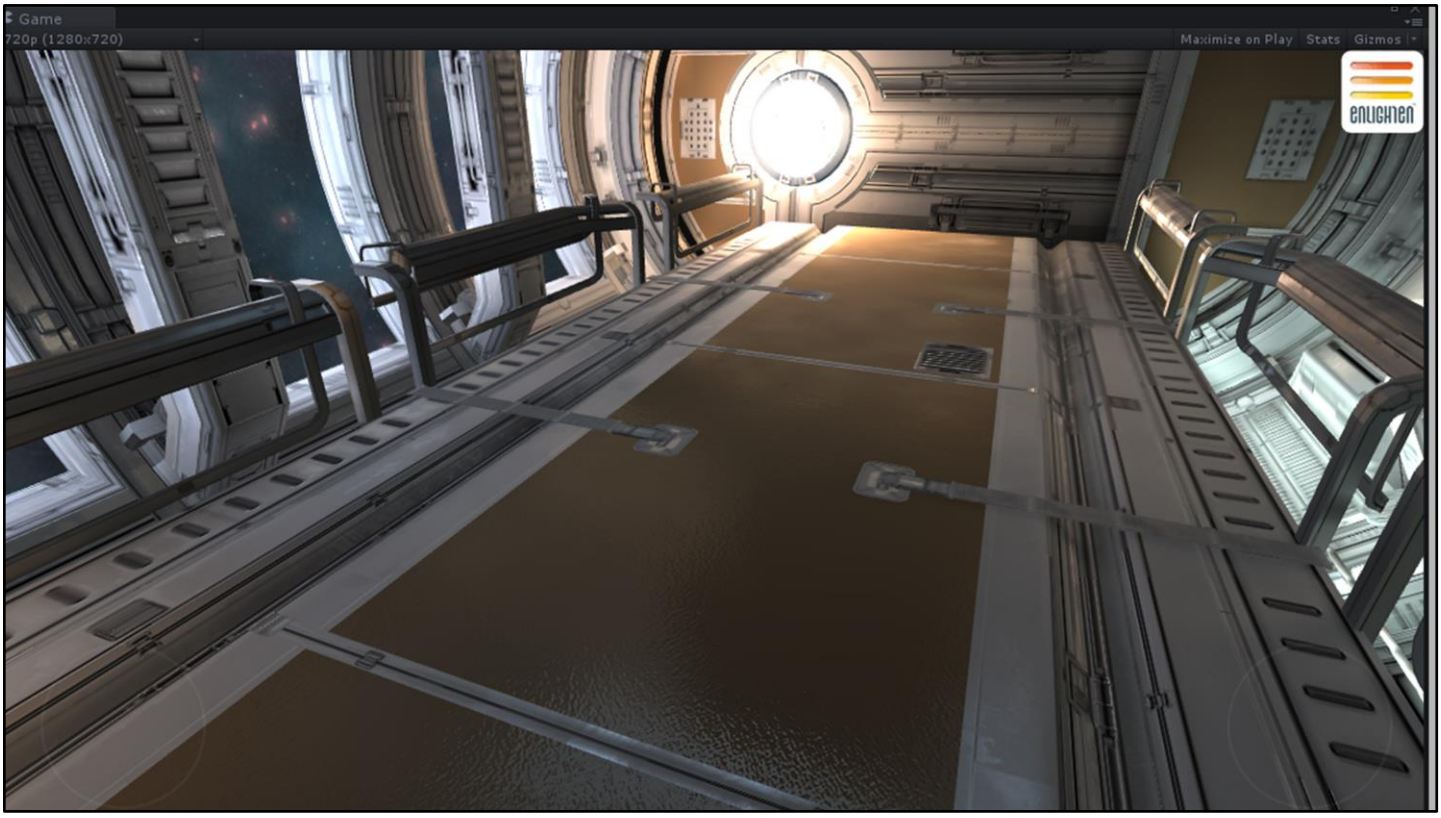
The main problem with cubemaps is placing the reflections properly. Without any further information, you have to assume that illumination in a cubemap is infinitely distant. If you were to look at the floor and see a reflection of the sky, moving around on the floor won't change the reflection. This is fine for a sky, but if the content in the reflection is a nearby surface, such as a light on a ceiling, then moving across the floor should change the reflection.

To fix this we need to generate some information about how far away from the cubemap centre its contents are. There are many options but we opted for authoring an additional box alongside each cubemap. The box represents the location of the reflections contained in the cubemap, and because the geometry is simple we can "ray trace" our reflection vector against it to find the correct point of intersection and work back to how to lookup into the cubemap from that.

The details of how to do this have been covered a few times before. I would recommend reading 'behc's post on gamedev.net about this. He presents example source code and there's an .exe you can run. Sébastien Lagarde also wrote a number of blog posts and did a few talks on the subject. His blog is a helpful resource, so I recommend you dig about.

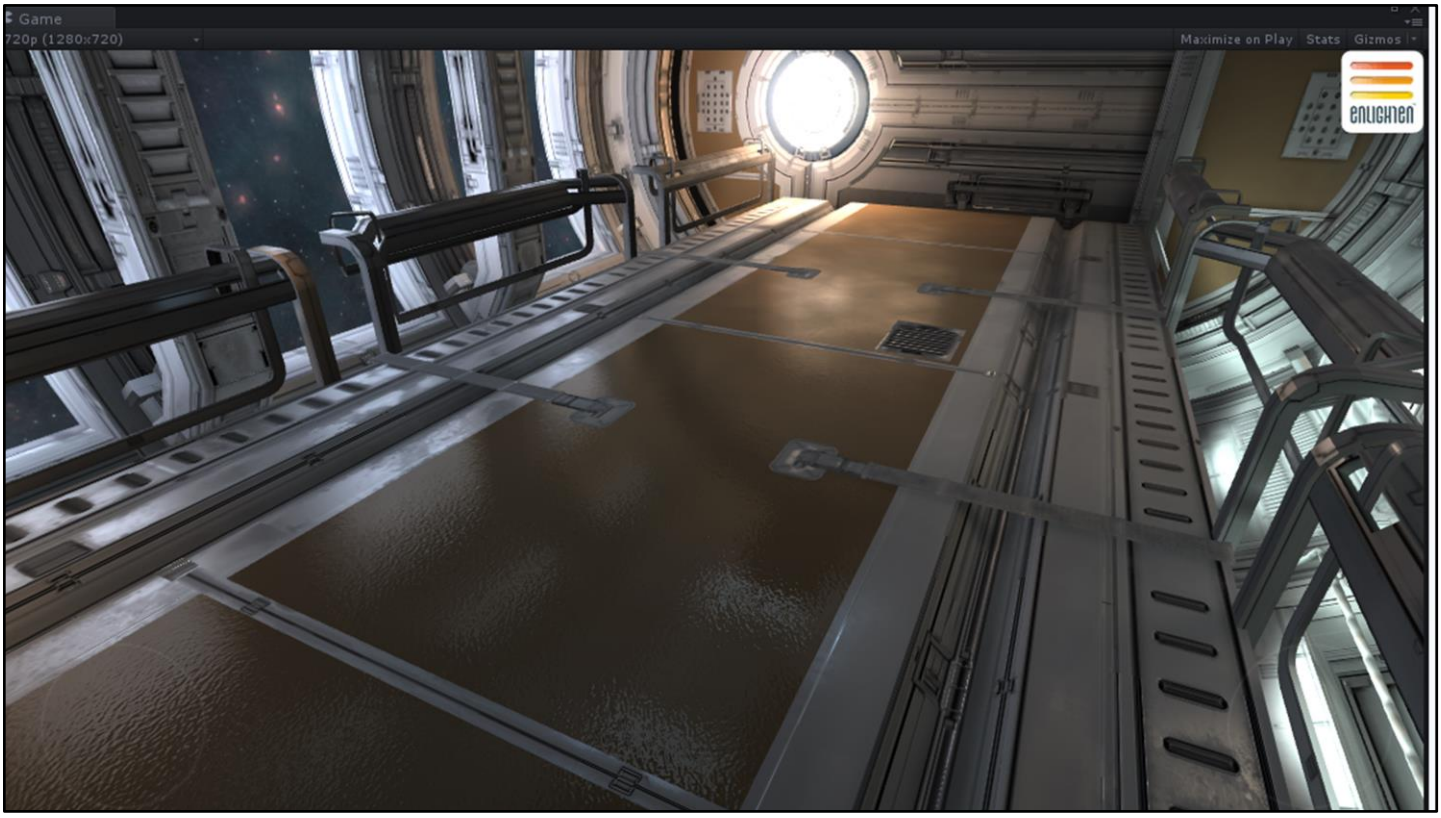
http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/

http://seblagarde.wordpress.com/2012/09/29/image-based-lighting-approaches-and-parallax-corrected-cubemap/



Here's the before/after.

This is without the effect of the authored box. There are some reflections,but they don't appear to match up with the rest of the scene.



This is with an authored box, placed to best fit the floor/walls/ceiling.

You can see the reflections from the windows (actually an emissive quad if you recall), the doors, lights and so on.

The potential of mobile

- Large gap between 100 W and 1 W
- Innovations required across the board
- ARM and Geomerics joint R&D

- **How do we reshape bandwidth-heavy techniques?**

The Transporter demo was a proof of concept for the Unity 5 integration. It was also a handy vehicle for seeing what we could squeeze into a shipping device, but this was not it's main motivation and I know there's a lot more we could have done.

One challenge was working with a limited number of dynamic lights. Rather than use the regular multi-pass approach employed by default in Unity, we adapted the main shader to always compute the 2 most important point/spot lights and then had to stay under this limit while authoring. It's also unnecessarily expensive as the contribution from the direct light is not significant and there are few pixel pixels that actually need 2 simultaneous lights. To make this more efficient we need to move to an alternative rendering system.

The obvious candidate would be deferred rendering, but unfortunately this is particularly expensive on mobile due to memory bandwidth limitations. In order to bridge this gap we have to look at different ways to achieve the same rendering efficiency improvements but avoiding the bandwidth implications.

This is one of the issues Marius will talk about next.

Agenda

- The "Now"
  - Real-time global illumination in Unity 5 with Enlighten
  - Physically-based shading on mobile
- The "Next"
  - Bandwidth-efficient graphics
  - Programmable blending
  - Pixel local deferred rendering

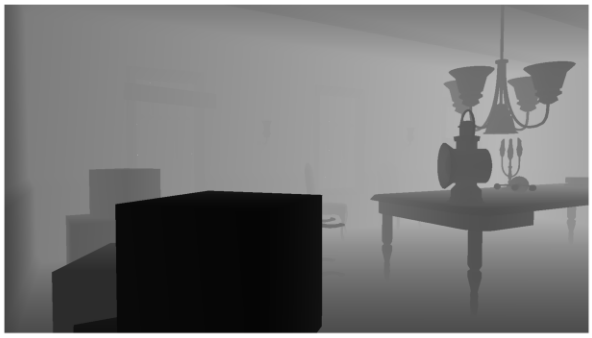
Bandwidth-efficient graphics

ARM

Bandwidth

Where does it go?

- Vertex load
- Varyings
- Textures
- Framebuffer output



ARM

When talking about bandwidth in the graphics pipeline, the main offenders are vertex load, varyings, texture lookups and framebuffer write outs.

Bandwidth

Where does it go?

- Vertex load
- Varyings
- Textures
- Framebuffer output

ARM

In modern games which apply various post processing effects, the texture and framebuffer output is often tied together. This means that memory is written out to memory and then read back in in order to make an effect possible.

The focus of this talk is on reducing this framebuffer-to-texture feedback loop.

Bandwidth

Where are we?

- Less power = less memory bandwidth
  - Desktop: 170 Watts to >300 Watts… That's just the GPU!
  - Console: 80 - 100 Watts (CPU/GPU/WiFi/Network)
  - Mobile: 3 - 7 Watts (CPU/GPU/Modem/WiFi)
- Need smarter solutions

ARM

While mobile GPUs are quickly catching up to the previous generation consoles (PS3, X360) in terms of compute performance, we are still trailing behind when it comes to available memory bandwidth.

The main reason for this is that moving data around consumes a lot more power than computing with it, so memory bandwidth becomes a major source of heat. So in a thermally constrained device such as a mobile phone you are basically limited by the laws of physics.

So we need some smarter solutions.

ARM® Mali™ GPU Rendering

- Mali is a tile-based deferred rendering architecture
  - Framebuffer is divided into tiles
  - Renders tile by tile
  - 16x16 tile size
    - Color
    - Depth
    - Stencil

ARM

Just to give you a quick background on how Mali rendering works.

Mali is a tile-based deferred rendering architecture. This means that we first do vertex shading, then all primitives are binned to tiles. When all that has been done for a particular framebuffer, fragment shading kicks off.

Fragment shading is done by rendering the frame buffer tile by tile, so all geometry that is binned to a particular tile will be shaded in draw order. When all the contents of a tile has been processed, the tile data can be written back out to memory.

The key giveaway here is that on a tile-based architecture - as long as you stay on-tile - all memory transactions will happen to fast on-chip memory. And this memory is something we have started exposing to developers by introducing a couple of new OpenGL ES extensions.

Extensions

Shader Framebuffer Fetch

- Fragment shader extensions for OpenGL® ES 2.0 and above
- Allows reading of existing framebuffer color, depth and stencil values
- Enables:
  - Programmable blending
  - Programmable depth/stencil testing
  - Soft particles
  - Reconstruction of 3D position
  - etc

http://www.khronos.org/registry/gles/extensions/ARM/ARM_shader_framebuffer_fetch.txt
http://www.khronos.org/registry/gles/extensions/ARM/ARM_shader_framebuffer_fetch_depth_stencil.txt

ARM

The first one is Shader Framebuffer Fetch. This is actually two extensions, one which give you read access to the previous fragment color and another one that gives you read access to the previous fragment depth and stencil values.

Again, since we're a tile-based architecture, this means that we're reading from on-chip memory - which is very cheap and does not access external memory.

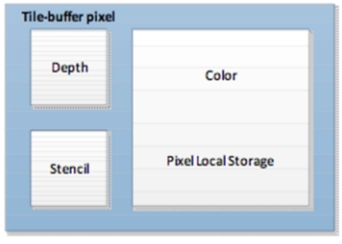
This enables use-cases such as programmable blending, programmable depth/stencil testing, soft particles and a convenient way of reconstructing the 3D position of any pixel.

The screenshot on the right shows the extensions put into use. The rendering in this particular image have separate passes for reflection and refraction. The refraction pass uses programmable blending to blend RGB color components with different per component blending weights based on the water depth. When rendering the water surface, the framebuffer depth value is read and used to softly blend the water surface with the nearby shoreline.

Extensions

Shader Pixel Local Storage (PLS)

- Fragment shader extension for OpenGL® ES 3.0 and above
- On the ARM® Mali™ -T600 series this amounts to 128-bits per pixel
  - Mali-T760 can support even more data per pixel
- Enables reading and writing the current pixel's data that is persistent throughout the lifetime of the framebuffer
- Independent of framebuffer format



http://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_pixel_local_storage.txt

ARM

The next extension I want to talk about is Shader Pixel Local Storage. This is an extension that we've been working on for the last year or so. It requires OpenGL ES 3.0 - that is mainly because of interactions with integer arithmetic.

So on Mali-T600 GPUs and above, each pixel is actually allocated 128 bits for color data. This extension allows the application to partition this data however they like - from 32-bit floats down to raw bit access. The layout of the Pixel Local Storage block is completely independent of the frame buffer format you're rendering to - so the memory written out in the end will still conform to the current frame buffer format.

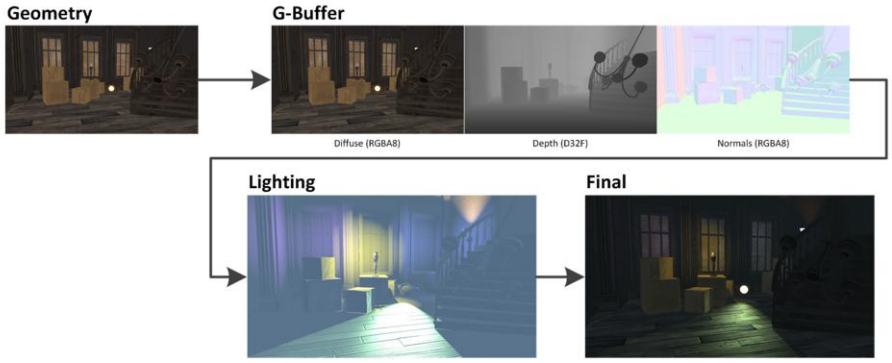
This enables reading and writing data that is persistent throughout the lifetime of the frame buffer, across draw calls and fragment shader invocations. This means that you can pass data between fragment shader invocations to build up your final pixel value in a much more flexible way than before. On mobile, where you really want to stay on-chip and do everything in a single pass, this extension is an extremely powerful tool for reducing memory bandwidth.

Deferred Shading

ARM

Deferred Shading

- Popular technique on PC and console games
- *Very* memory bandwidth intensive
- Traditionally not a good fit for mobile



ARM

Deferred shading is typically implemented using multiple render targets and requires large amounts of memory bandwidth.

It's been a popular technique on console and PC games for quite some time, but due to its bandwidth intensive nature it hasn't been popularized on mobile yet.

This is a perfect use-case for Pixel Local Storage. You can treat the Pixel Local Storage as you're typical g-buffer, all the data will remain on-chip so you're not wasting power and bandwidth when writing and reading the g-buffer data. Typically you would also combine this with Framebuffer Fetch to recover 3D position from scene depth. So you basically have all your g-buffer data available on-chip.

An alternative to deferred shading is to do multi-pass rendering – this adds a lot of geometry overhead, you potentially get poor light culling and there's no easy way to do proper HDR.

From the GPU and driver perspective doing deferred shading this way will be seen as a single pass - everything is rendered to the same frame buffer, only modifying certain parts of the Pixel Local Storage in the different render passes.

Deferred Shading

Resolve

- Compute final pixel color based on the Pixel Local Storage data
- Output to current framebuffer format

ARM

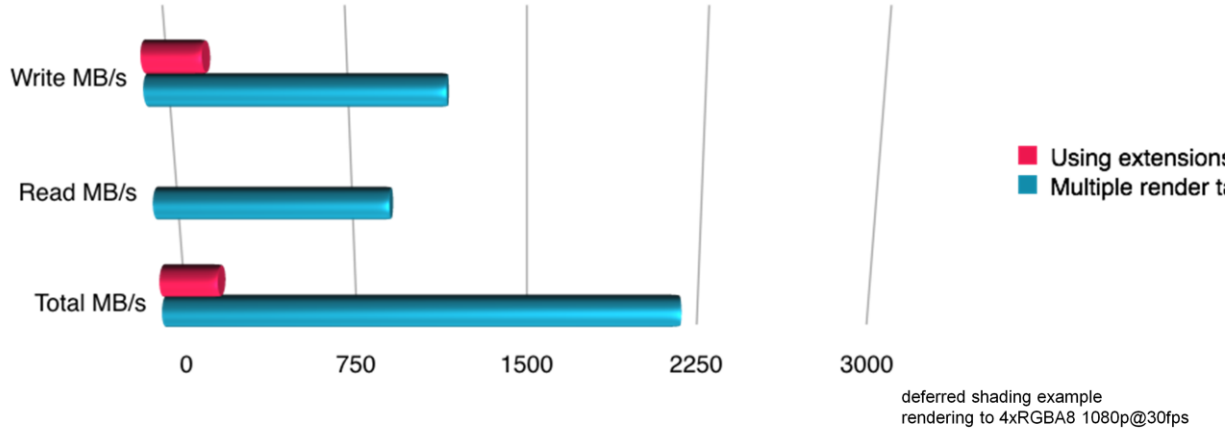
Using the Pixel Local Storage extension to do deferred shading requires a separate resolve pass at the end to ensure the output value conforms to the current frame buffer format.

This can basically be a full-screen quad that converts the Pixel Local Storage values.

Here you would typically also do your HDR tone mapping and possibly apply other post-processing effects.

Deferred Shading

Bandwidth Comparison



deferred shading example
rendering to 4xRGBA8 1080p@30fps

ARM

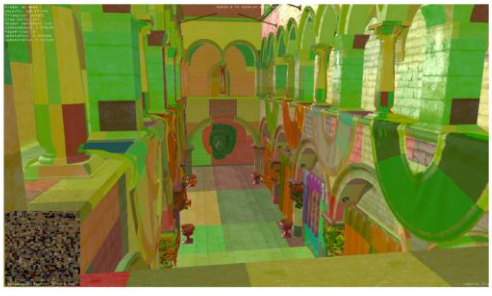
These are some external memory bandwidth figures.

As you can see, multiple render targets consumes 2,1GB/s, while the implementation with the extension only consumes 240MB/s. You can also notice that the implementation with the extensions only has a single write out to the main framebuffer, while the MRT implementation will have to write out g-buffer data, light accumulation, depth - then read most of it back in multiple times until it finally writes the final image to the main framebuffer.

So there's a huge win here when using the extensions compared to multiple render targets.

Future

- Various deferred shading/lighting
- Order independent transparency
- Deferred virtual texturing
- Volume rendering
- etc, etc, etc



http://geomerics.com/downloads/SIGGRAPH-2013-SamMartinEtAl-Challenges.pdf

ARM

And just to give you an idea of some of the other areas we've looked at for these things.

Various approaches to doing deferred shading/lighting. Sam also did a talk about this at Siggraph last year.
Various approaches to Order Independent transparency - Use Pixel Local Storage to store multiple fragment color and depth values
Deferred virtual texturing - Store virtual texture coordinates in Pixel Local Storage and basically do a single resolve to lookup into pagetable/physical texture.
Various approaches to volume rendering - store multiple entry- and exit-points of geometry in Pixel Local Storage.

So there are lots of opportunities here, and we'd love to hear if anyone else have good ideas!

Questions?

ARM

Thank You

marius.Bjorge@ARM.com
sam.Martin@geomerics.com

Also thanks to:
Jesper Mortensen
Kuba Cupisz
All at Unity and Allegorithmic

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Any other marks featured may be trademarks of their respective owners