
SIGGRAPH2012
The 39th International Conference and Exhibition
on Computer Graphics and Interactive Techniques

(This talk was given by Sam on the SIGGRAPH Mobile track at SIGGRAPH 2012, while Matt moved house)

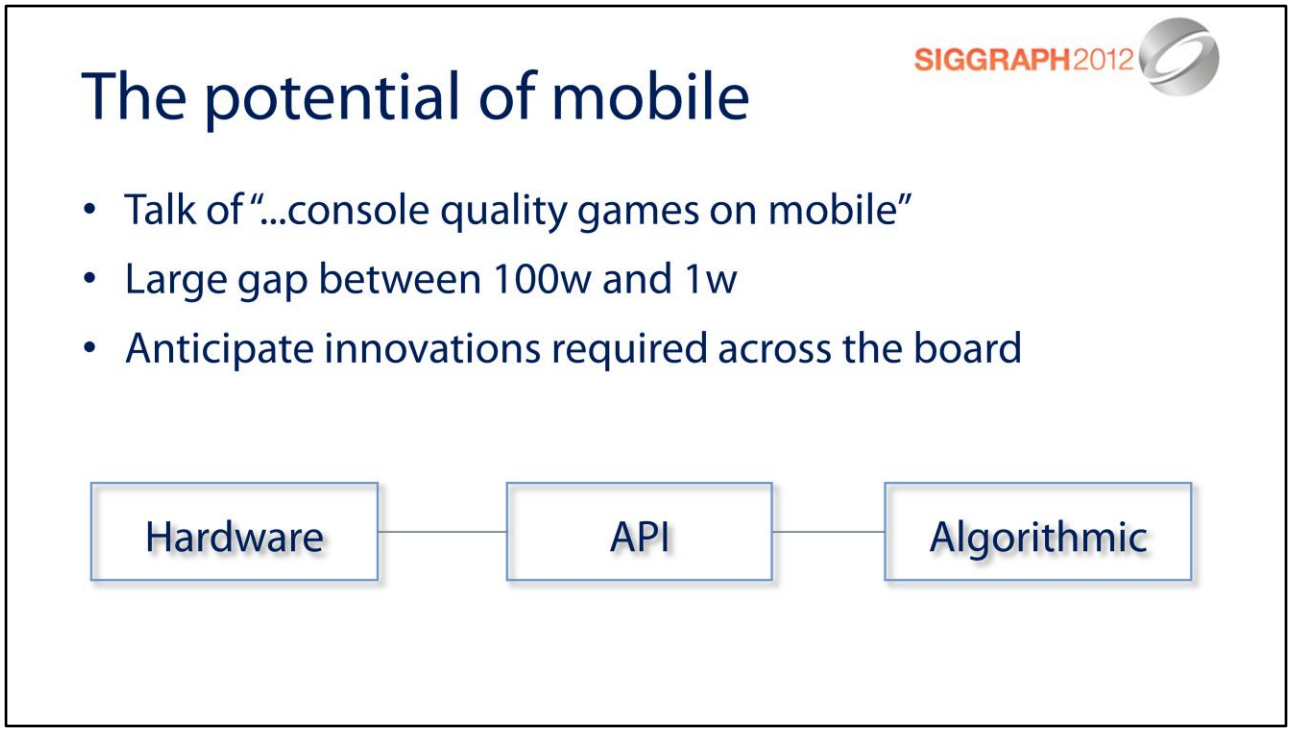
Hi!

First, to give you some background: We are from Geomerics where we make a real time lighting toolkit called Enlighten. Enlighten is lighting middleware for high end game graphics, and includes a unique approach to realtime radiosity lighting for both in-game and offline usage. We've been working on Enlighten since 2006 and the first titles using Enlighten came out last year, including Battlefield 3, Need for Speed: The Run, EVE Online (and so on). This xmas we will see the next Medal of Honor using Enlighten.

So dynamic lighting has been a big part of our lives a while now, and our core focus has been on continually pushing up the quality bar of lighting for AAA games on PC and console. This is still our focus.

But over a year ago we started a new project to investigate high quality lighting on mobile. We saw mobile GPUs becoming significantly more powerful and felt they were close to the tipping point where current console graphics, including Enlighten, would be viable.

Our intention was to port Enlighten, explore what we could do with dynamic lighting on the top end of mobile GPUs, and then start to feel beyond the current state of the art.



We did this because we believed the mobile market was undergoing it's own mini-revolution, and we wanted to understand how much potential there was for high quality graphics on mobile.

Mobile graphics is currently in "wild west" state. As such, I will talk about some of the experiences we've had getting dynamic lighting running on a range of GPUs. But I've been assuming that the audience for this talk is also interested in the future as much as the present. So as well as describing our work and results today, I want to focus on the opportunities that exist in mobile graphics, and the large number of challenges that remain. It is possible to do cool dynamic lighting today, but it's the future work that's really interesting and exciting.

Now, the first thing to note is that the difference in power between a PC or console and a phone is huge. I tend to find that a lot of conversation about "console quality on mobile" assumes that this gap will be closed purely by advances in hardware and length of the current console cycle. There are other notable hardware factors that have helped, such as the increase of market interest in high quality GPUs, which has allowed hardware manufacturers to make bigger chips that cost more but also do more. But there's often little mention of software.

Despite some PR to the contrary, this gap has not yet closed. It's perhaps not that far off, but based on our experiences I think we should not be relying on advances in hardware alone to close the gap. There is simply a lot of potential for improvement in software. We should be looking to all areas of the software stack for innovations that will help. I'm a software guy, so this talk will focus on the API and algorithmic ends of things. So the main theme is: What can we do in software to really bring high quality lighting to mobile?

Enlighten Palace App

SIGGRAPH2012

- iOS & Android
  - Wide range of GPUs (4-40 fps)
  - Use extensions where possible
- Geometry and textures as original PC/console targets
  - 8x 1024^2 albedo + normal maps
- Focus on dynamic lighting
  - Heavy per-pixel work

Well, you can't explore a new platform without writing some code. So we wrote a test bed app that also doubles as a demo for Enlighten. In the code there's a bunch of flags to control behaviour, and we build the app differently depending on what we are doing.

It's a fairly simple dynamic lighting demo with an asset we lifted directly from our PC/console version. The intention was to have an app where the feature set ticked as many lighting boxes as possible without becoming overly complicated. For instance:
- We use the touch screen to move the lights and camera.
- There is a dynamic shadowing sun (drag finger to move), and a time-of-day light animation.
- Enlighten is used for all radiosity, indirect specular, environment light, baked light configs.

It was important for us to cover a wide range of devices and platform options, which allows us to test how different techniques work out across the board. A fair amount of work went into this. We spent roughly the same amount of time of each GPU and discussed performance with the hardware guys to take out any low hanging fruit.

I won't go into detail on per-hardware figures in this talk. But I will note that we see a very wide range in performance in mobile GPUs, even when restricted to top of the line models. There's more than an order of magnitude different just in mobile GPUs, which is quite a scaling problem. But on the other hand, we can't really compare apples to apples here, because some architectures lack extensions which force us to do expensive things.

1. Per-pixel materials
2. Per-pixel lighting
3. Dynamic shadowing
4. CPU lighting

Heres' a rough categorisation for the talk today.
We'll cover per-pixel materials and lights, where I really mean "normal" lights ie. Directional, spot, point...
We'll then talk about shadows and finally CPU lighting – where we will focus mainly on indirect lighting (bounce lighting, environment).

But this is a rough outline only, and there are some recurring themes that I think are interesting.

1. Per-pixel materials
2. Per-pixel lighting
3. Dynamic shadowing
4. CPU lighting

First up...

So lets start by imagining that we'd like to see this on mobile. It's an asset we are currently working on in-house.

The lighting is simple – just sun + environment + radiosity.

If we strip away the albedo, you can see just the surface shape and lighting.

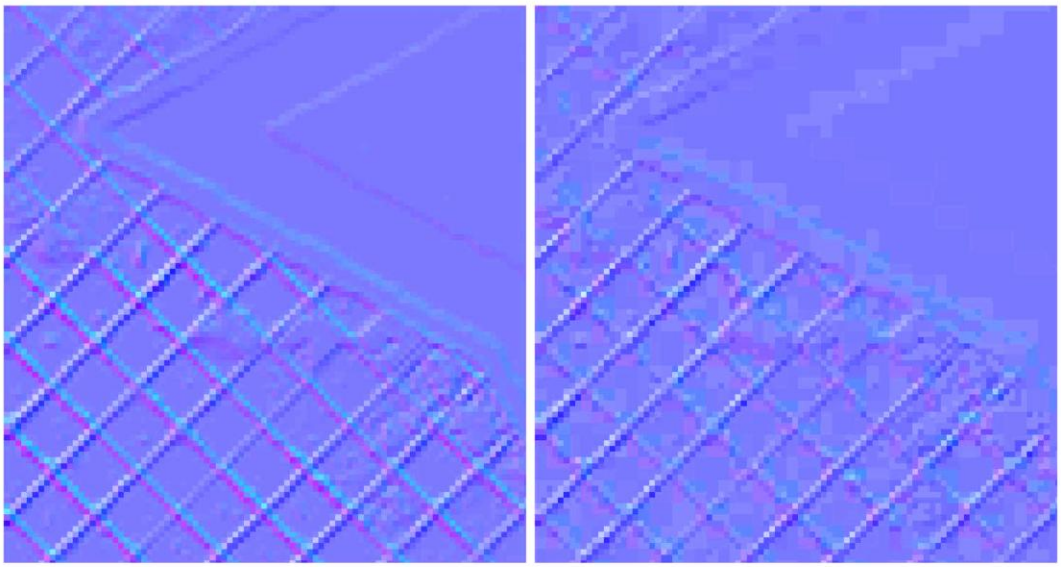
This asset has a lot of normal maps, which is where all the detail comes from.

It’s kind of obvious to state, but this asset shows why we need decent per-pixel materials in order to do worthwhile per-pixel lighting. It’s a combination of the two (materials and lighting) that we actually want.

If we strip away normal maps, the scene looks a bit more ordinary.

Perhaps higher poly than usual, but it's definitely the interaction of the lighting and normal maps that's important here.

Before/after ETC1

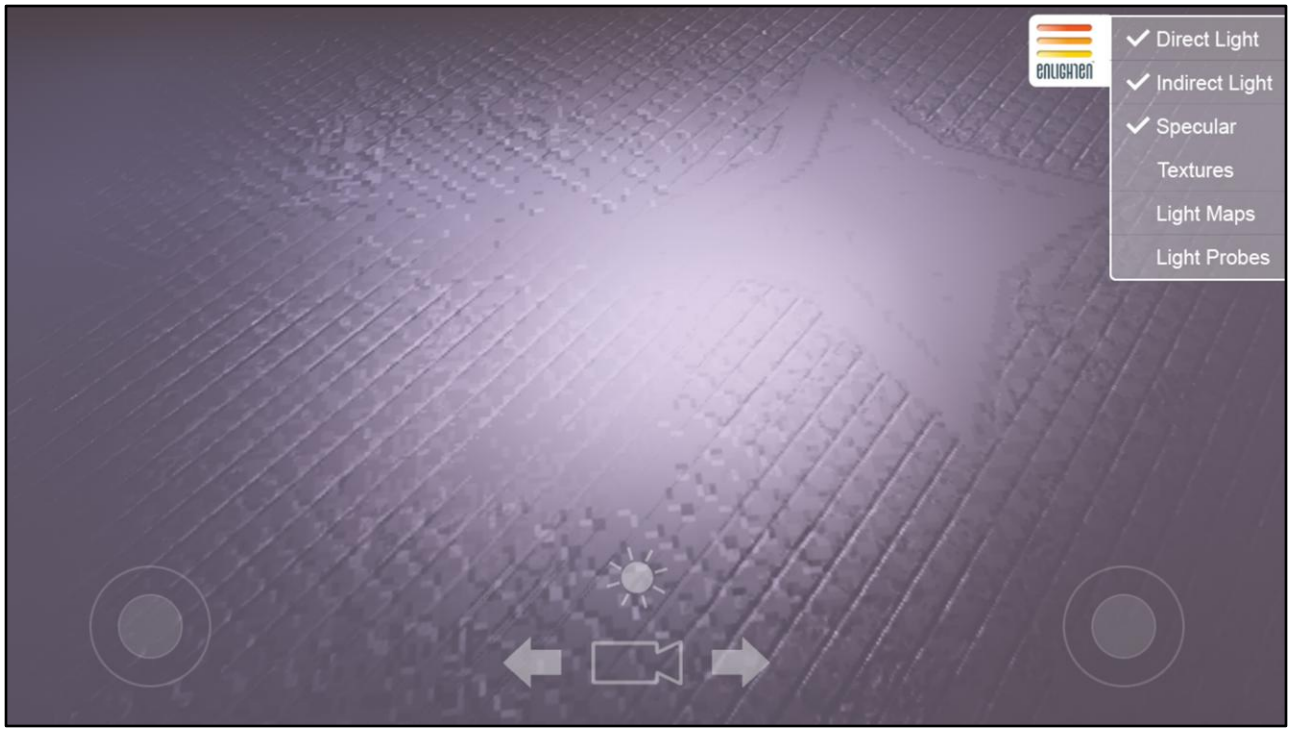


So, we'll need to start by getting some normal maps and compressing them.

Here we hit our first problem – normal map compression is not exactly a strong point of the ubiquitous ETC1 format.

(Have increased contrast in the picture for display purposes)

I'll skip a detailed explanation of exactly why ETC1 isn't very suitable, but suffice to say, it's the encoding format that kills this. Not much we can really do about it.



Here’s the same ETC1 compressed texture in use with indirect specular highlight.

Note blocky patches near bottom of image and general collapse of definition.

Oh dear.

Per-pixel normal maps

- Essential to high quality lighting
- Compressed normal maps
  - PVRTC supports normal maps (iOS, some Android)
  - ETC1 gives poor results (all Android)
  - Uncompressed "important" normal maps (universal)
  - Other hardware-specific formats?
- Should be a temporary blip ☺

Well, it's not quite that bad. If you rely on hardware-specific formats you can do better.

In particular, PVRTC (which is available on all of iOS) is ok. And you can get reasonable mileage out of keeping "important" normal maps uncompressed, and ditching the rest. This is what we do on devices that only have ETC1 - we keep the normal maps on the floor, and remove the rest.

I think it's fair to say that this is a pretty temporary blip. It's a well known and understood problem, and….

… breaking news!

There is now an extension to Open GL ES2 and ES3 (also recently announced) that introduces a very cool new texture format called ASTC. For an overview, check out the High Performance Graphics talk from this year: "Adaptive Scalable Texture Compression", Nystad et al.

http://www.highperformancegraphics.org/media/Papers/HPG2012_Papers_Nystad.pdf

I'd agree with Aras that it's awesome. So, in principle we just need to wait for this to make it to hardware.

It's regrettable that it's not part of ES 3.0 by default (which incidentally adds ETC2 by default, but this format also isn't much use), but not much can be done about that.

Albedo, specular?

- Specular, gloss, cubemaps?
  - Fine in theory…
  - but expensive!
- Textures with colour not a strong point of ETC1 either...
- Need to pick your battles

```
float ApproxPow(float x)
{
    // See http://bit.ly/3Eefi4

    // For a pow() exponent of 32, M = 16
    const float  A = 1.959;
    const float  B = -0.959;
    // max(Ax + B, 0)^M
    float r = max(A * x + B, 0.0);

    // log2(16) = 4 successive multiplications
    return r *= r *= r *= r *= r;
}
```

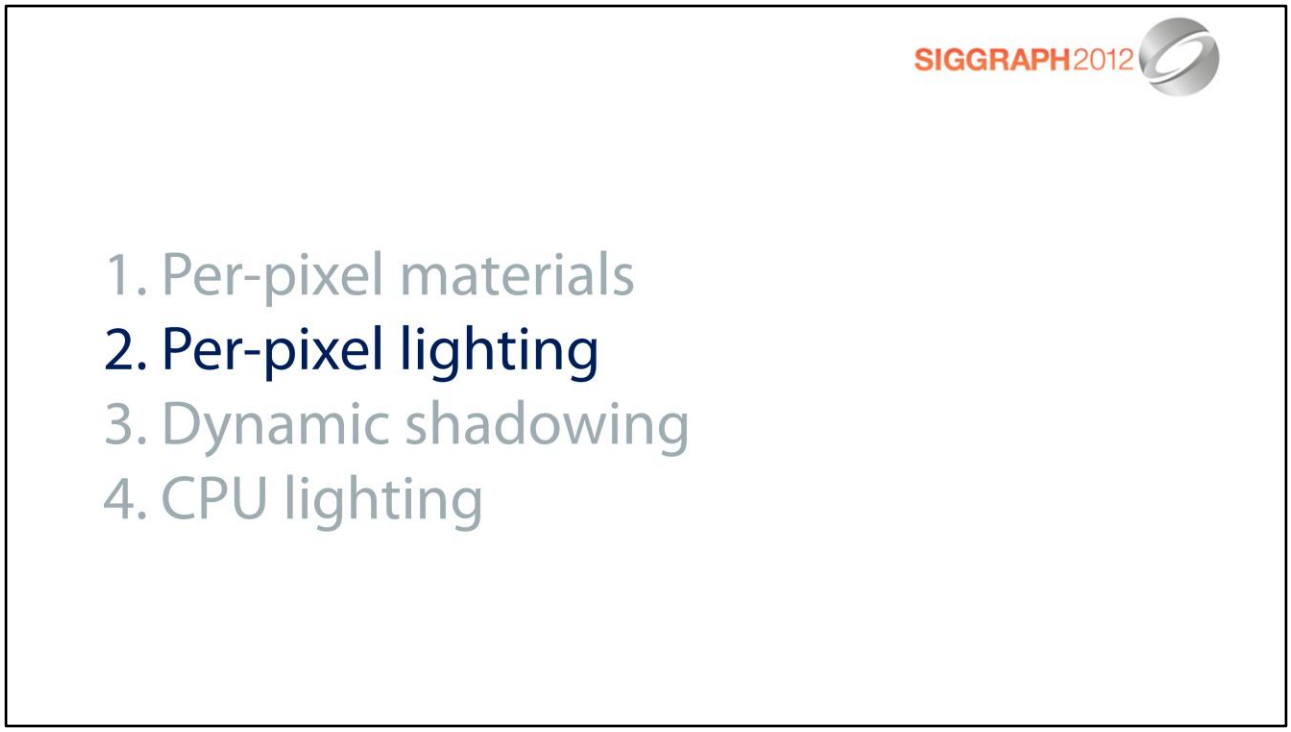
What about other per-pixel material details? Well, it's possible in theory at least. The API is in some sense ahead of the hardware here.

The main issue is that you are currently very constrained by the number of shader instructions you can run. We are almost always pixel shader bound, and shaving off instructions pretty much always speeds things up. There's no missing features as such, just a very limited budget for useful work.

In particular, pow is very expensive – and quite common in lighting! - but can be approximated. We chose to use an approximation dug off gamasutra, but another option would be to try a 1D lookup table in a texture. We didn't try this option, suspecting that the approximation would be better, but this guess might turn out to be wrong. A texture would at least allow you to change the power on the fly, as we are effectively hardcoding it.

We have to try and pull all work back to the vertex shader, and cut as many corners as possible. There are a lot of obscure overheads and micro optimisations required. Most of this optimisation is also done "blind" as profilers and low level tools are a bit patchy at present (but improving).

In short, you have to pick your battles, and we'll just need to wait for the hardware to improve some more.



Lets look at lighting

Rendering options

SIGGRAPH2012

- On PC/Console
  - [Tile-based] [clustered] deferred
  - Light pre-pass
  - Forward+, …
- On mobile
  - Single pass, forward rendering → **per draw call culling**

So how should we render with a bunch of lights?

There are lots of rendering options on PC/console.

Mobile is more… traditional ☺

At the moment you are largely back to the old days of combining lights into SHs, or building shader combinations (what we are doing), or just baking stuff (which we also do).

Mobile deferred lighting?

SIGGRAPH2012

- Viable on at least 1 GPU today
- Light pre-pass prototype [Yeung 12]
  - iPad2… but 480x320
- Multiple render target support
  - Not in OpenGL ES 2.0, but is in 3.0
- Another temporary blip ☺

Well, to say deferred lighting is not possible is not wholly true. It is on the edge of being viable, and definitely possible on some devices.

In particular, anything with a PowerVR 5xx MP4 (PS Vita, iPad3), has enough head room and sufficient extensions to do light pre-pass rendering.

One significant hurdle is the lack of multiple render target support in OpenGL ES 2.0. If you restrict yourself to a light pre-pass renderer and live with two passes over the scene, you can just about get it running. To be able to do light pre-pass, you must have the half_float or depth_texture extension to get sufficient depth precision.

This is in some sense a temporary hurdle, as the newly announced ES 3 will have MRT support by default, and other hardware is already showing that also has sufficient performance, such as the Mali T604.

G-Buffers for lighting?

SIGGRAPH2012

1. Per-pixel light culling
2. Efficient lighting accumulation
3. Screen-space effects (e.g bloom, DOF, ...)

- Lighting does not need access to other pixels
- Rasterize depth -> use depth to drive/cull work -> shade

- Important challenge:

Effective, power efficient light culling

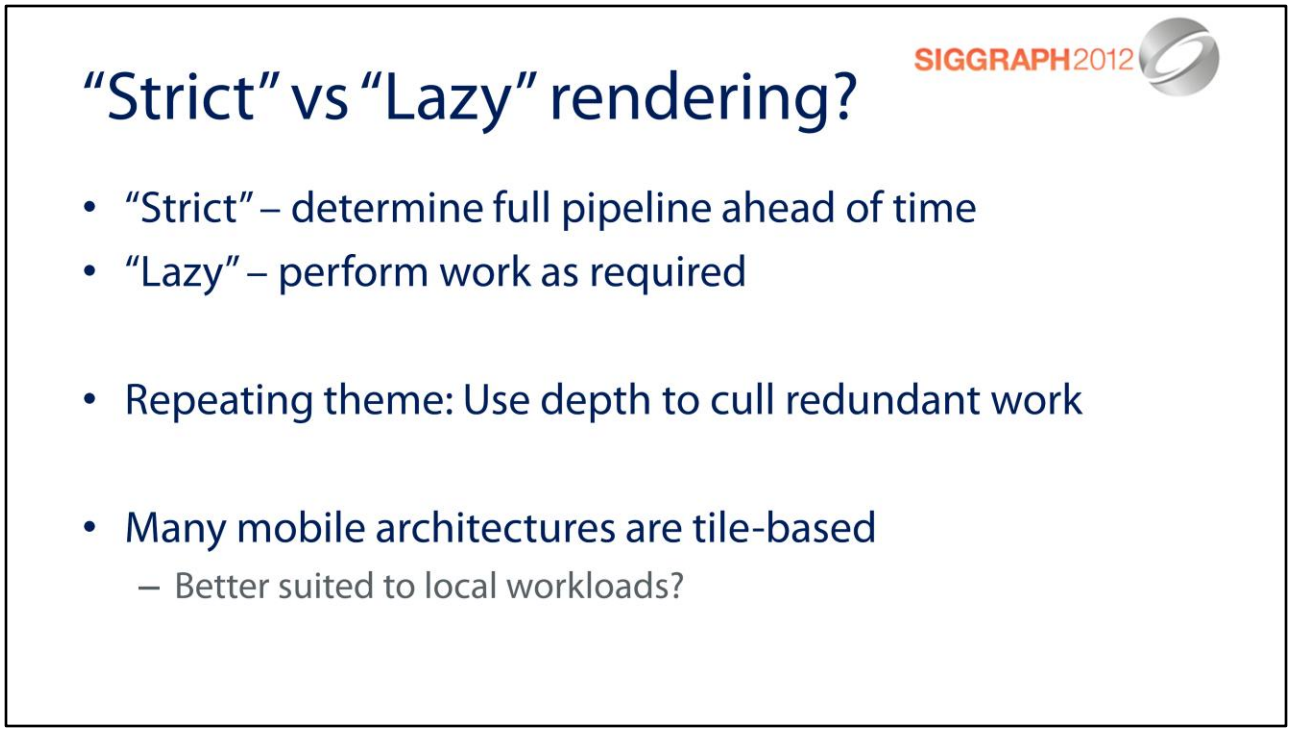
So far, so straightforward… but at this point I'd like to stop and question whether this is actually the best route for mobile.

Deferred rendering, and the large G-buffers that came with them were a great solution for discrete GPUs (100-300 watt devices). These architectures excel at maximising memory bandwidth – just what you need for working on large G-buffers.

But if you are only considering lighting (the first 2 points), not bloom etc, then you don't need them. When doing lighting, the pixels only ever access data associated with themselves, not their neighbours. You don't ever need the full set of buffers, just the data for the pixels you are currently working on. In this sense, lighting is more of a streaming problem than a raw memory bandwidth problem.

To come back to the point I made at start of talk: I think we should expect some innovations across the board to close the power gap on mobile. This one looks like an excellent candidate.

The key thing to get out of deferred shading (and variations on it) is efficient light culling. We are using rasterisation of both the scene and light geometry to efficiently compute the intersection between the two, so we only do work where necessary. Surely there is a more power efficient way to do light culling than creating huge buffers?



Here's a related, but more troublesome problem. The current graphics APIs force us to describe everything we want to render upfront. We must guess at what work will be required and submit the lot. This forces rendering to be conservative and submit more work than is actually required. It's not really possible to defer decisions until later in the pipeline – and so we end up doing too much.

To borrow a few terms from functional programming, what we currently have is a bit like a "strict" evaluation model. We always do all the work we might need, and do it upfront. In contrast, what we need in several situations is "lazy" rendering, where we only do the work once we discover that we actually need it. For instance, if we never see a light onscreen (perhaps because it's occluded), we don't need to compute the shadow map for it. The same goes for culling the drawing of entire objects. What if an object casts a shadow but nothing on screen actually receives that shadow? There was no point drawing it into the shadow map in the first place.

If you need to do this kind of culling at the moment you effectively have to duplicate the work the GPU will do yourself on the CPU - perhaps not the most power efficient approach.

It's quite easy to find situations where knowing whether something is actually drawn would allow you to optimise your workload. It's a pretty common theme, but the problem extends beyond this. If something is in the distance, or heavily blurred, you could simplify your shading (e.g. the decoupled shading paper).

Many mobile GPUs are also tile based. Not all, but a lot. Perhaps this would provide a natural level of granularity?

GPU or CPU?

SIGGRAPH2012

- Why assume GPU does all the work?
- CPU-side tile based culling?
  - API and synchronisation challenges

Q: What graphics could/should the CPU be doing?

Coming back to light culling: Along with working on the best approach for how the light culling should be done, is to work out where would be the best place to do it.

It's easy to assume that graphics implies running on the GPU. But all mobile devices (and a lot of non-mobile devices) are all unified. GPU solutions tend to be pretty brute force and less adaptive than what you might write for the CPU.

This is a point I want to stress: I think there is a huge opportunity to significant increase both the flexibility and capabilities of mobile graphics by making better use of the CPU.

Light culling is just one possible use. It's already very common to move all non-trivial vertex processing back to the CPU anyway, and I will give another example (CPU lightmap generation) in a moment.

But there's a huge stack of other problems the CPU could be helping out with: Early occlusion culling, screen space interpolation, procedural content generation, …

A related question is "how" should the CPU be included. This is a much more thorny issue: OpenCL? Extensions to OpenGL? It's a big issue to unpack, and not something I'm going to speculate on today.

1. Per-pixel materials
2. Per-pixel lighting
3. Dynamic shadowing
4. CPU lighting

Shadowing…

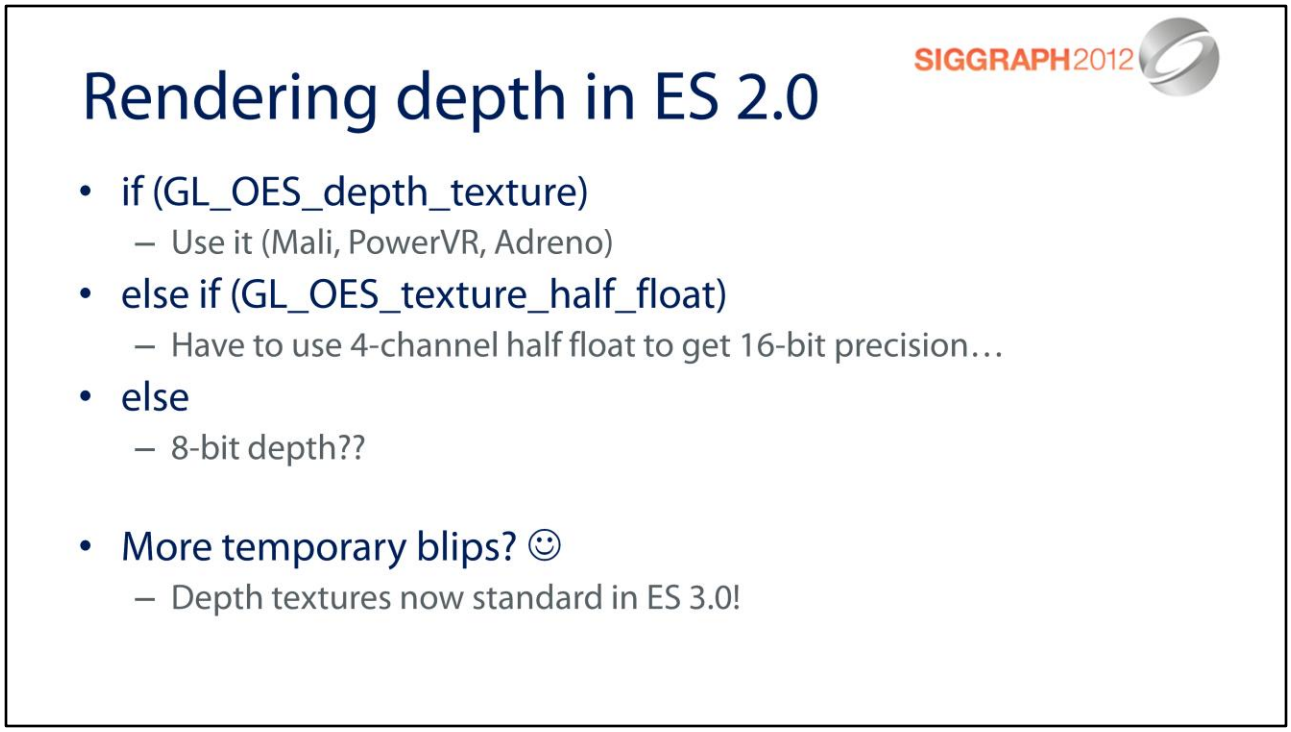
Shadow maps

- Rendering depth can be efficient
  - …but requires extensions in Open GL ES 2.0
- Good old PCF
  - Hardware extension support not free or widely available
  - Today: minimal set of taps only

Shadows on mobile at present boils down to:
- First: whether you can render depth efficiently enough (if at all)
- And then: whether you have enough per-pixel power to actually sample it

It appears that rendering depth is cheaper than sampling it. You can generate some fairly large shadow maps without melting your GPU, but don't expect to be able to take much more than a single tap at the moment.

So hard shadows are ok-ish, but the blocky edges are hard to kill off.



Rendering depth is actually a bit prickly due to the limited number of texture formats in OpenGL ES 2.0.

The main problem is that the depth_texture extension that allows you to bind a texture to the z-buffer isn't standard. If it is present, everything is fine, and thankfully this covers quite a lot of GPUs.

Otherwise, you have a few choices to make.

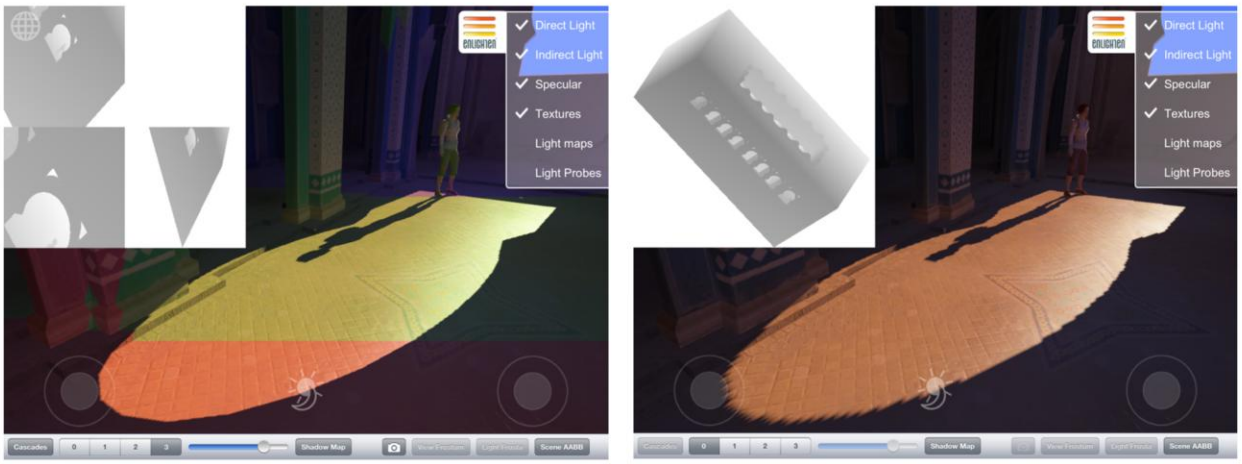
Single channel 16-bit formats don't really appear to exist, so if you opt for this as a fallback you are stuck with 4 channels.

We use 4-channel half-float as a first fallback, which – amazingly – actually works reasonably well on at least on one arch. But it's hardly the most efficient approach as we waste 3 channels.

It has been suggested that we could also try using a normal 8-bit RGB(A) texture and pack 24 (or 32) bits of depth across the channels. This may work, but we've not tried it yet.

I would largely expect these issues to just go away in the not-too-distant future. OpenGL ES 3.0 has depth_texture in by default.

With/without cascades @2k²



In our app we use shadowing for the sun. Full scene shadows from the sun is a tough challenge, so we thought it would be a good test case for mobile GPUs.

The traditional approach to this is to split the frustum up into “cascades” and compute a shadow map for each cascade each separately, then select the appropriate cascade in the pixel shader while shading.

So we gave this a go.

Cascade shadow maps

SIGGRAPH2012

- Required for decent sun light
  - Usable, but expensive
  - Fallback: 2k resolution for scene
- Unavoidable "dependent" sample
  - Branch-less lookup [Aras 2009]
  - 1-3 cascades. Can't do 4 – not enough varyings.

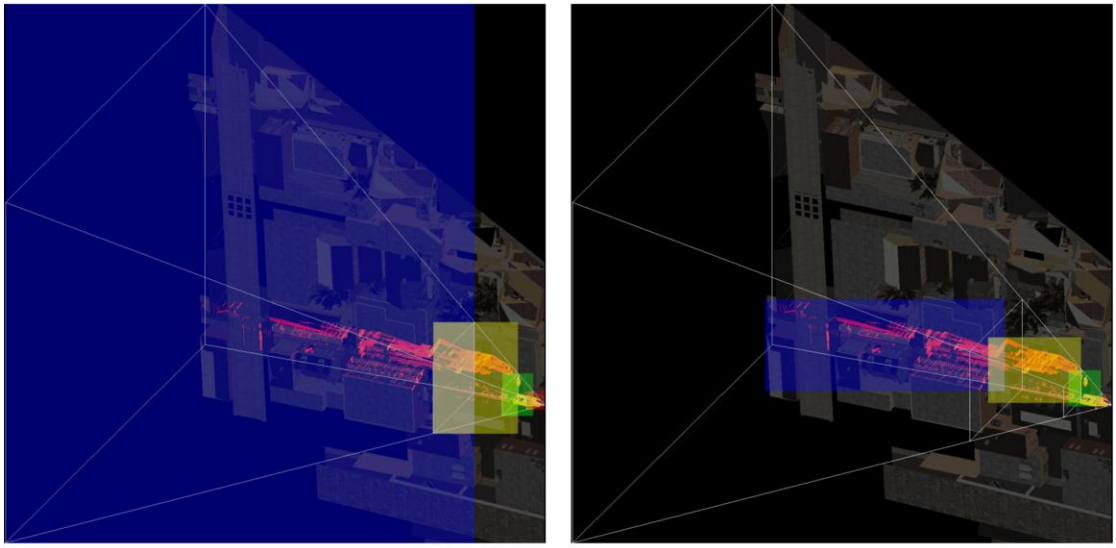
It turns out to be actually usable, but only on the most powerful devices at present.

Going beyond 2 cascades is difficult. The main problem is the limited number of varying slots as each cascade requires another set of UVs. Adding a 3rd cascade was significantly more expensive than the first two on most devices, and took us up to the limit on varyings, so 2 appears to be a sweet spot.

To do the lookup we use a single lookup with Aras's no-branch trick. Unfortunately, because we choose which shadow map coord to lookup based on depth, we get a "dependent" texture lookup, which is generally bad for performance. There isn't a huge deal we can do about this at present.

As an alternative we could take all the (non-dependent) samples and then pick the result we want, but this doesn't exactly sound very power efficient. I guess more advanced pre-fetch logic might help?

Sample Distribution SM?



Cascade shadow maps are there because we have a resolution problem. Our perfect solution would only produce shadowing information for the pixels we have onscreen, but this is not really expressible at present, so we simply try to match the shadow map to our on-screen pixels as tightly as possible.

Sample distribution SMs is an effective way of doing this, and is a really interesting technique to look for in the future on mobile. It's interesting not just because it's simple and effective, but also because it's completely inexpressible with the current OpenGL API!

CSMs are very wasteful in terms of the utilisation of the shadow map. We chop up the camera frustum based on depth, but it's very common that large areas of the frustum are not used. By calculating the actual depth bounds for the pixels on screen you can do a much better job. This requires the screen depth to be available and reduced to min/max ranges. (Very similar to the light culling problem early). Using the depth bounds we can produce much tighter bounds on the cascades. This is the SDSM technique Andrew Lauritzen et al introduced a couple of years ago.

There are 2 obstacles. The main obstacle is that the depth ranges need to be generated and used during rendering. There is just no way of doing this without stalling the GPU at present. The second is still resolution. SDSMs are great, but the cascades they generate are 'unstable' and you need a minimum resolution to avoid seeing crawling. Somewhere around 4x 1024 cascades are probably the minimum here, so we aren't there yet, but also not that far off.

1. Gather inspiration
2. <insert crazy ideas here/>

I hate shadow maps.

They are the best solution we have at present, but I'd still consider shadowing to still be an unsolved problem. There are huge resolution and aliasing issues, they tend to be very inefficient (in almost all senses), and they only really work for hard shadows.

If there's ever an area that could do with a bit more innovation it's here.

I'd also be open to the idea that a solution could involve dedicated hardware and not be particularly programmable. I'd rather have an inflexible, but working solution, than a flexible but unusable solution.

1. Per-pixel materials
2. Per-pixel lighting
3. Dynamic shadowing
4. CPU lighting

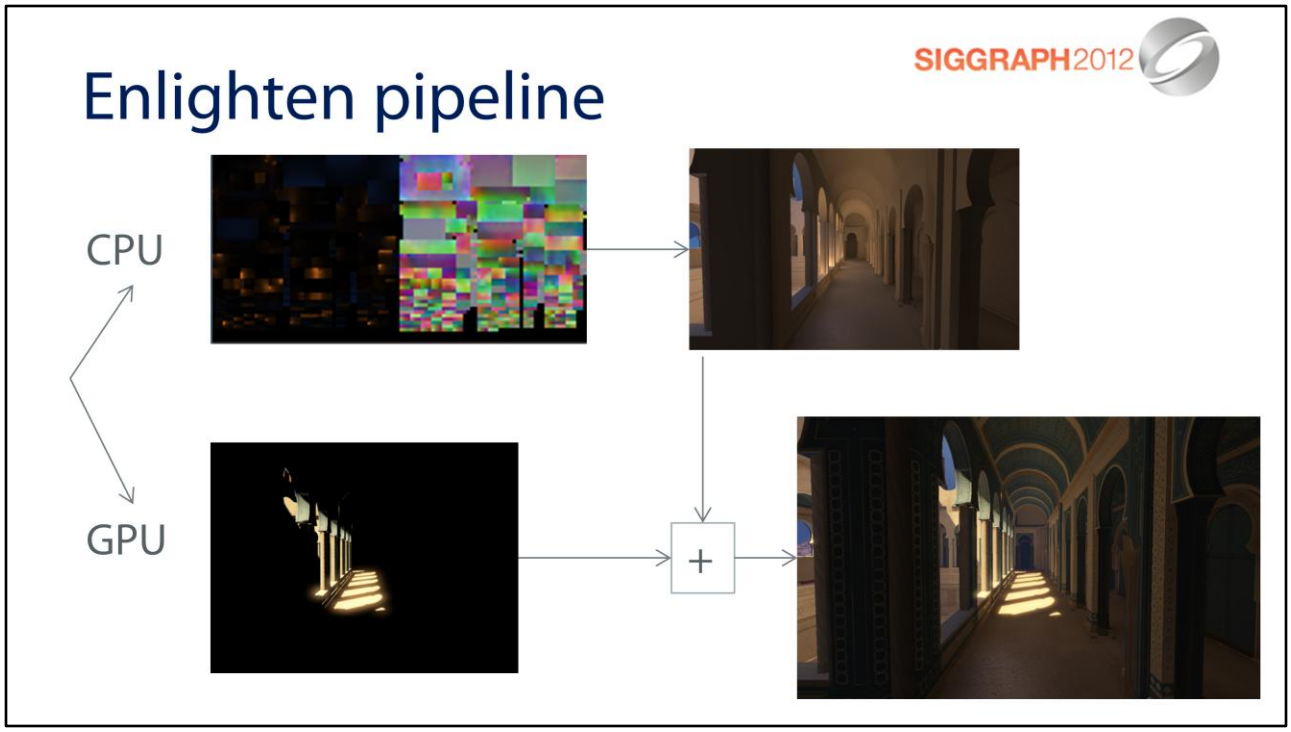
Ok, lets look at CPU lighting.

Here’s our final scene. It has both direct and indirect lighting.

Here’s the direct lighting. It’s just the sun.

And here's just the indirect lighting. This is all the bounce light from the sun, plus the blue environment light.

(And the final composite again)



Enlighten is effectively a lightmap generator. It generates lightmaps containing just the indirect lighting. The direct lighting is still handled by the GPU as usual, and simply added on top at the end. As far as the GPU is concerned, the lightmap may as well have been statically baked.

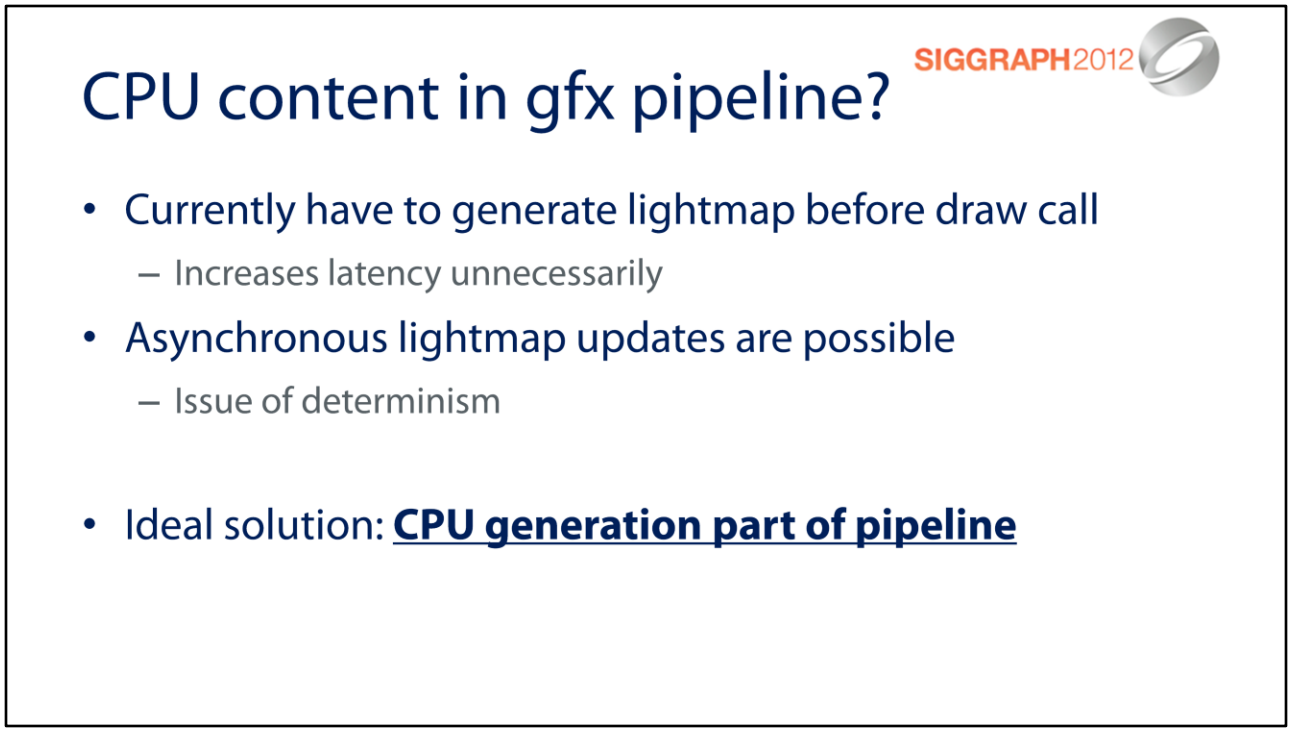
Our indirect lighting is directional, so we can relight fairly arbitrary geometry, work with normal maps, and produce an indirect specular term.

The point for this talk is that our lighting pipeline shares the work across both the CPU and GPU. The CPU is generating lighting data to be consumed by the GPU.

I like it because I think it's a great example of using the CPU to do graphics work.

Lighting is not the only thing you can generate a CPU texture for. Textures are just a means to interpolate data – in this case, over a surface. But you could just as easily generate:
- Textures in screen space
- Vertex data
-Volume textures, cubemaps,
- Maybe per-tile data? (cheaper version of screen res?)

If we had more interesting data structures, the CPU could generate much more interesting things. Int32 formats? Pointers perhaps?



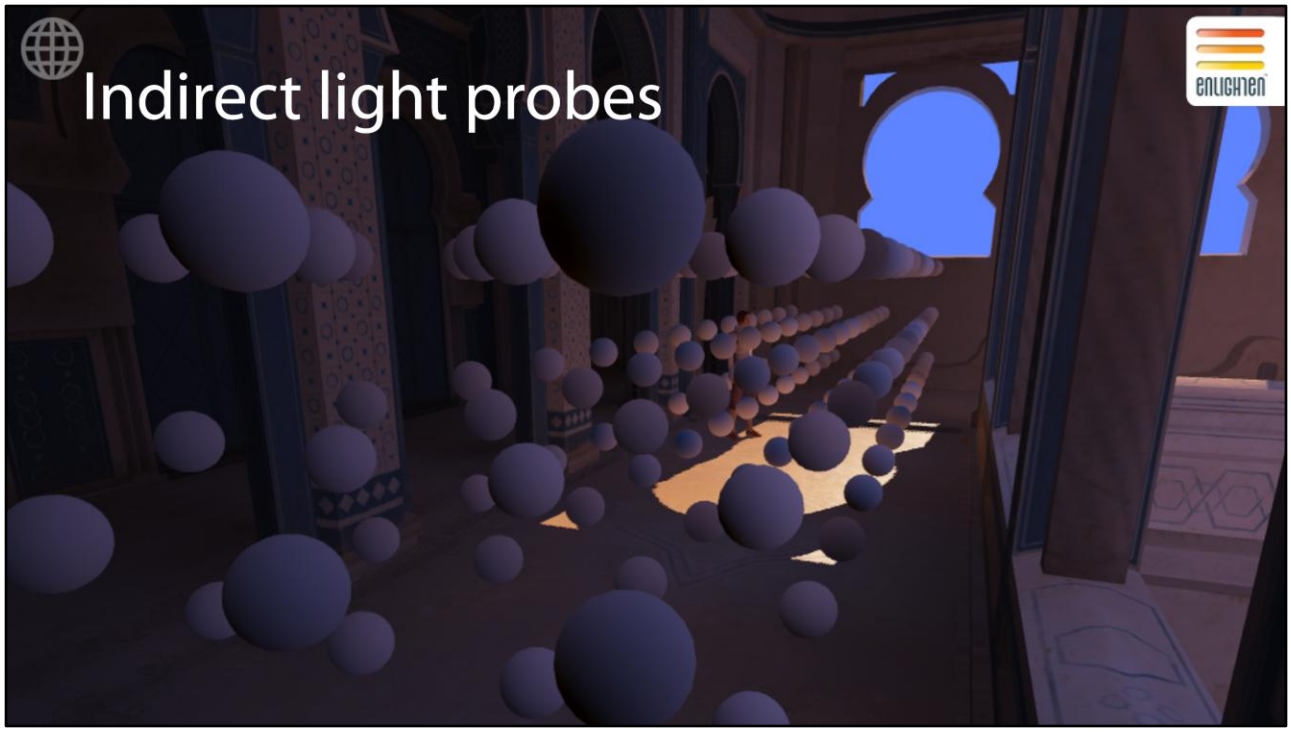
But there are annoying inefficiencies with doing work on the CPU with the current API.

There will inevitably be some overhead from the driver. If you are generating data every frame, I would expect the driver to have to copy and buffer it until it's required. And the CPU only ever sees a 'linear' data layout for textures, but it's quite likely that the GPU will used some form of swizzled format internally, implying a conversion. Lower level access would allow you to short cut some of this, but trades off usability. We haven't noticed an obvious overhead in our work so far, so I won't worry about this for now.

Latency is somewhat of a concern. With the current API, you are forced to do all CPU work up front, even though the actual results won't appear on screen for a few frames. If you have a lot of CPU work it will delay the point at which you can start submitting draw calls, and at some point you may need to do the CPU a frame ahead.

Now, on several architectures (say, for instance, Mali), after the CPU submits all the draw calls, the next frame is spent on doing all the vertex processing. It's not until the frame after that that it actually starts doing per-pixel work. So there is already a 2 frame latency, and any textures will not be accessed until the 2nd frame. We could just as well have done the CPU generation work in the frame after we submitted it.

This is really another side effect of not having the CPU being part of the graphics pipeline. It would be nice to be able to submit a CPU job as part of the pipeline. Whether or not we will ever see this, I have no idea, but it would certainly make graphics a lot more interesting.



Here's one final example of where we could be doing things more efficiently, given some additional flexibility.

As well as generating lightmaps, Enlighten also generates "light probes" to light anything that wasn't lightmapped (typically dynamic objects). These are just points in space where we can generate spherical harmonics.

We currently interpolate them on the CPU per dynamic object, and just set the resulting spherical harmonic as shader constants. This is how the girl in the image (if you can make her out) is drawn. The main limitations of this shader constant approach are we don't have a means to do more general interpolation (which is important for larger objects), and are limited to changing the probes to per draw call.

Grids are easy to interpolate, so an alternative is to put the light probes in a volume texture, but it's not particularly feasible to cover the world in a huge volume texture.

We need a means to do interpolation of more unstructured data. This kind of workload is fine on the CPU, but not really a great fit for vertex/pixel shaders. But it's also hard to do upfront on the CPU because without access to screen depth (or per tile depth) you can do more work than is necessary, and would be limited to generating lightmaps or vertex data when you really need the interpolated result in screen space. It's plausible at least, that the CPU could do a better job when in running in the graphics pipeline.

SIGGRAPH2012

CPU lighting

- Enlighten on ARM
  - Think: pre-calculated form factors
  - [Martin, Einarsson 2010]
- Indirect and low frequency direct-ish lights in lightmap
  - Environment, emissive materials, bounce lighting
  - Emissive materials => Area lights
  - Area lights => means to add more direct lights!

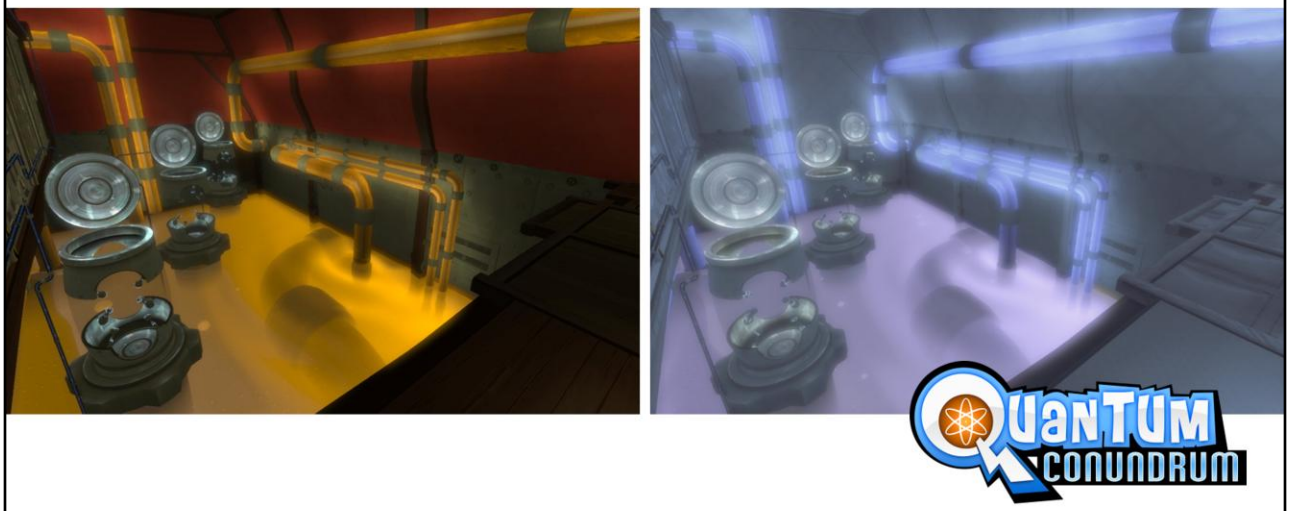
Enlighten is a *bit* like a classic form factor solution. Conceptually at least, we generate all the form factors up front with a load of ray tracing, and compress the resulting matrix down. Then at runtime we can take a vector of the incident light, and do a matrix multiply to get the bounced light. (In practice, there are details ☺).

A very helpful corollary of our approach is that we can make any surface glow and compute the result of that illumination “for free”. This allows us to add low-frequency (ie. soft) area lights using emissive surfaces, often on hidden light geometry.

So we can add soft area lights using our approach, without ever touching the GPU.

The cost of Enlighten on the CPU in comparison to the GPU cost of rendering direct lights, particularly anything with a shadow map, is tiny. So this is a big performance win.

Dynamic area lighting



This technique has actually been used in a shipping game.

Quantum Conundrum, a PS3/Xbox360/PC title that uses enlighten at runtime, does pretty much all lighting with area lights.

They have created a very soft cartoony look – quite different to the Battlefield 3 style of realistic lighting ☺

In the game they have this concept of “dimensions” – fluffy, anti-gravity, etc... You hop between them to solve puzzles, and each looks different. As you switch between them they animate the (hidden) area lights to re-light the scene appropriately.

To conclude…

The Future

- Some blips but lots of potential
- Lots of R&D to be done!
  - Efficient handling of lights
  - Shadowing
  - HDR formats, correct interpolation, texture compression
  - Mixed CPU/GPU work
  - …
- Will we see mixed CPU-GPU graphics?

To conclude, there are hurdles doing dynamic lighting on mobiles at present, but it's possible if you pick your battles, and will become widely viable very shortly.

I want to underline that mobile graphics is at a really exciting time, with lots of potential, both in terms of work to be done and the impact it can have on people. We've hugely enjoyed working on mobile GPUs, and believe they have a lot more to offer.

But I do not believe that we should assume mobile graphics simply adopts existing practices. Mobile should not blindly follow it's bigger older brethren.

We should be questioning the received knowledge of "best practices" and asking whether we should reshape them for a mobile platform.

If I have to single one out: it's the legacy of the PCI express bus. I think the opportunity for improvement here is very significant.

As I said at the start, we are excited by these future opportunities, and I hope we highlighted some of the potential to you today.

Thank you! QA?

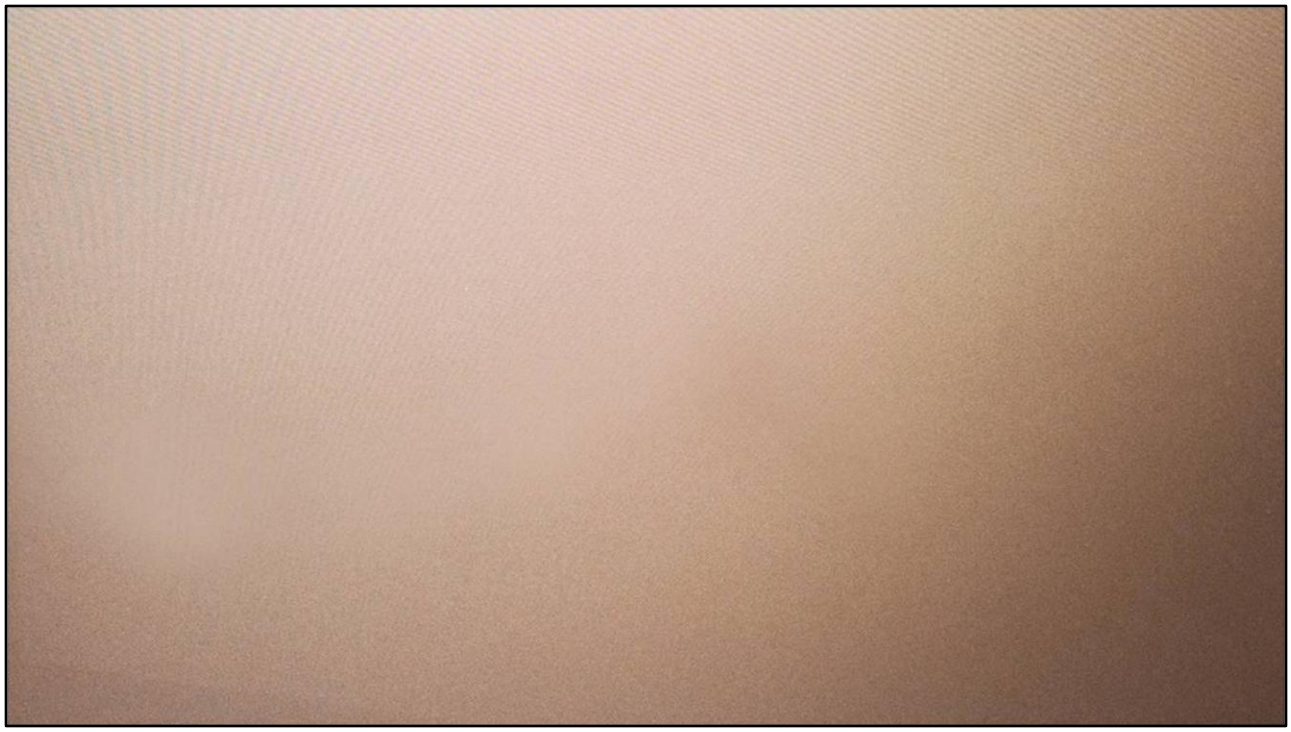
sam.martin@geomerics.com **@palgorithm**
matt.wash@geomerics.com **@maltzero**

With support of:

- Simon Jeung, "Light Pre Pass Renderer on iPhone"
  - http://simonstechblog.blogspot.co.uk/2012/03/
- Aras Pranckevičius, "Deferred Cascaded Shadow Maps"
  - http://aras-p.info/blog/2009/11/04/deferred-cascaded-shadow-maps/
- Sam Martin, Per Einarsson, "A Real Time Radiosity Architecture for Video Games"
  - Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2010, July 2010
- Andrew Lauritzen et al, "Sample Distribution Shadow Maps"
  - Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2010, July 2010
  - Asset used in the SDSM images courtesy of Valve
- And **many thanks** to all at: Imagination Technology, ARM, Broadcom, Intel, NVIDIA, Andrew Lauritzen , SIGGRAPH Mobile, and the team at Geomerics

Thanks!

And feel free to say hello or get in touch via email/twitter.



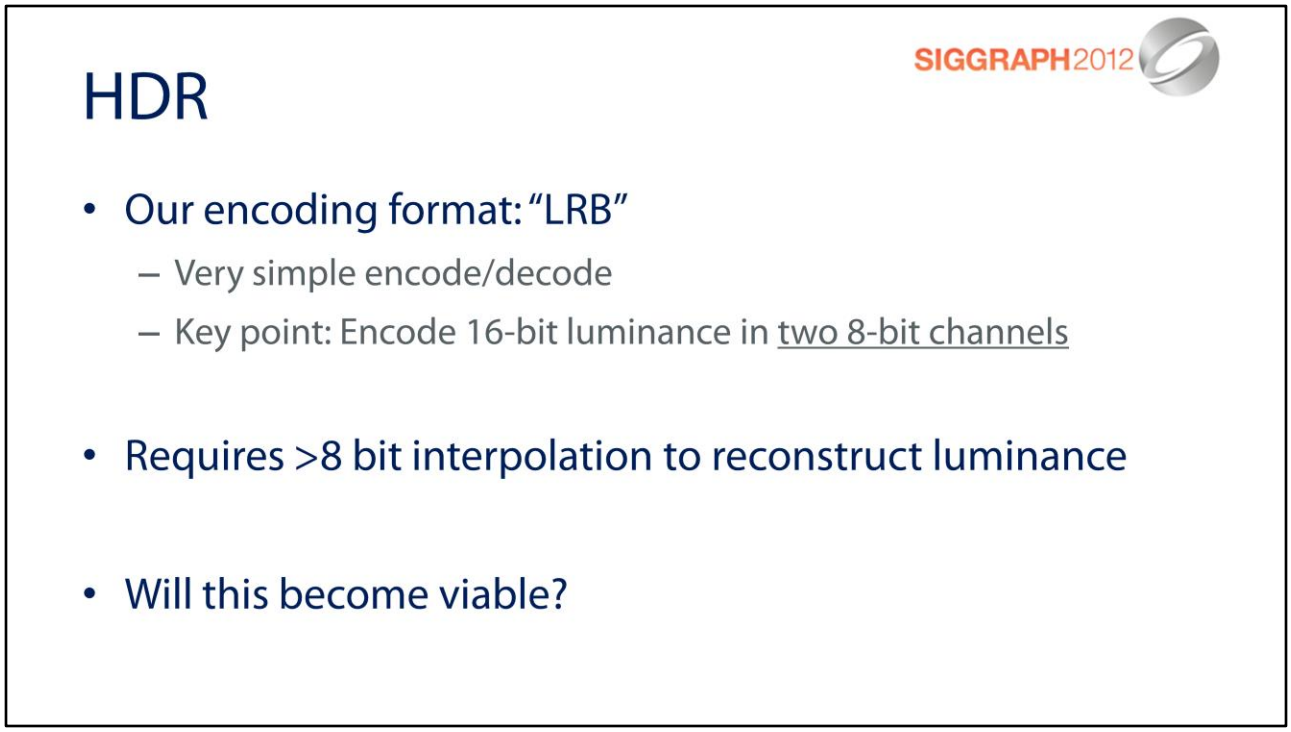
HDR

SIGGRAPH2012

- HDR is a storage problem
  - Increased range required for lightmaps
  - Half_float and half_float_linear are not that common
- Can get good results with 4-byte format
  - Variety of encodings exists
  - Need more than 8-bit precision texture interpolation
  - Precision is not part of the OpenGL ES standard

Bonus slides!

[In the actual talk, I never went through any of these]



On desktop/console GPUs the requirement for a decent interpolation precision is covered by pretty much all modern GPUs. Only the PS3 has a problem, but even that's solvable with a slight change in texture formats. Mobile is very different case where it's far less common.

From what I understand, we'd need the standards to require sufficient interpolation precision for this 16-bits-in-2-channels approach to be safe to use everywhere. Making high precision interpolation optional and having a per-texture flag would also work (much like sRGB).

Even being able to query the precision and fallback to fp16 if necessary would help, as we currently have to check device ids. But in order to fallback to fp16, you still need support for it (it's an extension in ES2) and you need to check you can linearly interpolate it (another extension). There are some devices that support fp16, but not linear filtering of it, so you have to support all possible combinations which has some unfortunate shader fallout.

On the other hand, whether it's better to use fp16 or a 4-byte encoding in the long run is not completely clear to me. The encoding does add additional shader instructions, and the additional interpolation precision presumably adds extra gates to the hardware. It's also less flexible that fp16 and requires a 'max luminance' to be specified. But it is half the size, and is therefore more memory bandwidth friendly.

[I did have a slide which had a table of what HDR approach works on which hardware, but I've cut it out to protect the guilty ☺. Drop me a line if you are interested]

This is what happens when the hardware doesn’t interpolate textures with sufficient precision!

Some of you may be thinking that 2k squared does sound a bit large.

The issue is that shadow maps, particularly ones for the sun, do not really take very well to scaling down.

You *can* drop resolution, but even 2k is only on the edge of being ok.
Taking multiple samples to try and soften edges can go some way, but appears to be more expensive than using a higher resolution shadow map

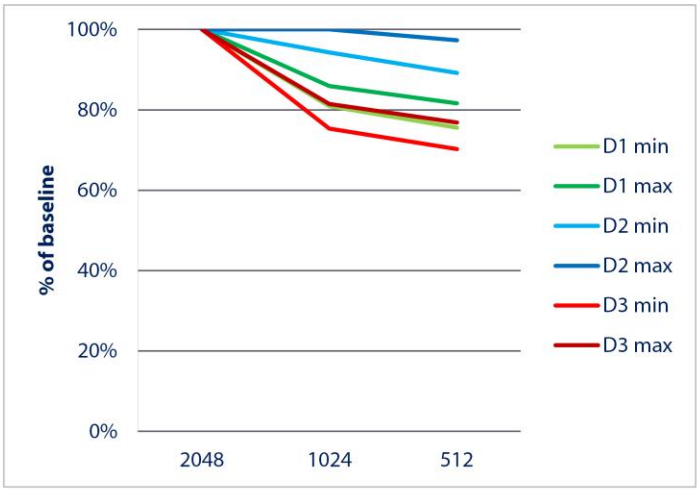
(And by 2k I mean the total texture size, not the size of the individual cascades)
For most devices that can do CSMs 2 cascades appears to be a sweetspot, with 3 significantly more expensive than the first 2.

Surprisingly resolution is not as big a deal as it may first sound. On several top end mobile devices, 2k is not amazingly slower than 512 (ie. 10%).
But then again, some have a bit of a cliff at 1024, and other older GPUs scale more steeply. So mileage varies, but in general, resolution is not as bad as it might sound.

Shadow resolution

SIGGRAPH2012

- Resolution not hugely dominant factor
- 2 cascades ok for most devices
- PCF / multi-taps are expensive



[Caveat: I would take the results here with a pinch of salt. The basic point that generating large resolution is not always as bad as it may sound. But I wouldn't read into the graph too much – 3 devices is not a big enough sample. I didn't use this slide in the end because it wasn't ready, and didn't add much. I've left it in the bonus slides because I think it's interesting to note that at least 1 device (D2) *really* doesn't care that much about resolution ☺]

Use mixed resolution?

SIGGRAPH2012

1. Render shadow factor half-res
2. Upscale and composite in second pass.

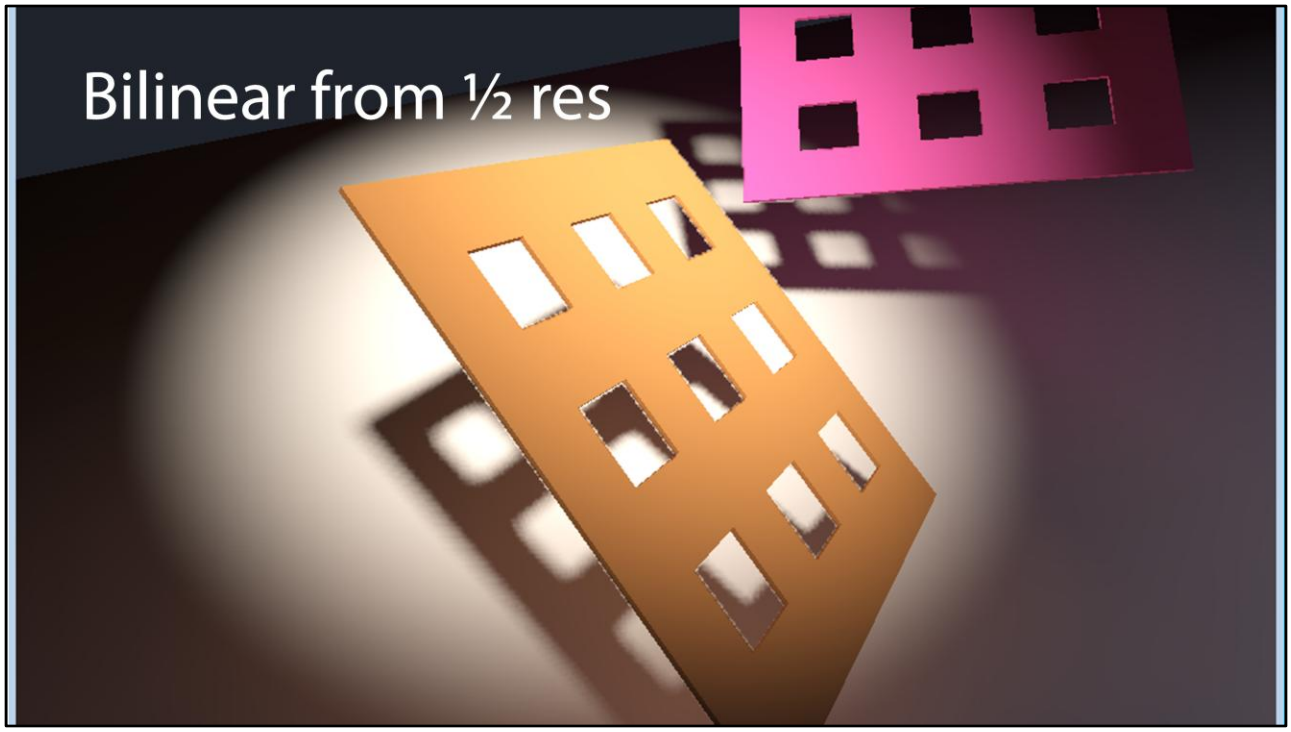
- Get more for less!
  - Although edge aware upsample not exactly free ☺
- On our "to try" list

Another weapon in the battle for decent efficient shadows may be as simple as upscaling.

Mixed resolution rendering, and clever sampling/reconstruction is all the rage at the moment. Perhaps this is something that should get more wide adoption on mobile?

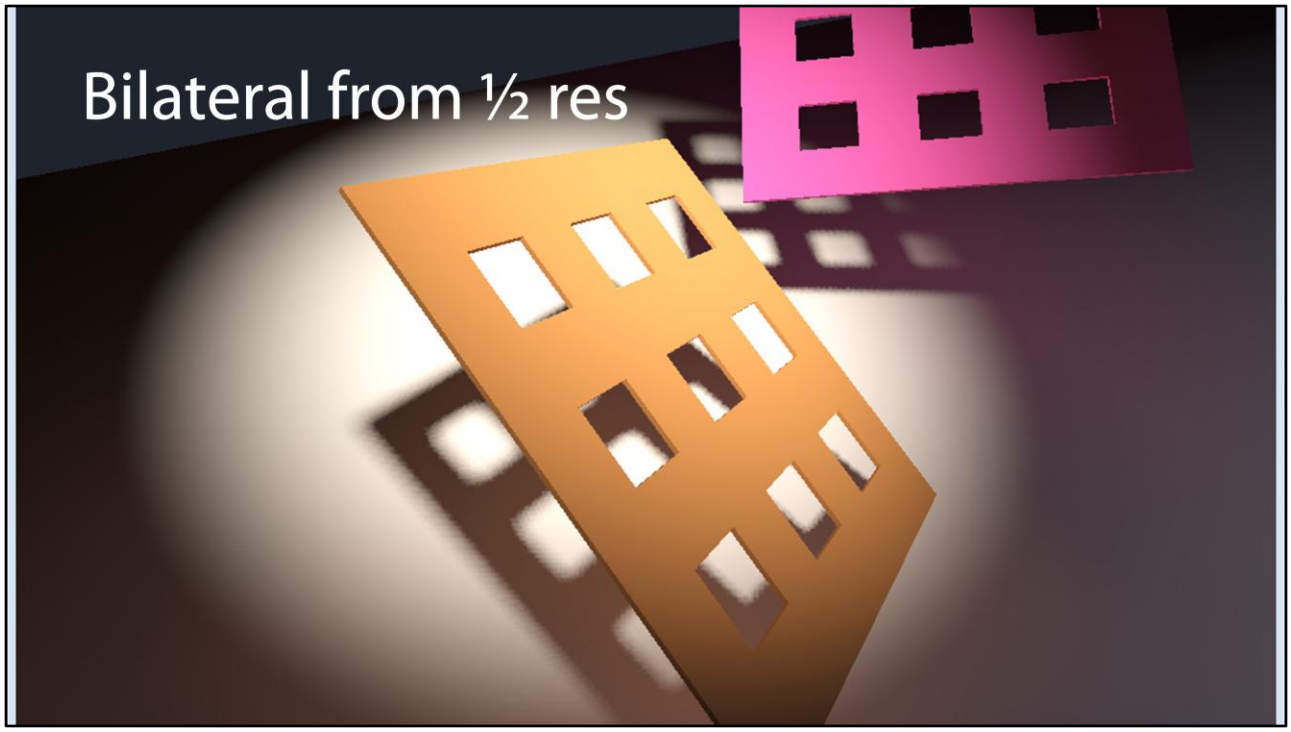
Basic idea here is to generate the shadow in screen space, at a lower res, possibly blur (SSSMs), and upscale (maybe at the same time).
Note that for most 3D game/graphics content, you want to run a "sensible" per-pixel res and use hardware upsampling where possible. Retina resolutions are overkill.
Prefer smoothness/alias-free images to high res.

Really simple approach works rather well here. In particular, shadows are a great case for upsampling.



Bilateral filters are not rocket science anymore, but here's a quick example of how useful they can be.

Here, the shadows have been computed at ½ screen res. If we naively upsampling, using good old fashioned bilinear filtering, you get a load of edge issues...



But if you simply weight the samples by a depth heuristic, you can pretty much fix all the issue. We've found it to be remarkably robust.

The trade off here is extra aliasing – you are computing less information so if your shadow content is particularly high frequency you will get flickering as you zoom out.
You alleviate this to some degree with a bit of blurring, or a larger filter.

In a different project, I've found ½ res upscaling to be fine, but you can't really go beyond that without aliasing and blocky shadows getting you.