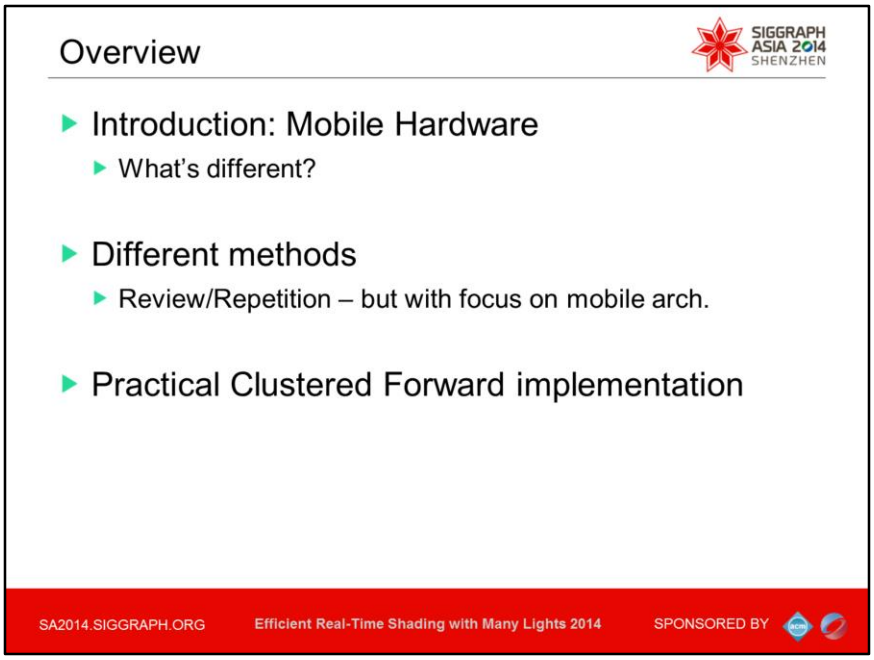
Many Light Rendering on Mobile Hardware

Part 4/4

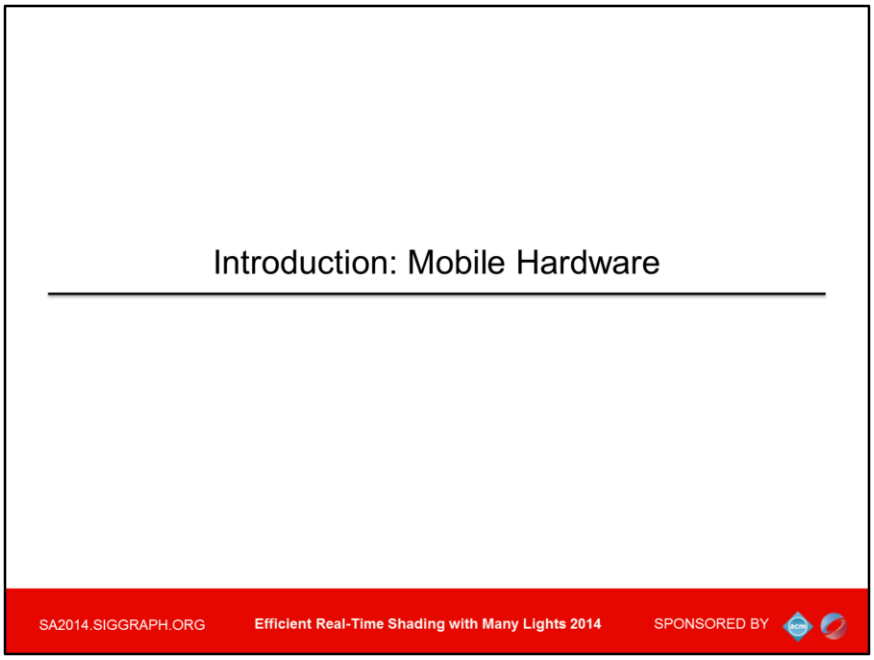
Presenter: Markus Billeter
- Visualization and Multimedia Lab, University of Zürich
- Chalmers University of Technology



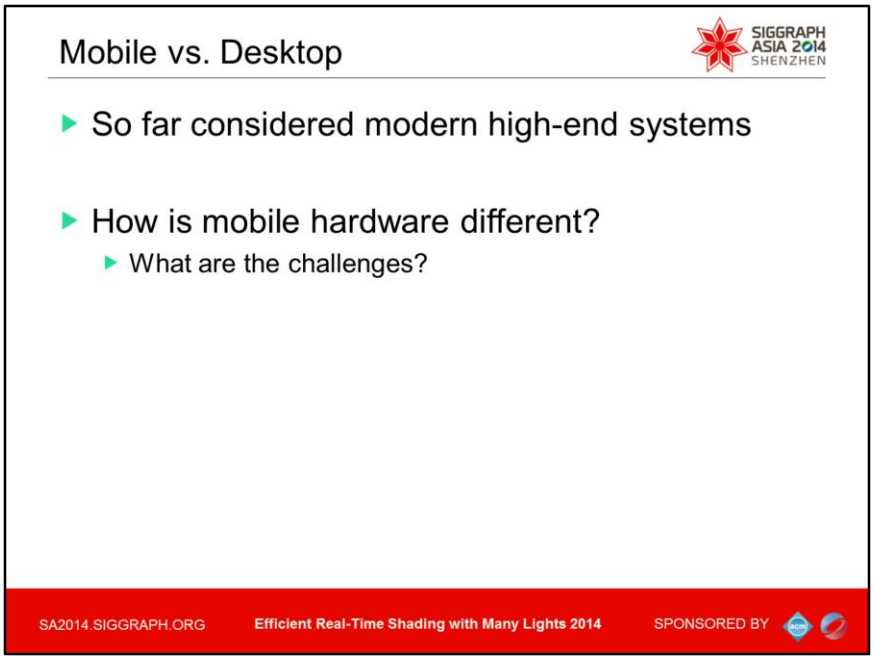
The presentation consists of three parts. First, I'm going to talk about mobile hardware in general, and provide some motivation as to why it deserves special considerations when picking a many-light rendering method.

After that, I'm going to revisit the different many-light rendering methods. Besides repetition from earlier parts of the course, there will be a focus on the method's properties with respect to mobile devices. Additionally, I'm going to present two new methods that were designed specifically with mobile GPUs in mind.

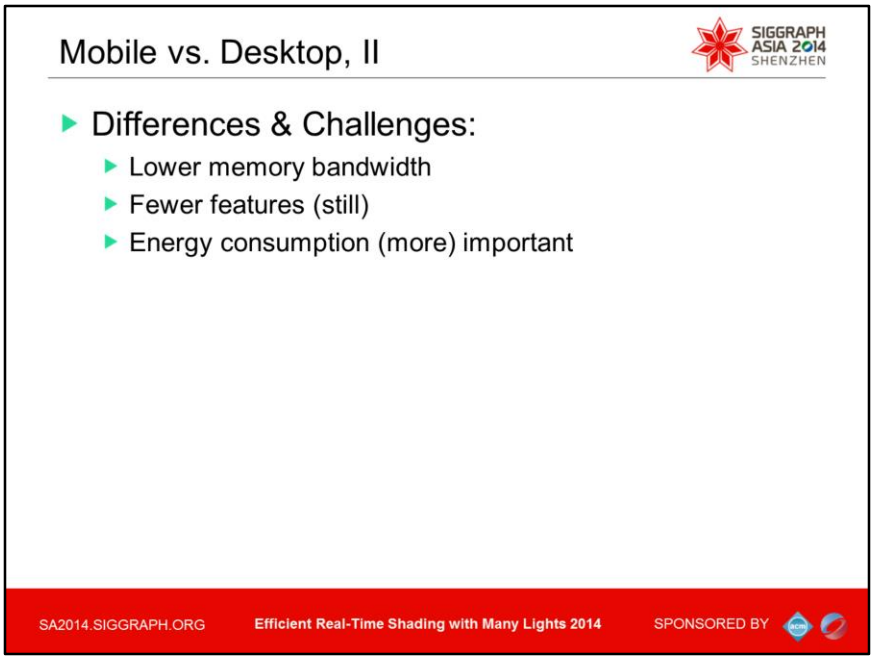
Finally, the last part provides some details about an implementation of many-light rendering on mobile devices, specifically some Android-class hardware..



Let’s get started with the introduction to mobile hardware.



So far, this course has mainly considered high-end systems – i.e., desktop-class GPUs and perhaps consoles (Emil's part). Now that we're looking at mobile hardware, we need to find out how it differs and what kind of challenges it poses.



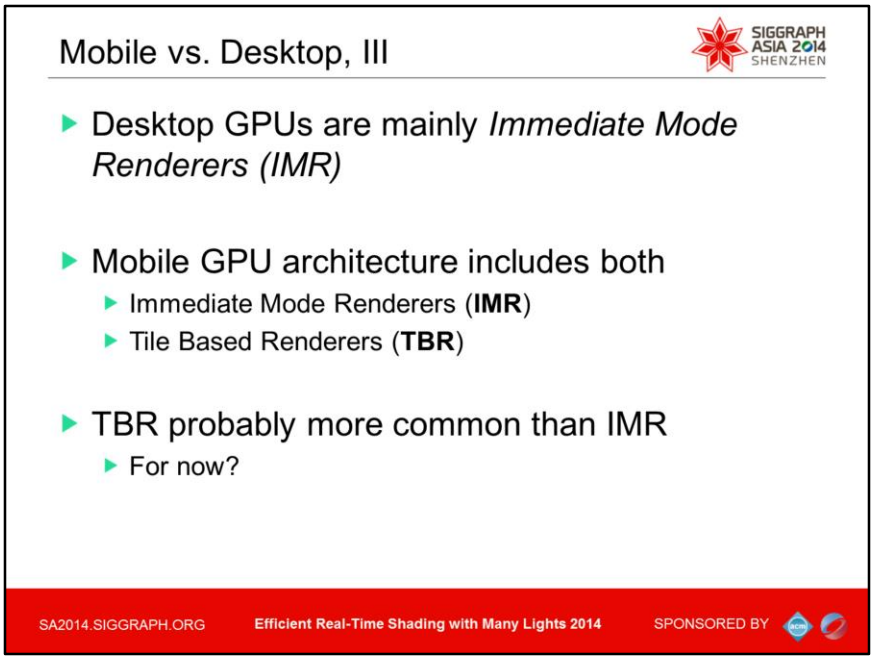
One of the main differences is that most mobile hardware will have much lower memory bandwidths when compared to desktop systems. Most devices share RAM between the GPU and CPU, so memory bandwidth will also be shared – unlike desktop GPUs which have separate VRAM.

For now, mobile GPUs still have fewer features, when compared to modern desktop GPUs.

When talking about mobile hardware, it's difficult to not mention energy consumption. If we can do something to reduce energy consumption on the software side of things, it's definitively worth considering. Not only will that improve battery life, but it might also allow the device to run cooler, and thereby avoid forced underclocking due to thermal limits. On the other hand, energy consumption is probably one of the more difficult items to deal with from a software perspective.

Some of this is improving. For example, features that are well-known on the desktop side of things are being added to mobile APIs, as newer versions become available.

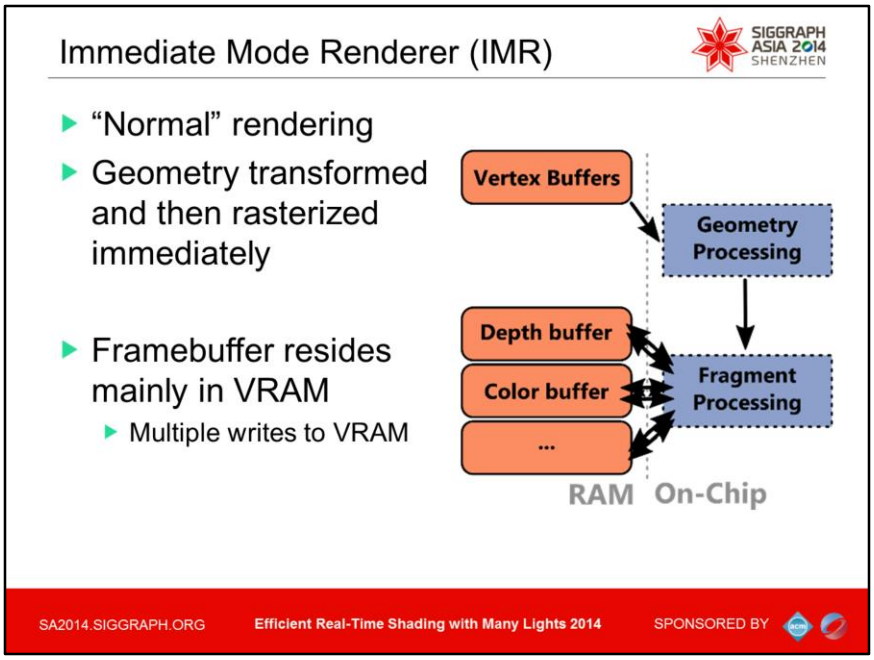
However, if you're targeting devices that are currently out there – i.e., what you might have in your pocket right now – you will still notice some of the limitations.



Another difference lies in the architectures that are commonly available.

Desktop GPUs are mainly *Immediate Mode Rendering* architectures, or *IMR* for short. On the other hand, in the mobile space both immediate mode renderers and *tile based renderers*, or *TBR*, are common. For now, *TBR* is probably the dominating architecture among mobile GPUs.

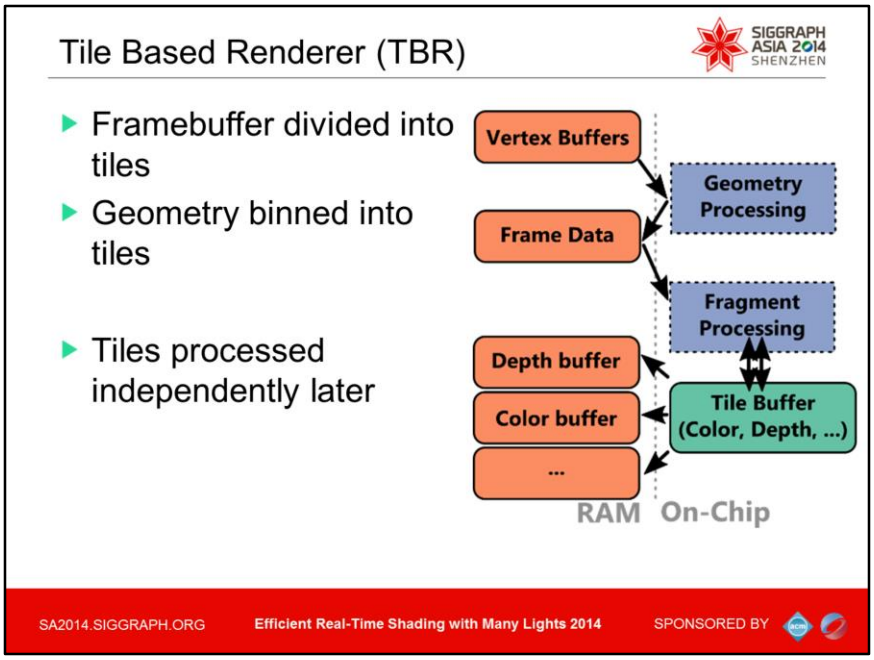
Let's quickly look at the differences between these architectures.



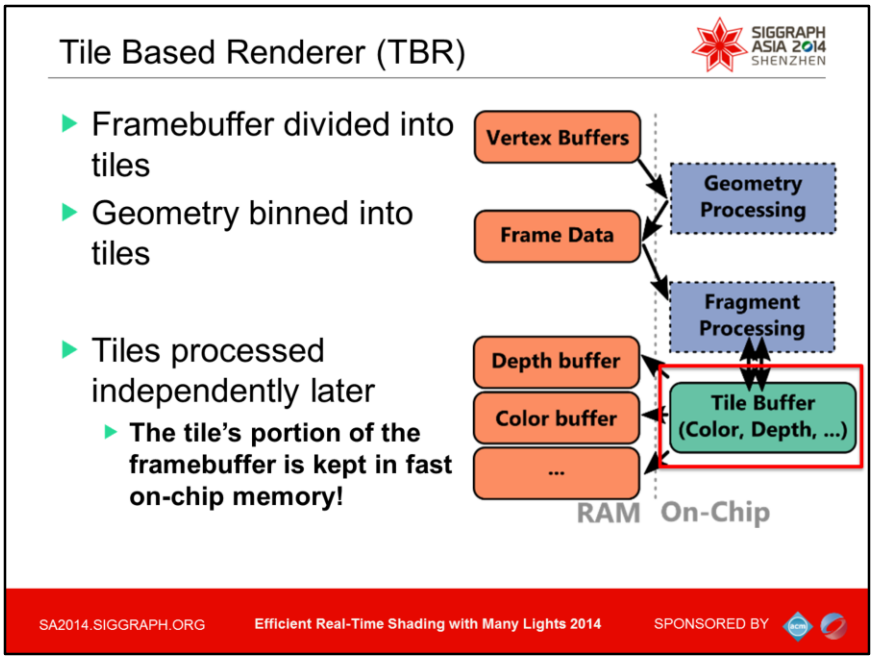
The IMR architecture is what I'd consider "normal rendering", i.e., what's typically illustrated when looking at something like the traditional OpenGL rendering pipeline.

Here, the geometry is stored in (V)RAM, and submitted to the GPU in batches. The geometry is then transformed by the geometry processing stage (vertex shaders etc.), and then immediately sent to the rasterization and shading via on-chip queues. The shading outputs the framebuffer data, which is merged into the framebuffer that typically resides in VRAM in its entirety.

The VRAM may be written to multiple times when there's overdraw.

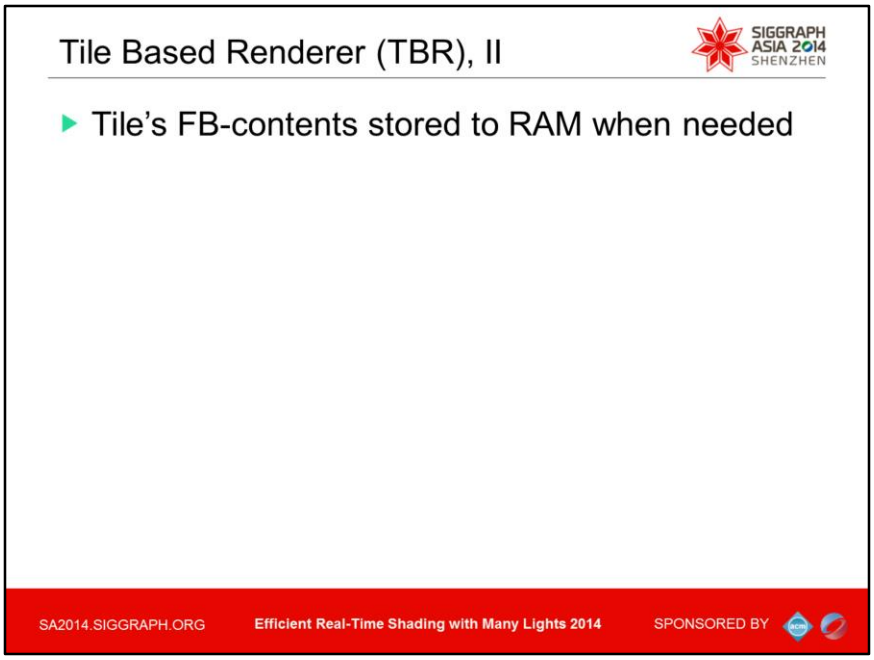


A tile-based renderer looks slightly different. It gets its name from the fact that the frame buffer/screen is subdivided into many tiles. When the application submits geometry, it's transformed as normal. But instead of being rasterized *immediately*, the transformed geometry is binned into. Later – for instance after all geometry has been submitted – each tile is processed independently.

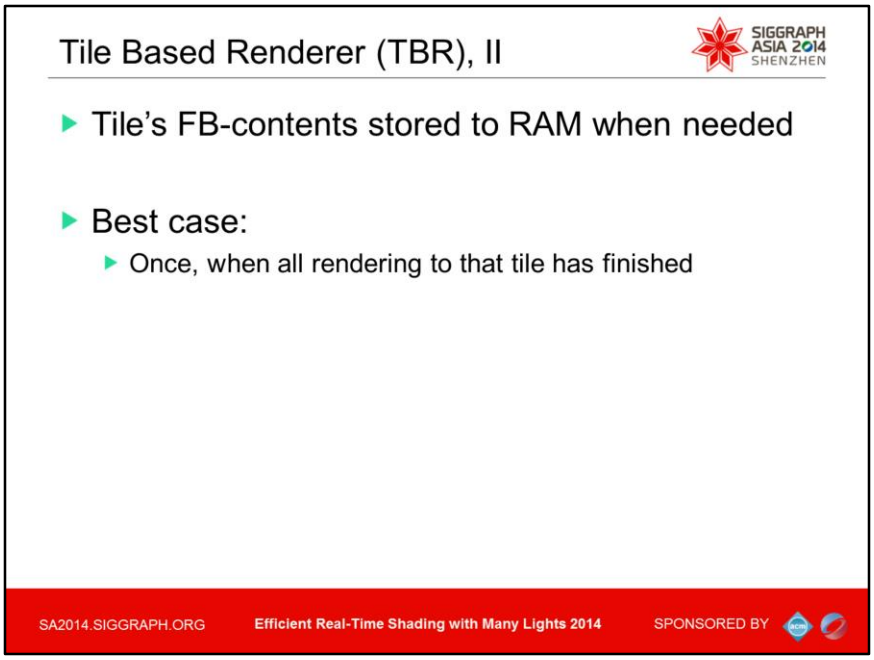


A tile-based renderer looks slightly different. It gets its name from the fact that the frame buffer/screen is subdivided into many tiles. When the application submits geometry, it's transformed as normal. But instead of being rasterized immediately, the transformed geometry is binned into. Later – for instance after all geometry has been submitted – each tile is processed independently.

At that point, the geometry associated with the tile is rasterized and shaded. The trick here is that the tile's portion of the framebuffer is kept in local on-chip memory, which avoids expensive writes to RAM.

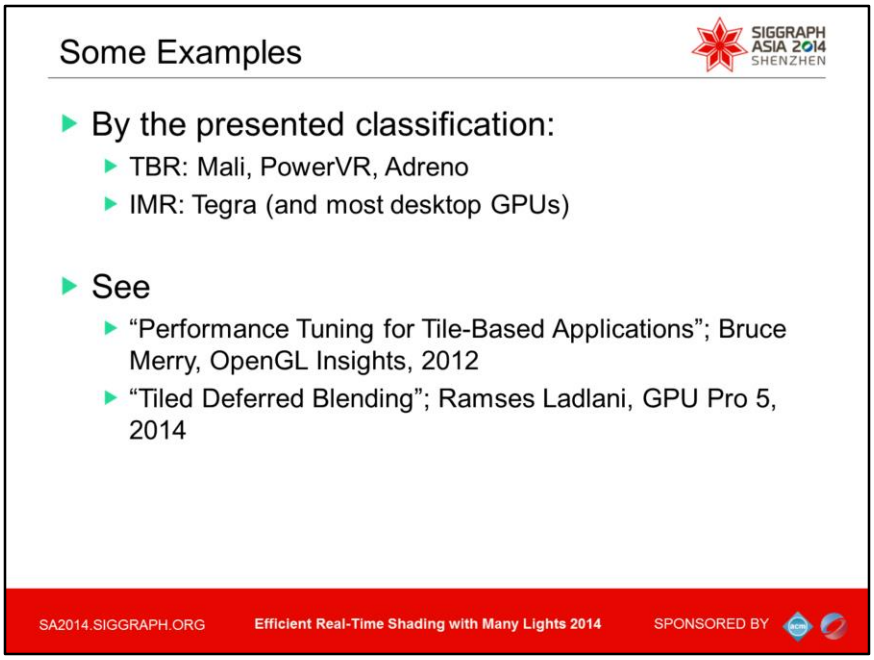


The tile's framebuffer contents are stored to RAM only when needed.



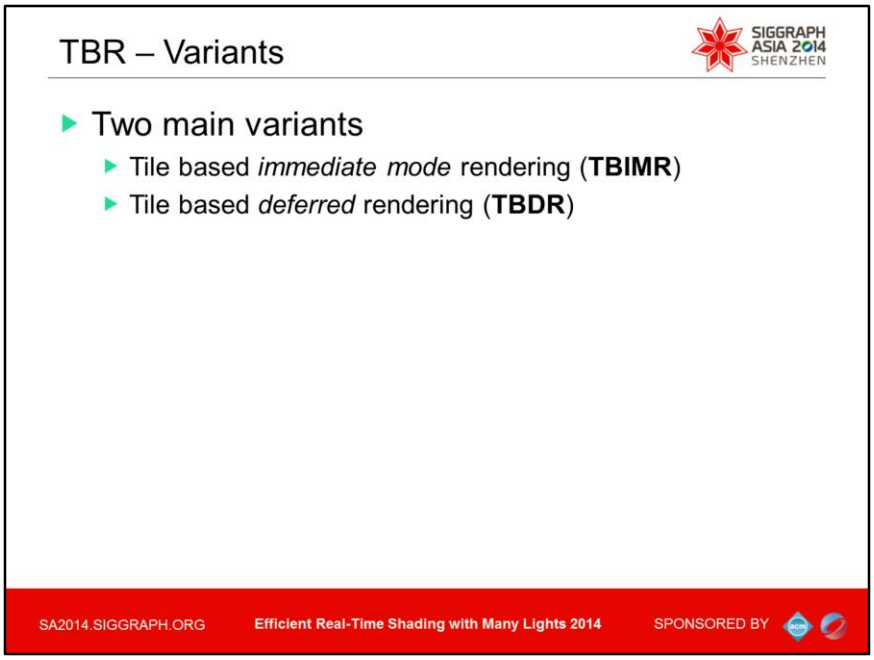
The tile's framebuffer contents are stored to RAM only when needed.

In the best case, that's once, at the very end of each frame, when all the rendering for that frame and tile has finished.

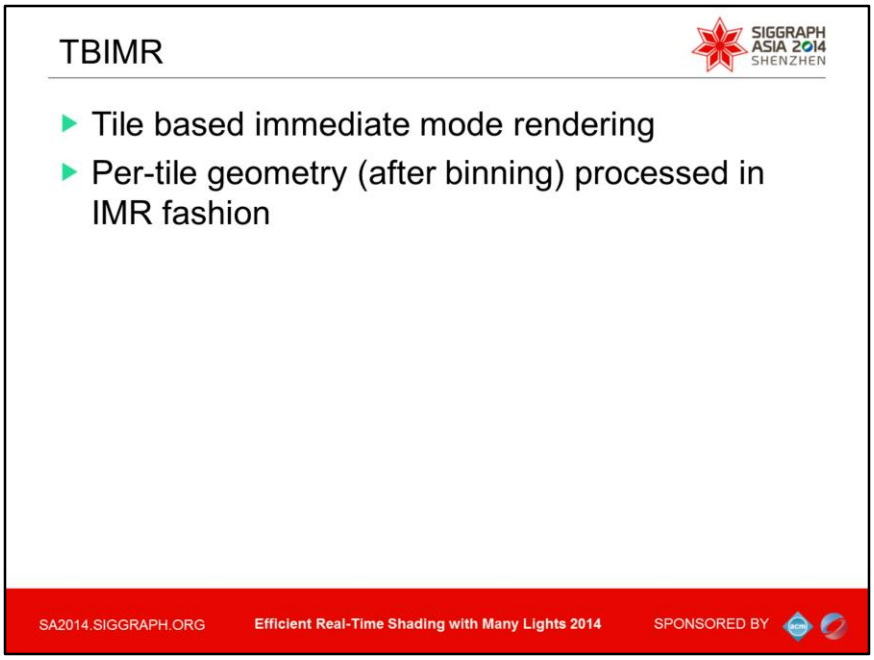


By the presented classification, the ARM Mali, the Imagination Technologies PowerVR and the Qualcomm Adreno GPUs are all tile based renderers. The only mobile chip that I know of in the mobile space that is IMR is the NVIDIA Tegra. Of course, most desktop GPUs are IMR.

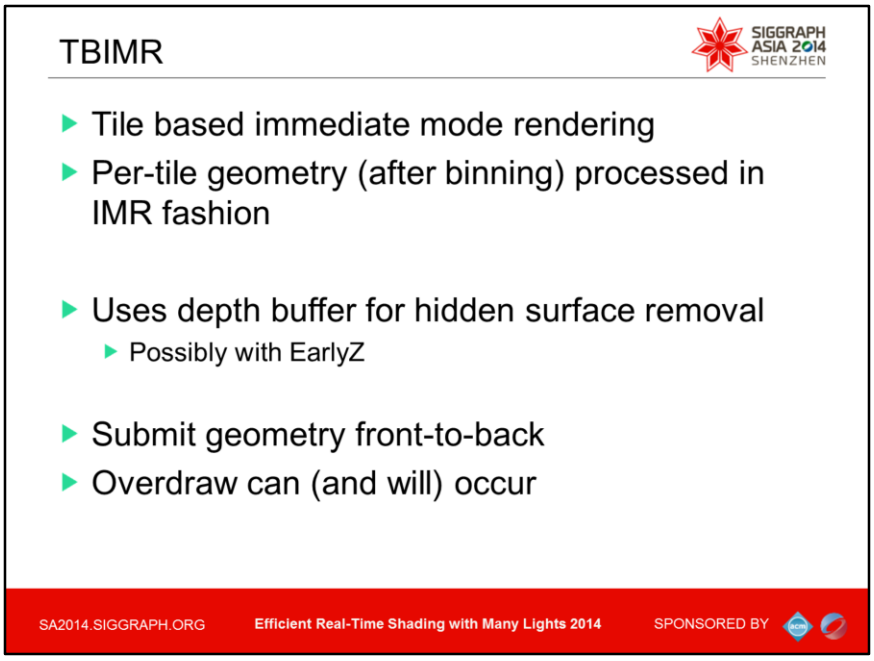
For more in-depth information regarding tile based renderers, and their characteristics, see the "Performance Tuning for Tile-Based Applications"-article. (References will be repeated at the end of the talk.)



Tile-based renderers additionally come in two variations. There's tile-based immediate mode renderers (or TBIMR), and there's tile-based deferred renders (TBDR).

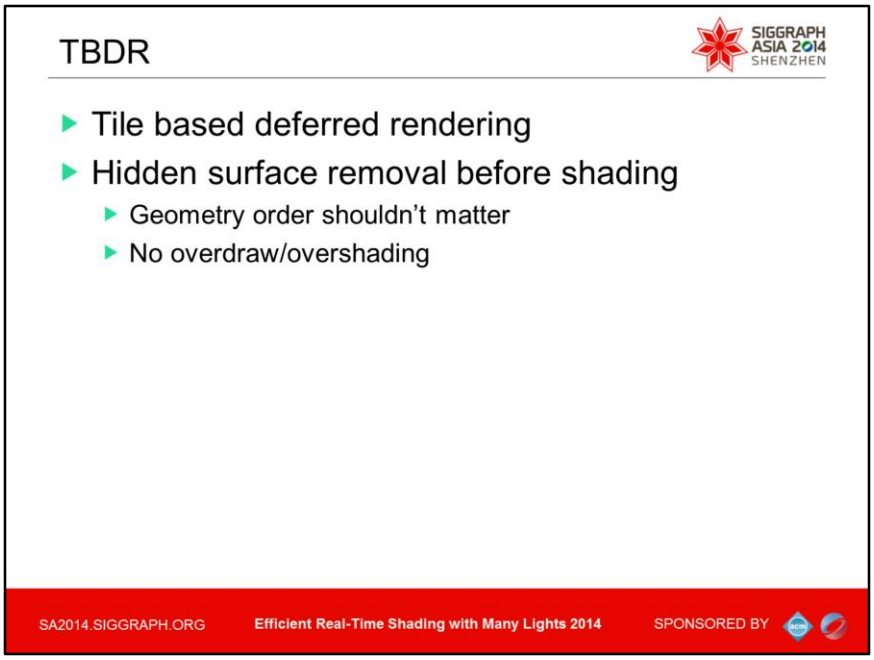


In TBIMR, the per-tile geometry is processed in an IMR fashion.



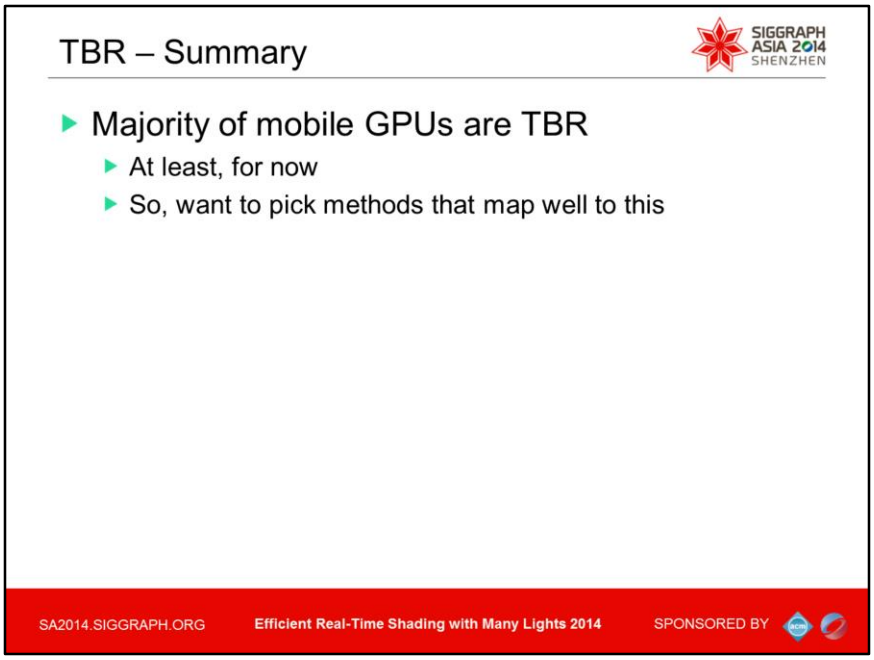
In TBIMR, the per-tile geometry is processed in an IMR fashion.

I.e., the geometry is rasterized and shaded in the order it's submitted through the API, and a Z-buffer equivalent is used to perform hidden-surface removal, possibly with typical IMR optimizations such as EarlyZ. An observation is that overdraw and overshading is possible, and that it's still useful to submit geometry front-to-back, for instance.

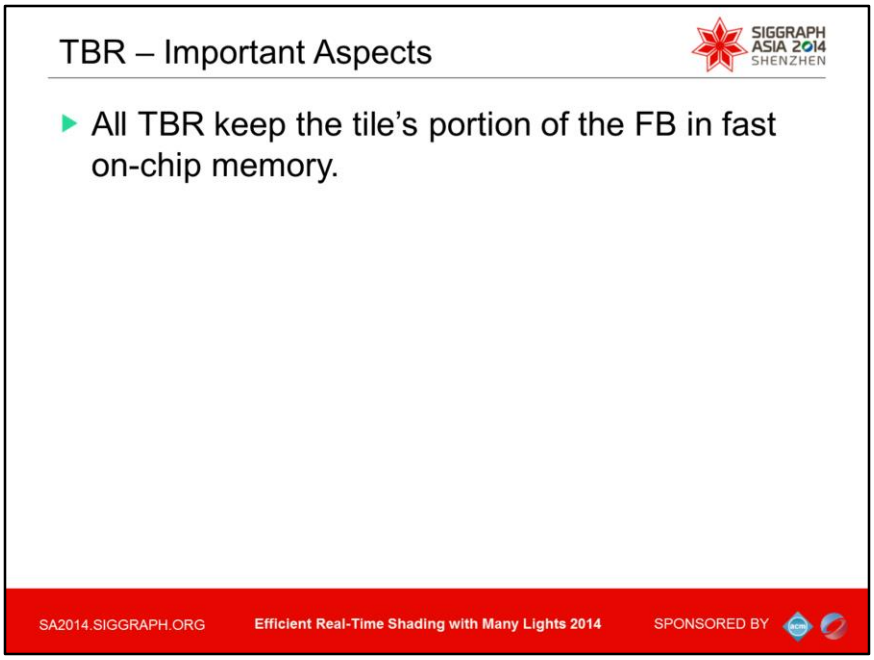


In TBDR the hidden surface removal is done before shading instead.

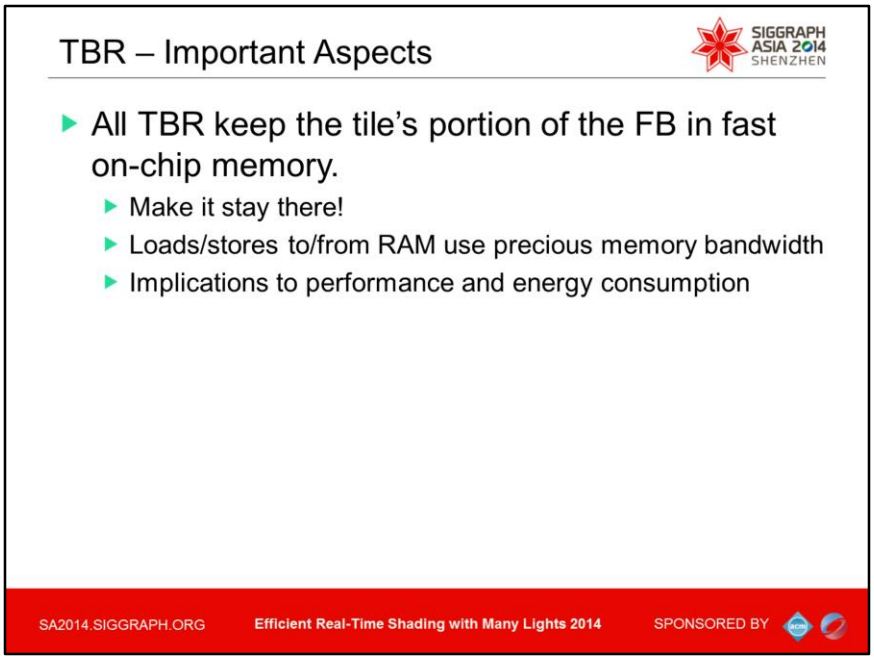
Therefore, the geometry submission order shouldn't really matter, and also there shouldn't be any overshading.



Just to quickly summarize the architecture related portion of this part: the majority of mobile GPUs are TBR, at least for now. So when we're looking at different methods, we'd like to pick a method that performs well on the TBR architectures.

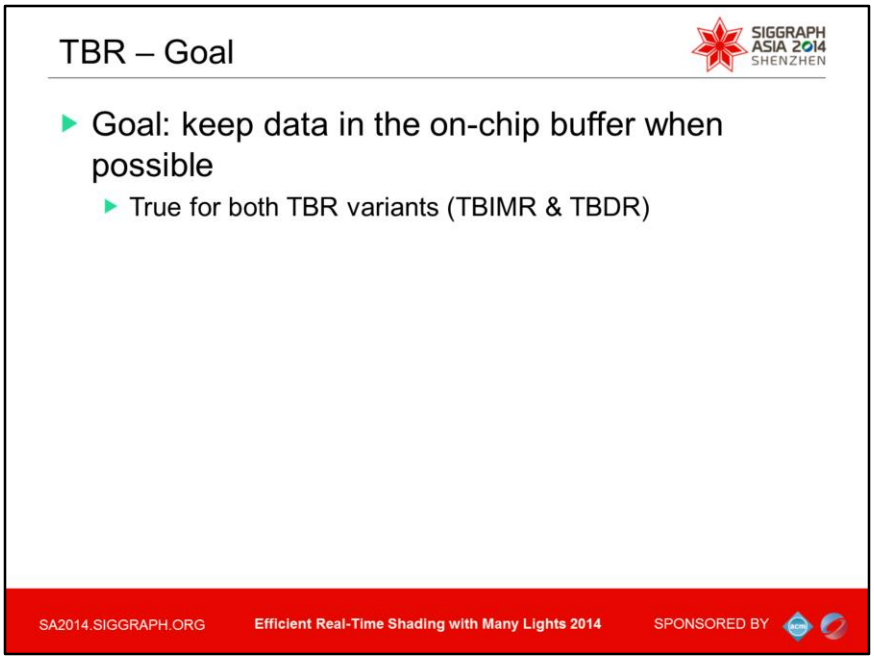


The key feature of TBR platforms is, that the tile's portion of the frame buffer is kept in fast on-chip memory.

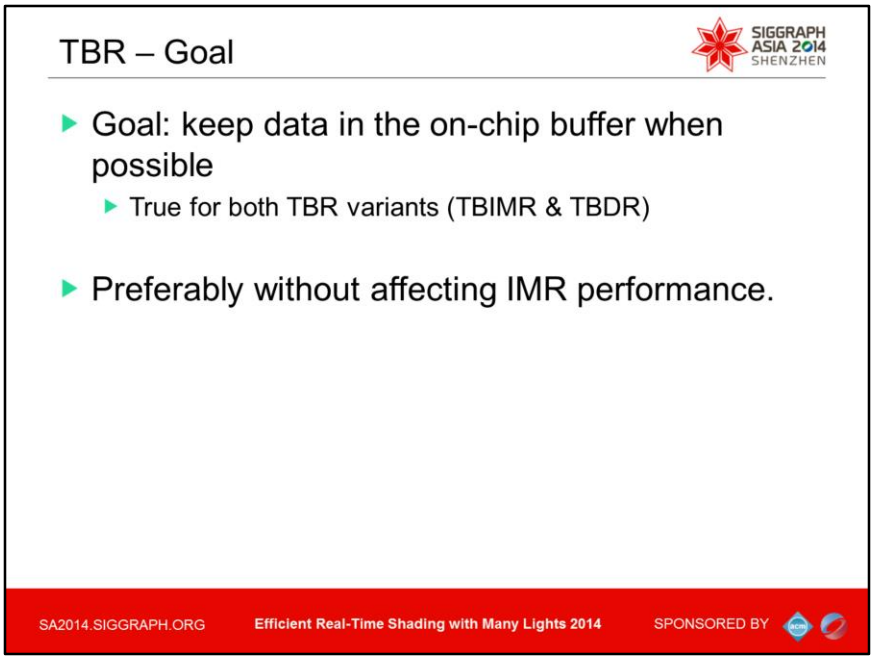


The key feature of TBR platforms is, that the tile's portion of the frame buffer is kept in fast on-chip memory.

We want to make it stay there, since storing to, and loading from, RAM is expensive in terms of memory bandwidth, which may further affect performance and energy consumption.

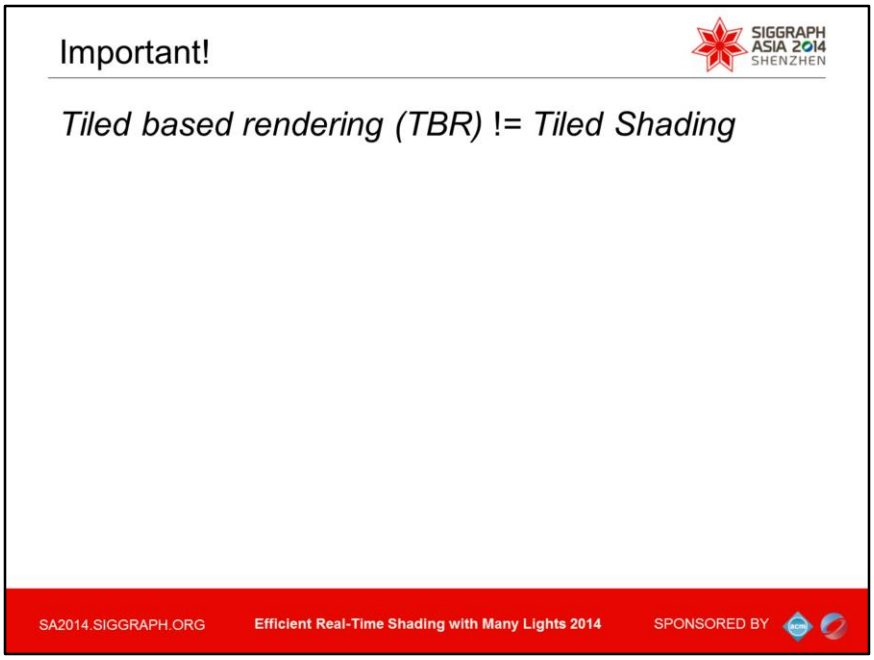


So, really, our goal is the keep the framebuffer data in the on-chip buffers for as much as possible. This is true for both TBR variants, i.e., for both TBIMR and TBDR.

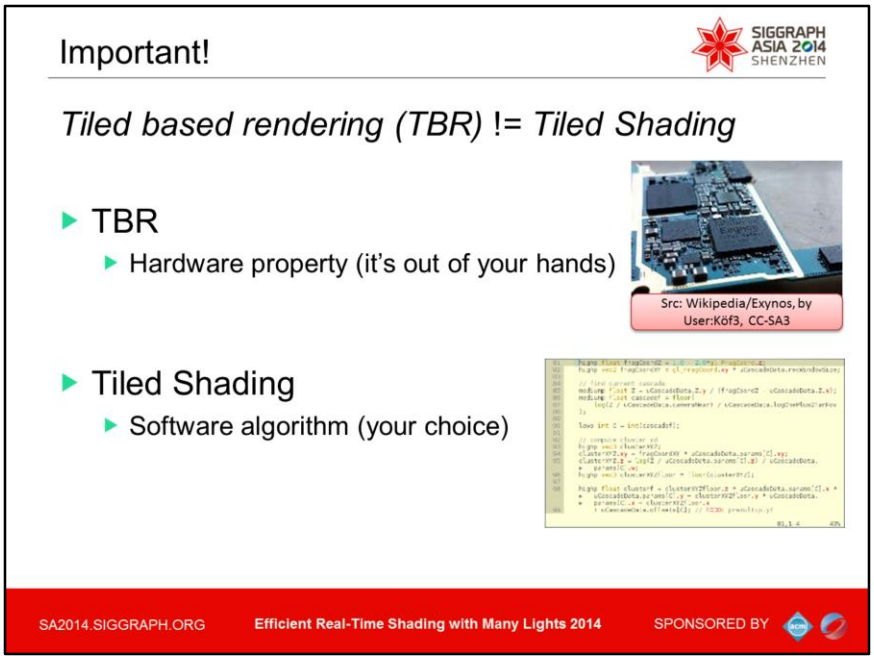


So, really, our goal is the keep the framebuffer data in the on-chip buffers for as much as possible. This is true for both TBR variants, i.e., for both TBIMR and TBDR.

And, we'd like to do this without affecting IMR performance negatively.



An important note at this point is that tile-based rendering (TBR), which I've been talking about so far, is not the same as tiled shading, which you've heard about in the earlier parts of the course.

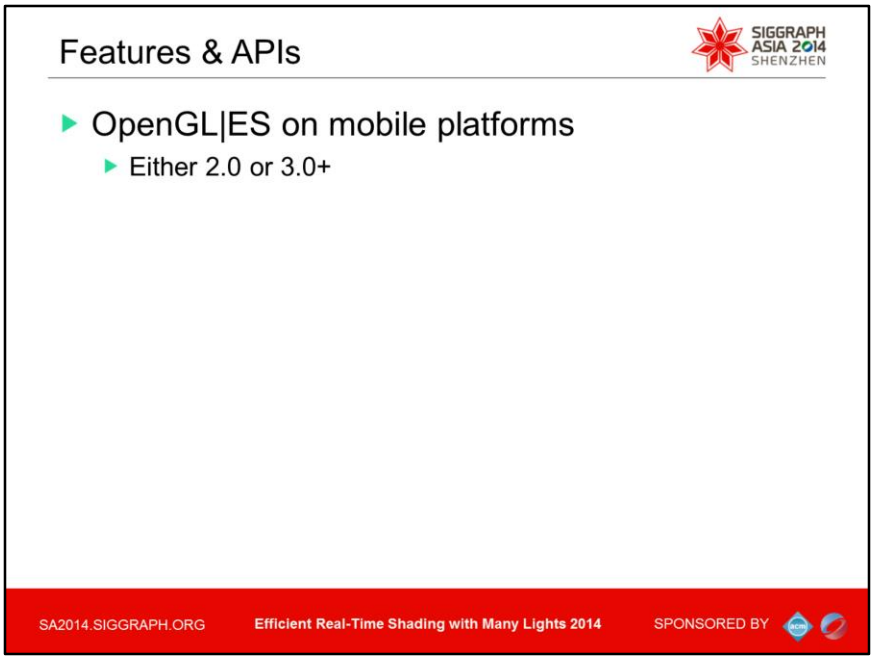


An important note at this point is that tile-based rendering (TBR), which I've been talking about so far, is not the same as tiled shading, which you've heard about in the earlier parts of the course.

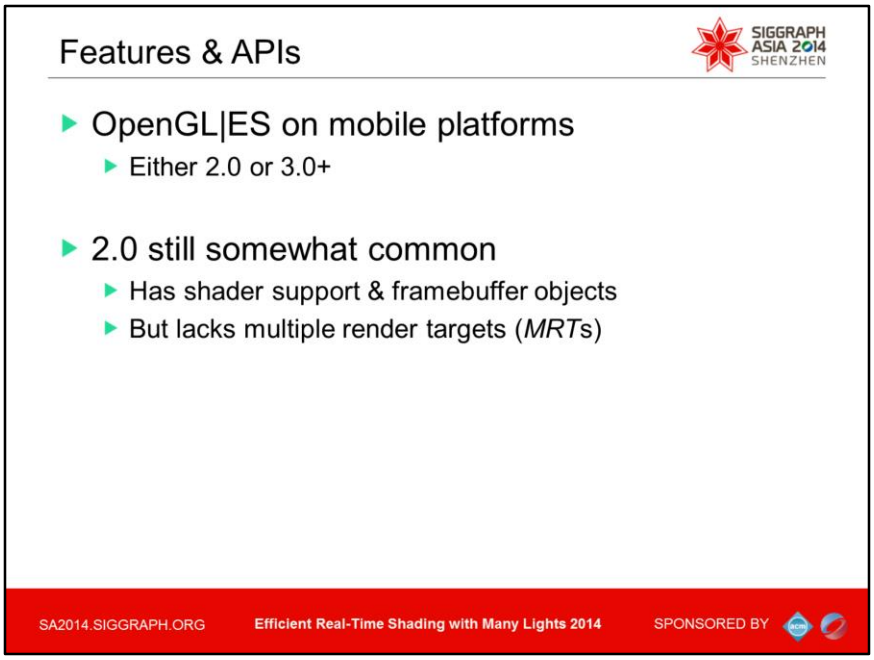
TBR is a hardware property, so it's largely out of your hands – unless you, for example, refuse to run on certain platforms.

Tiled shading, on the other hand, is a software algorithm, so you pretty much get to choose whether or not to implement it.

It's also perfectly valid to use Tiled Shading on a TBR platform, as we shall see later.

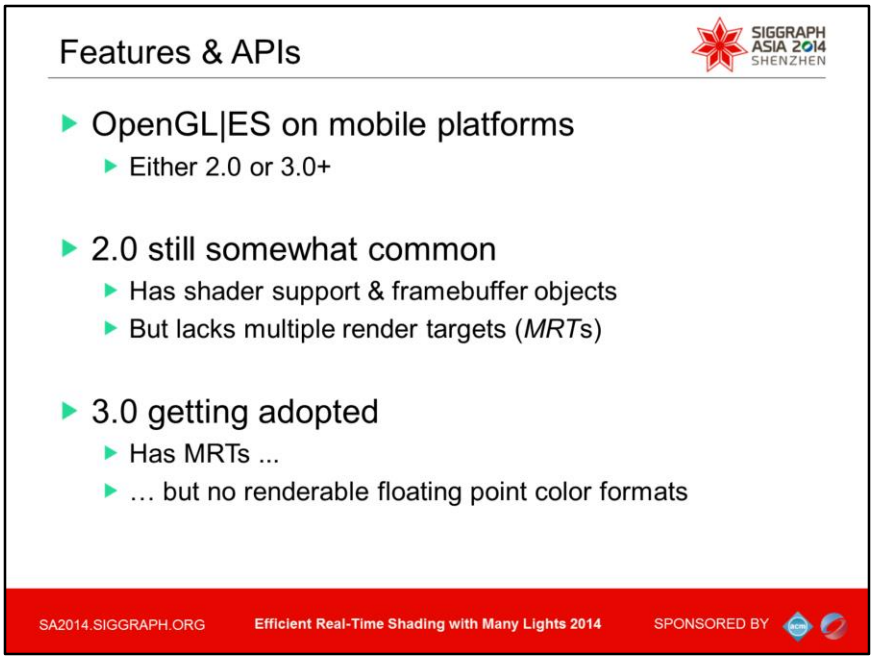


Another difference that I mentioned was in terms of features available on mobile platforms. So, if you're developing for a mobile platform, you'll likely end up using OpenGL|ES 2.0 or 3.something.



Another difference that I mentioned was in terms of features available on mobile platforms. So, if you're developing for a mobile platform, you'll likely end up using OpenGL|ES 2.0 or 3.something.

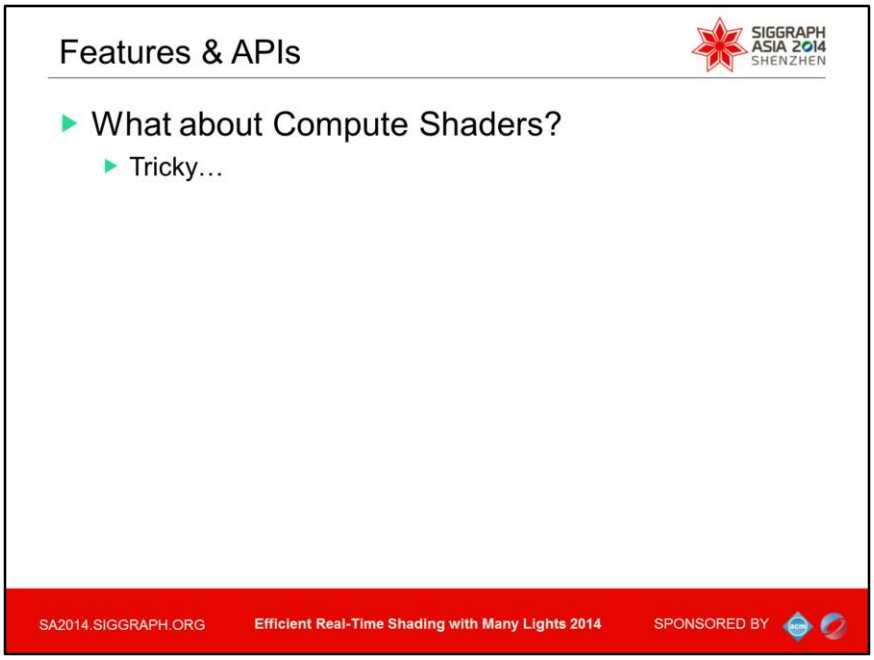
For now, I'd say that 2.0 is still somewhat common, at least if you're targeting devices that are out there right now. For us, the most interesting properties of OpenGL|ES 2.0 is its support for shaders and framebuffer objects. However, at least in the core spec, there's no mandatory support for multiple render targets.



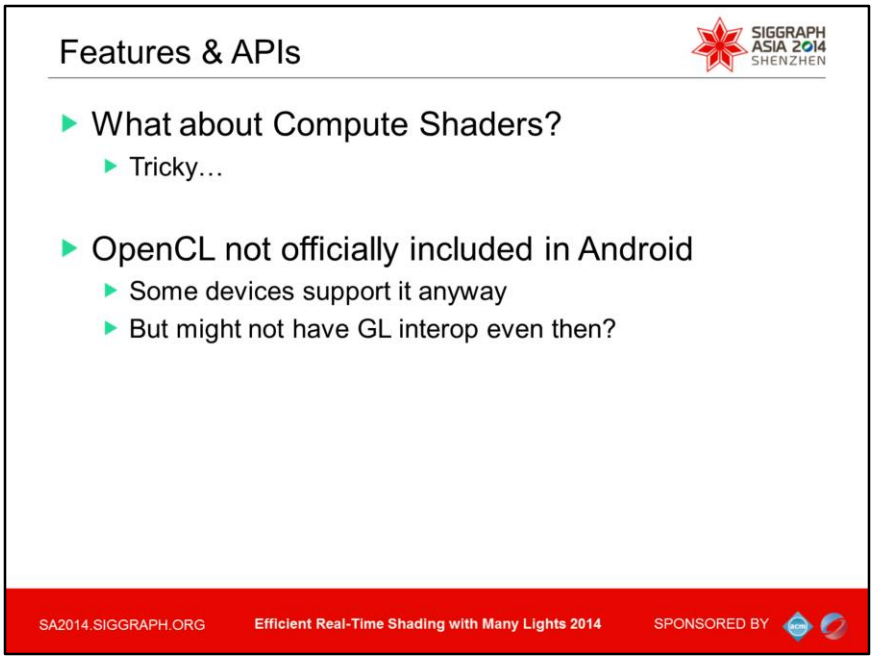
Another difference that I mentioned was in terms of features available on mobile platforms. So, if you're developing for a mobile platform, you'll likely end up using OpenGL|ES 2.0 or 3.something.

For now, I'd say that 2.0 is still somewhat common, at least if you're targeting devices that are out there right now. For us, the most interesting properties of OpenGL|ES 2.0 is its support for shaders and framebuffer objects. However, at least in the core spec, there's no mandatory support for multiple render targets.

OpenGL|ES 3.0 is getting adopted, and has quite a few interesting features. Most interesting for this talk is the in-core support for multiple render targets. However, without extensions, there are no renderable floating point color formats, which might be an issue for deferred-style methods.

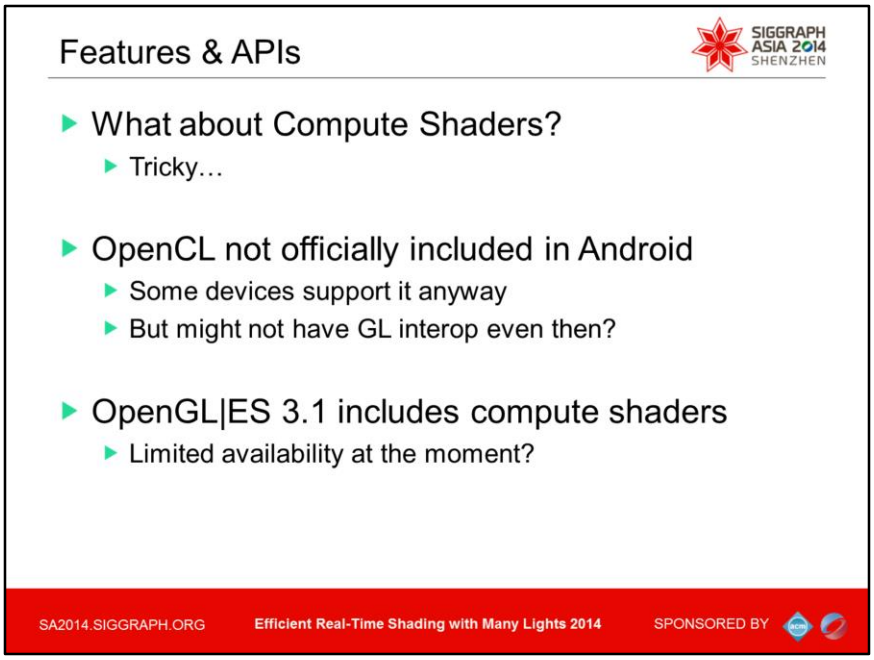


Another issue is related to compute shaders. Our techniques use compute shaders quite extensively, as you've already heard. The situation here is a bit tricky…



Another issue is related to compute shaders. Our techniques use compute shaders quite extensively, as you've already heard. The situation here is a bit tricky…

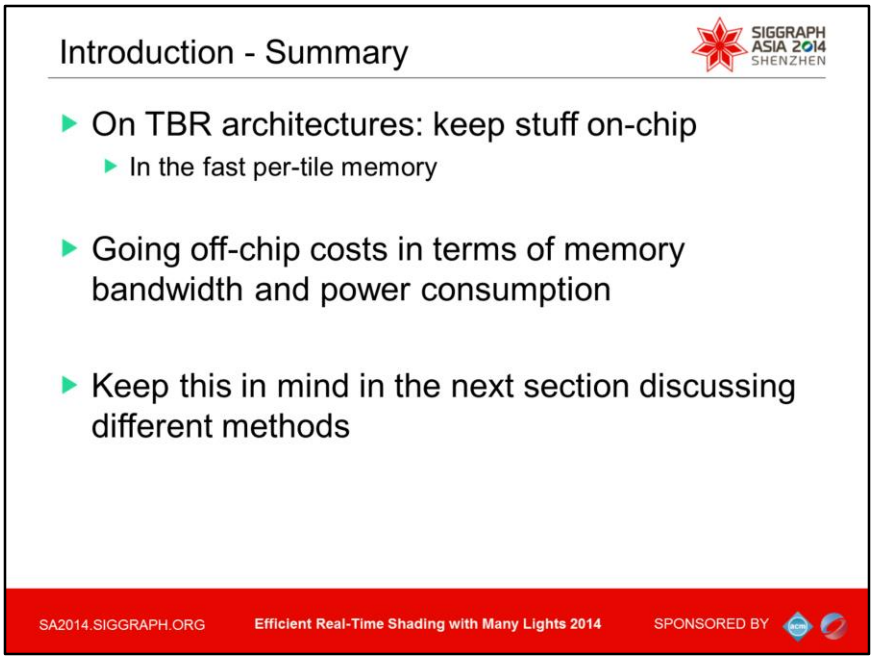
For instance, OpenCL is not officially included in Android – despite this, some devices include working OpenCL drives. Even so, these drives do not always include OpenGL|ES interoperability functions, so combining OpenGL rendering with OpenCL compute can be a bit tricky.



Another issue is related to compute shaders. Our techniques use compute shaders quite extensively, as you've already heard. The situation here is a bit tricky…

For instance, OpenCL is not officially included in Android – despite this, some devices include working OpenCL drives. Even so, these drives do not always include OpenGL|ES interoperability functions, so combining OpenGL rendering with OpenCL compute can be a bit tricky.

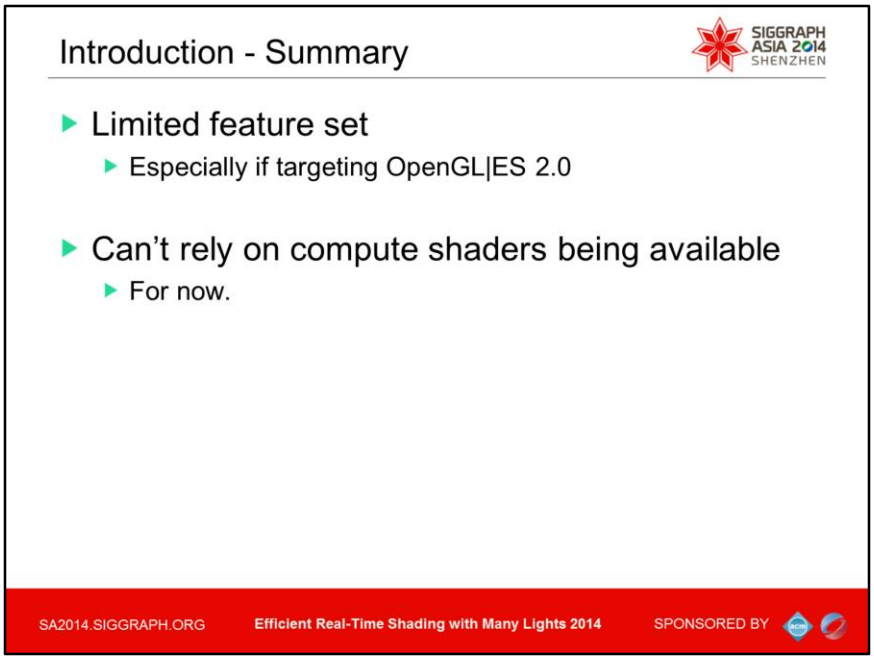
The good news is that OpenGL|ES 3.1 includes compute shaders, so this problem will solve itself given time. At the moment, adoption of ES 3.1 is a bit limited, though, so this might still be an issue.



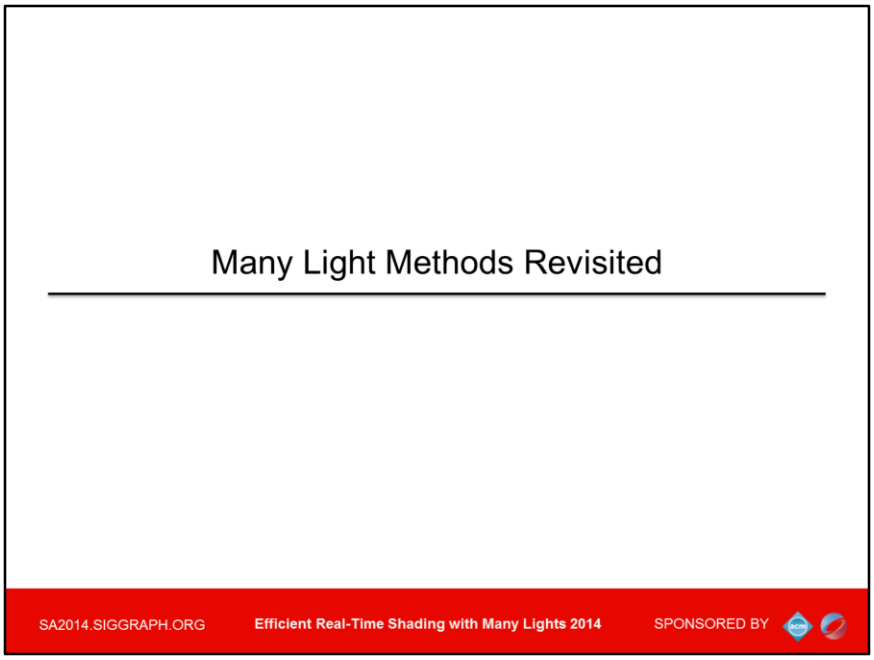
To summarize this first part:

On a TBR architecture, which is the dominating architecture in the mobile space, we want to keep the frame buffer data in fast on-chip memory whenever possible, since going off-chip is costly in terms of memory bandwidth and power consumption.

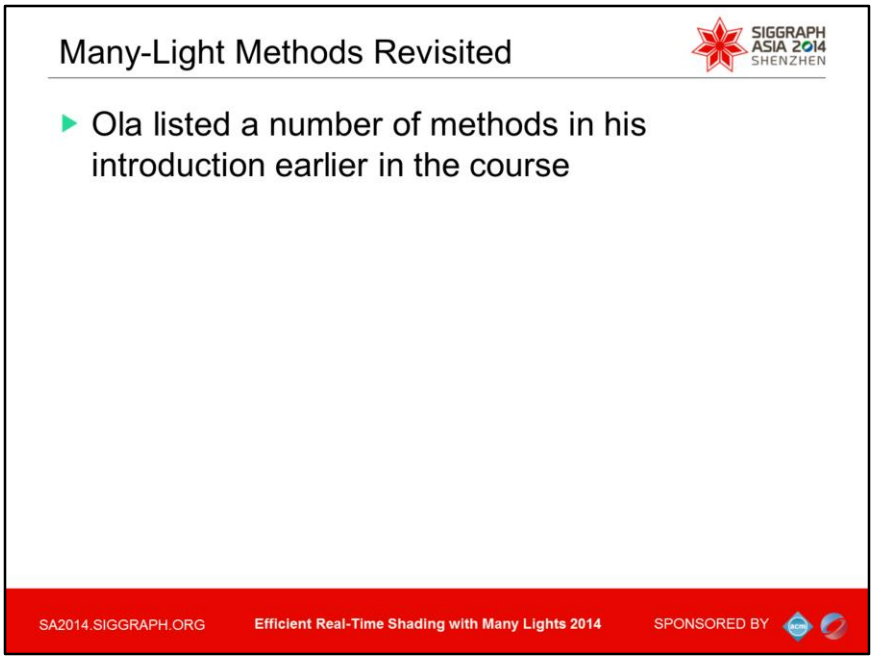
I'm going to refer back to this when discussing the different methods in the next part of this presentation.



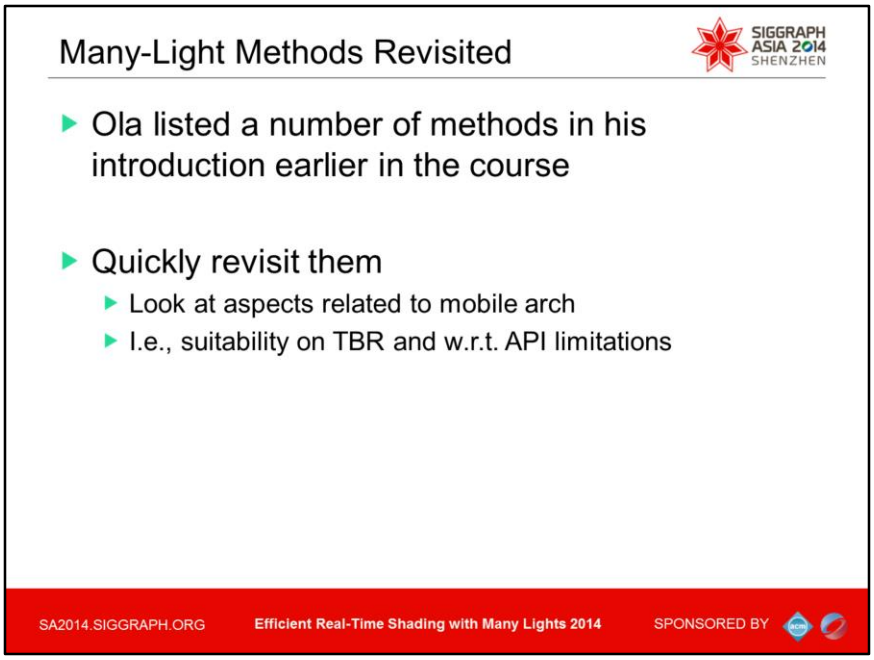
For now, there's still a somewhat limited feature set, especially if also targeting OpenGL|ES 2.0. Also, I'd say, that for now, you can't rely on compute shaders being available on device.



With this, let's move on to the next part of the presentation: the different many-light methods.

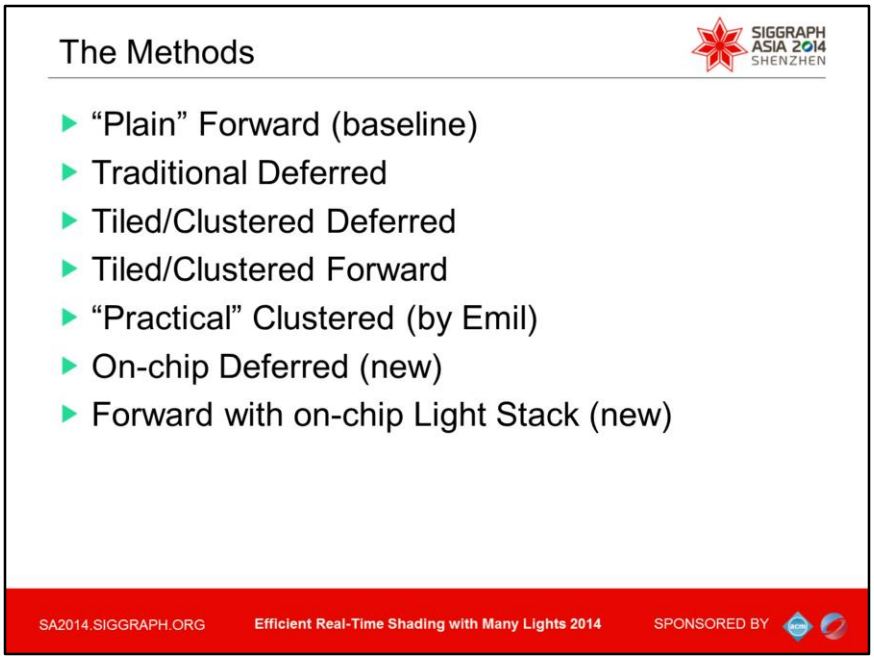


In the introduction to the course, Ola listed a number of many-light methods.



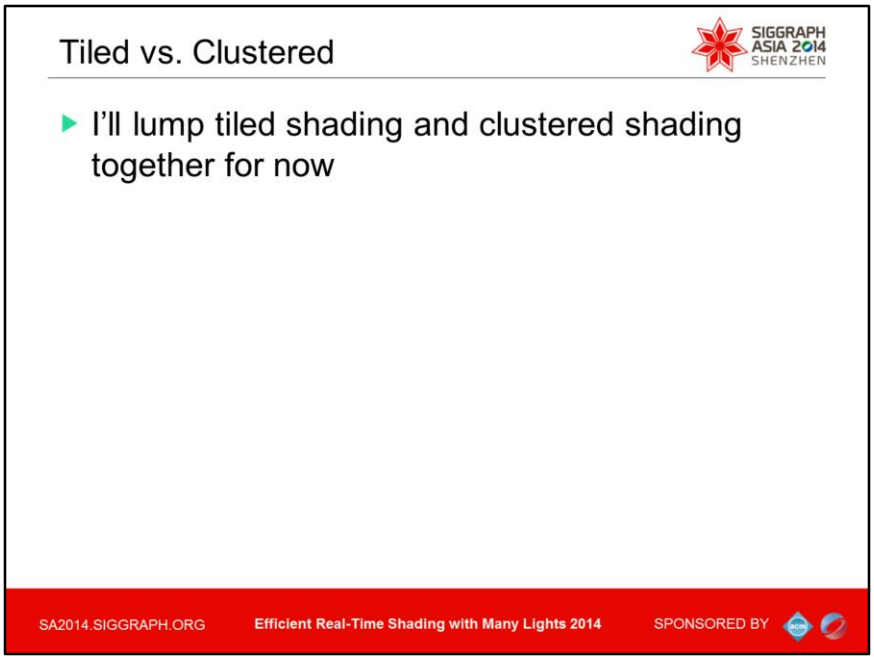
In the introduction to the course, Ola listed a number of many-light methods.

I'm going to quickly revisit these. This serves as a sort of repetition, but also as a service to those of you who preferred to sleep in today ☺. I'm also going to look at the different properties of these methods with respect to their suitability on TBR architectures, and what kind of API limitations that you might run into.

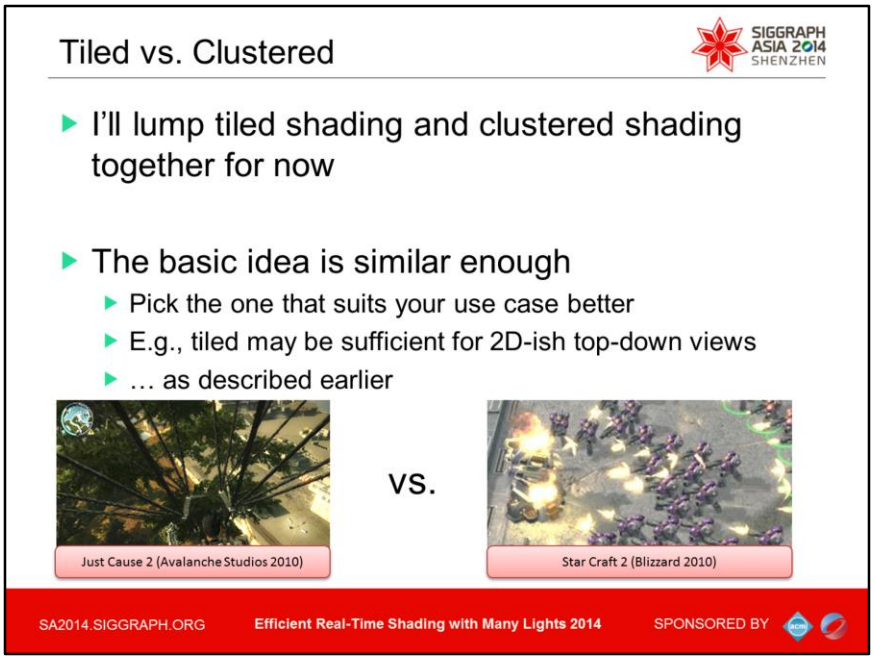


The methods that I'm going to look at are as follows:

- There's the plain forward method, which is included as a reference.
- Next, there's the deferred methods
- Then, we have the clustered methods, including the practical one presented by Emil
- Finally, I'm quickly going to introduce two new methods that were presented by Martin et al. at SIGGRAPH 2013. These methods target TBR-like architectures specifically.



For this review, I'm not going to make any distinction between tiled and clustered shading.

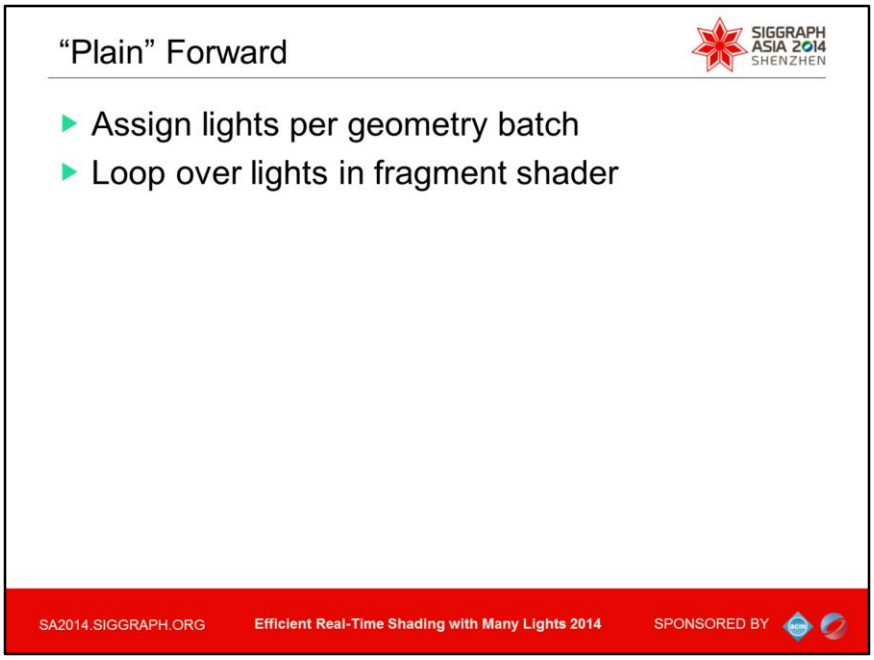


For this review, I'm not going to make any distinction between tiled and clustered shading.

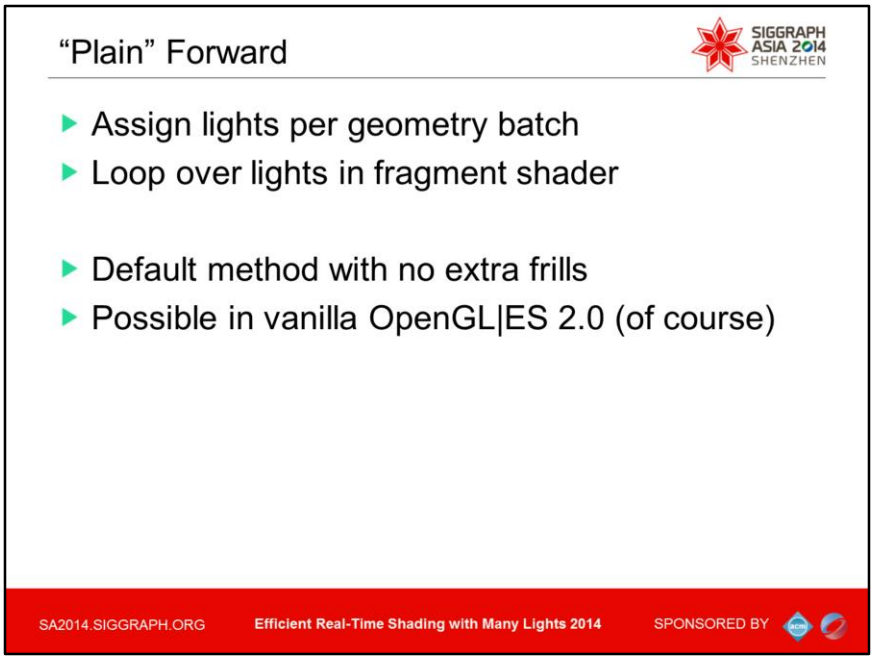
Tiled and Clustered are very similar in spirit – in fact, tiling is more or less a special case of clustering. When implementing any of the methods, you should pick the method that matches your use case better. So, for instance, if you have, for example, a largely top-down view with little depth-complexity and discontinuities, you should maybe opt for tiled shading. In this 2D-ish case, it likely performs better, because of the reduced overheads compared to dealing with a full 3D clustering.

If this isn't the case, i.e., if you have a first-/third-person view, clustering might be a better option, since it's more robust with respect to varying views, as discussed in previous parts of the course.

(Extra note: Even if opting for the "full" clustering, you might want to adapt it more to your use case. So, depending on your needs, a simpler clustering with e.g. fewer depth-layers may be worth it. Or, perhaps you want to something entirely different, like a world-space-based clustering?)

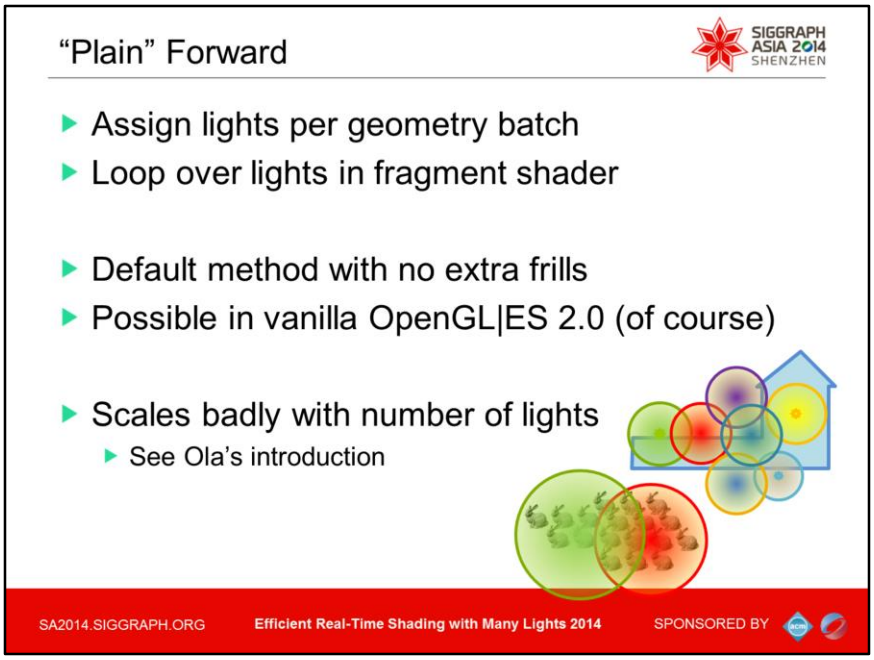


The first method that I'm revisiting is the "plain" forward rendering method. Here, you assign lights to each geometry batch that is to be drawn. During shading, you then loop over all the lights in your fragment shader and compute the lighting.



The first method that I'm revisiting is the "plain" forward rendering method. Here, you assign lights to each geometry batch that is to be drawn. During shading, you then loop over all the lights in your fragment shader and compute the lighting.

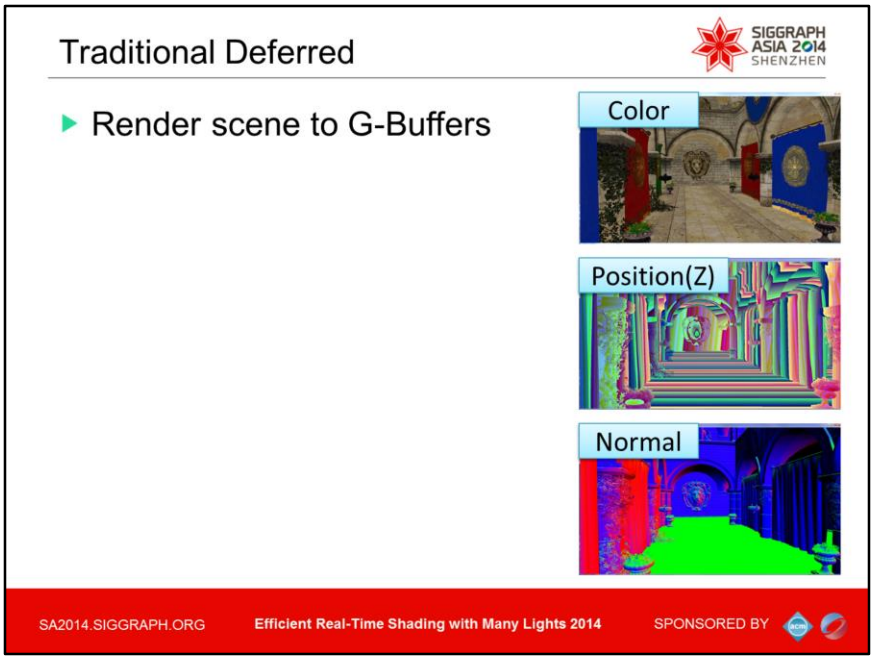
This is pretty much the text-book way of doing rendering in OpenGL, so obviously it's possible in e.g., OpenGL|ES 2.0.



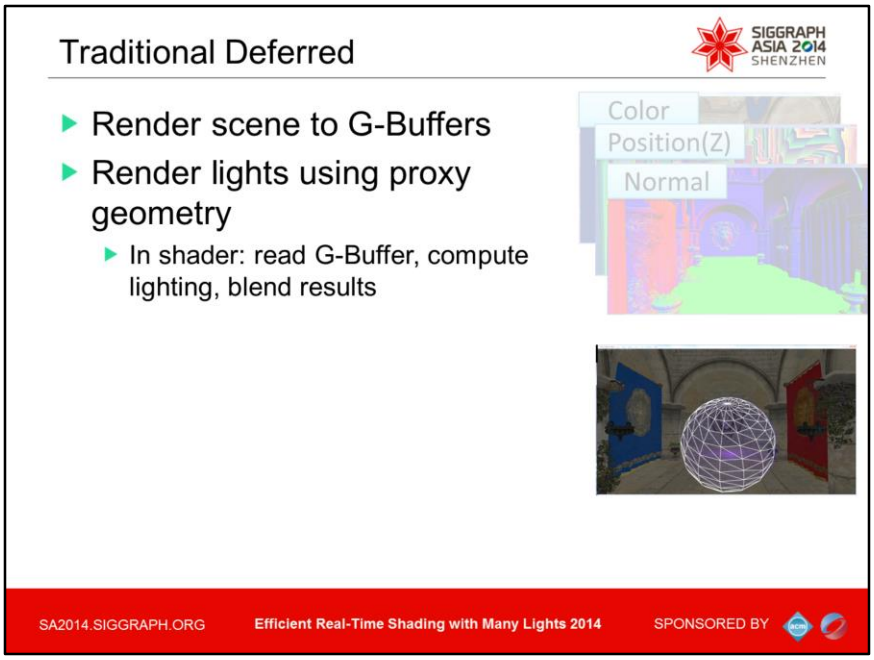
The first method that I'm revisiting is the "plain" forward rendering method. Here, you assign lights to each geometry batch that is to be drawn. During shading, you then loop over all the lights in your fragment shader and compute the lighting.

This is pretty much the text-book way of doing rendering in OpenGL, so obviously it's possible in e.g., OpenGL|ES 2.0.

It can support multiple lights, but as explained by Ola in the introduction, robustly supporting scenes with many lights is tricky.

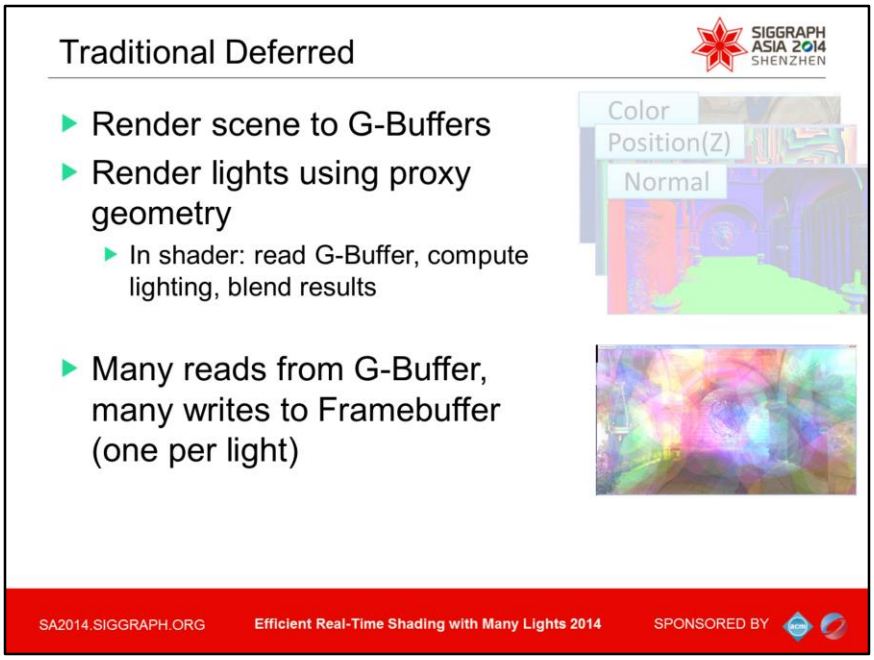


Next up, there's the traditional deferred shading method. Here, the scene is first rendered to generate G-Buffers that store the data we need to compute shading.



Next up there's the traditional deferred shading method. Here, the scene is first rendered to generate G-Buffers that store the data we need to compute shading.

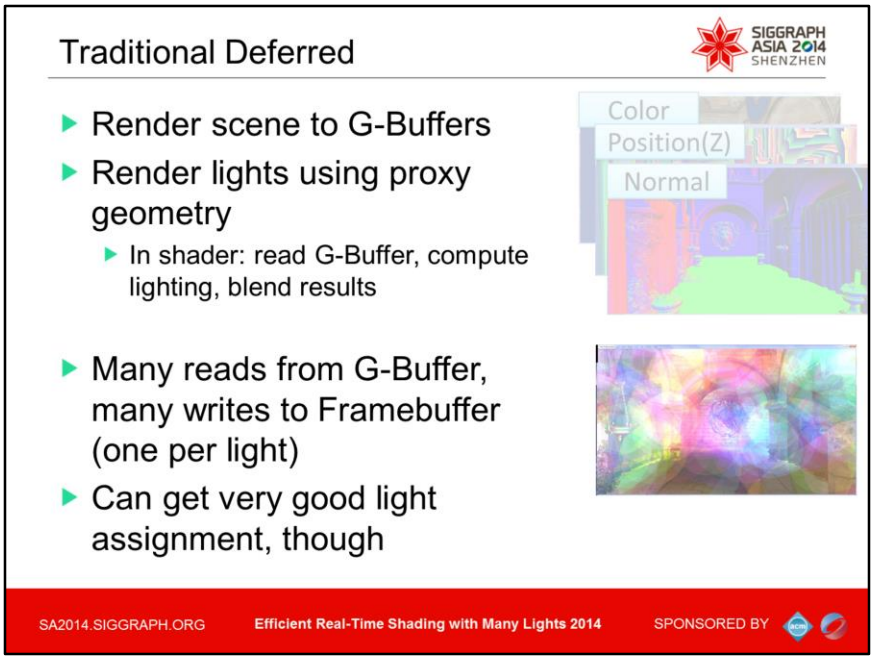
Later, lights are rendered using proxy geometry. For each generated fragment, we sample the G-Buffers, compute the contribution from the current light source and accumulate the results to the frame buffer via blending.



Next up there's the traditional deferred shading method. Here, the scene is first rendered to generate G-Buffers that store the data we need to compute shading.

Later, lights are rendered using proxy geometry. For each generated fragment, we sample the G-Buffers, compute the contribution from the current light source and accumulate the results to the frame buffer via blending.

This requires multiple reads from the G-Buffer, and multiple writes to the output color buffer, one per light, in fact.

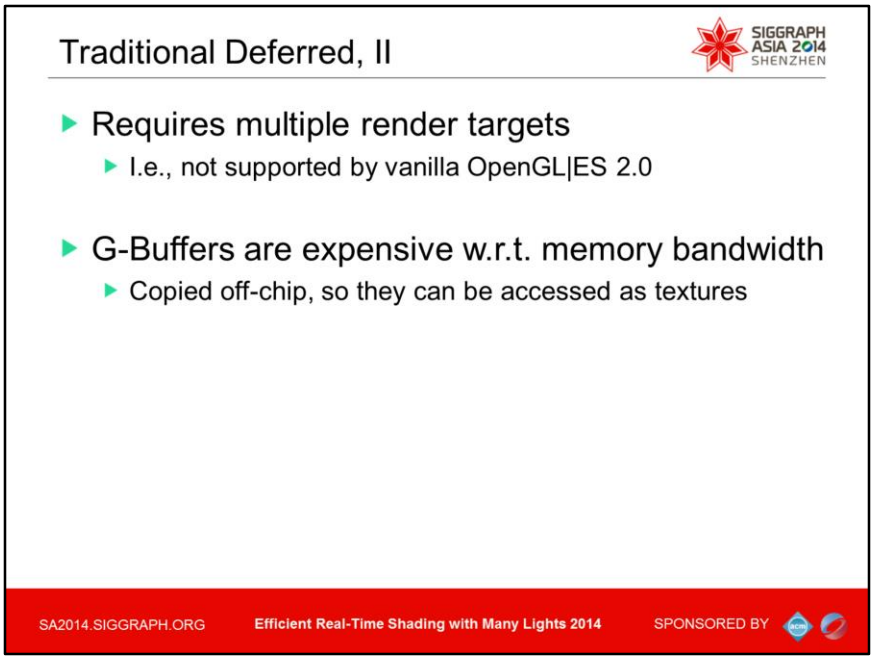


Next up there's the traditional deferred shading method. Here, the scene is first rendered to generate G-Buffers that store the data we need to compute shading.

Later, lights are rendered using proxy geometry. For each generated fragment, we sample the G-Buffers, compute the contribution from the current light source and accumulate the results to the frame buffer via blending.

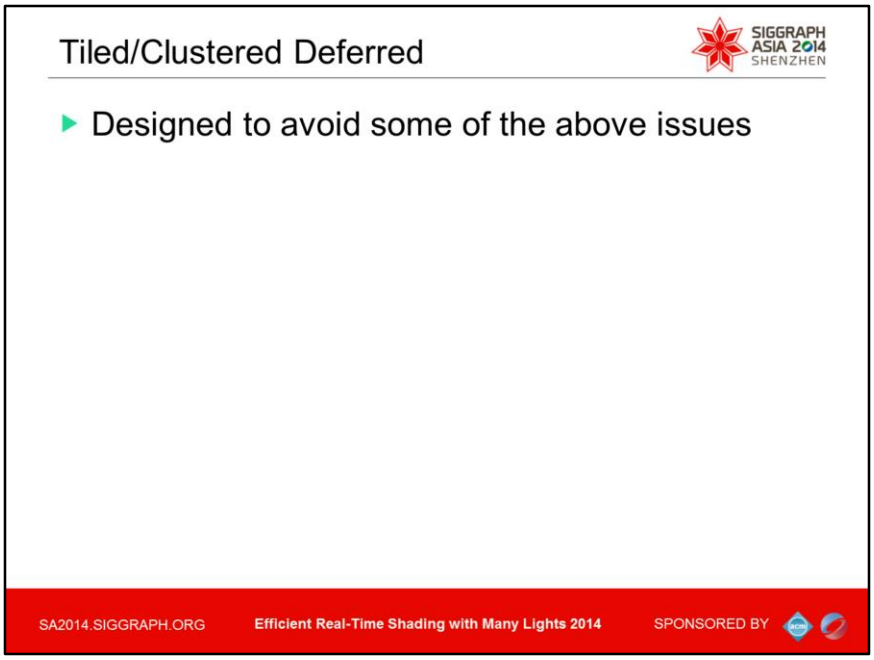
This requires multiple reads from the G-Buffer, and multiple writes to the output color buffer, one per light, in fact.

On the plus side, we can get very good light assignment with this method: with good proxy geometry, the light assignment can become pixel accurate.



Rendering G-Buffers requires support for multiple render targets, something that's not available in core OpenGL|ES 2.0. It's possible to work around this by doing a geometry pass for each G-Buffer output component, but doing that many geometry passes is typically not viable.

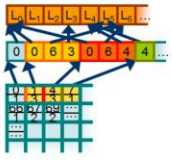
Secondly, in order to be accessible as textures during lighting, the G-Buffers need to be transferred off-chip and stored in RAM. Repeatedly reading G-Buffers is expensive with respect to memory bandwidth even on desktop GPUs, so memory bandwidth is definitively an issue on mobile hardware.



We then have the tiled/clustered deferred variation of deferred shading. It's designed to avoid some of the issues of the traditional deferred method – namely the repeated sampling of the G-Buffers.

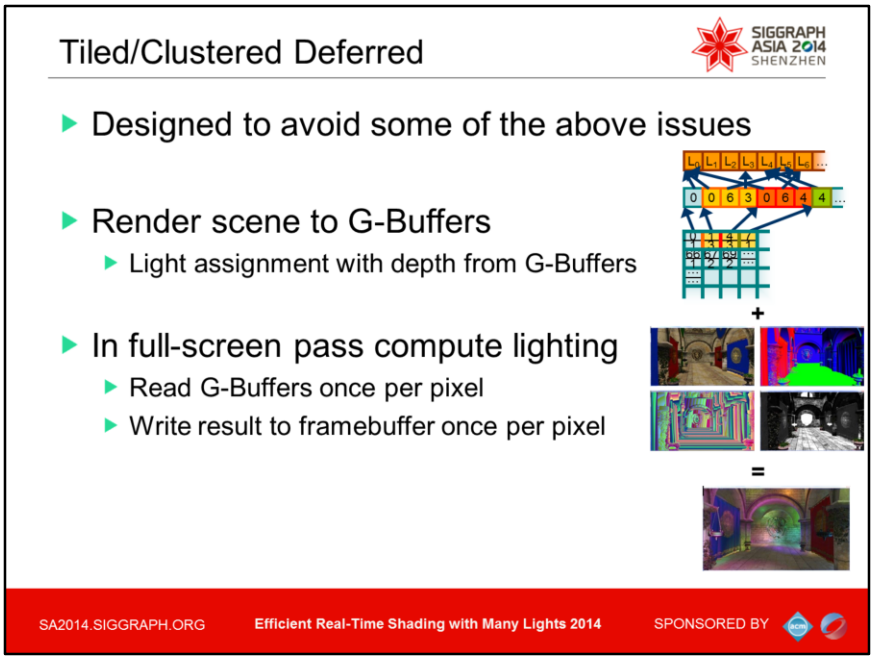
Tiled/Clustered Deferred

- Designed to avoid some of the above issues
- Render scene to G-Buffers
  - Light assignment with depth from G-Buffers



We then have the tiled/clustered deferred variation of deferred shading. It's designed to avoid some of the issues of the traditional deferred method – namely the repeated sampling of the G-Buffers.

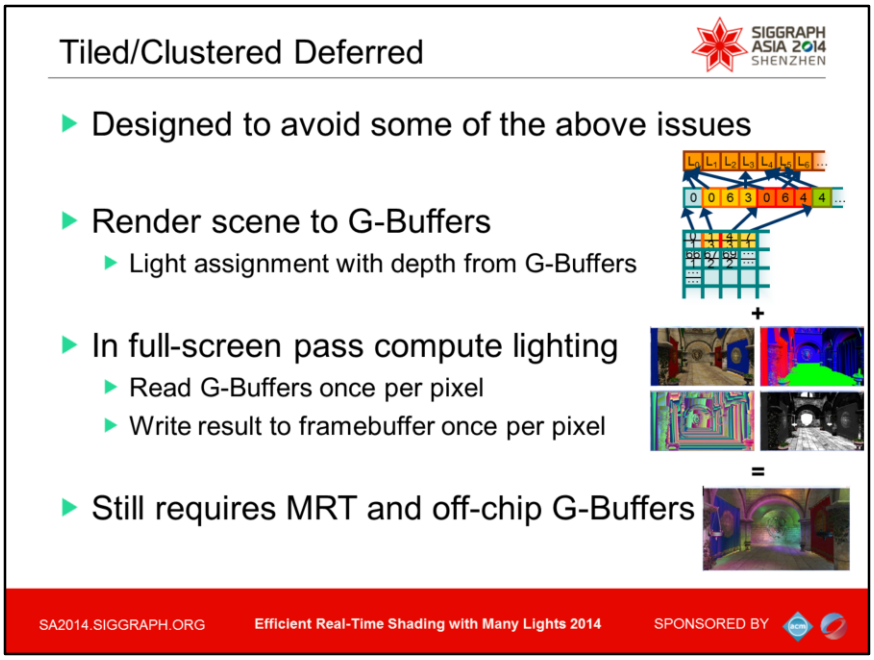
In these methods we again start by rendering the scene to G-Buffers. Using the depth-component from the G-Buffers, we then perform light assignment to tiles or clusters. From the light-assignment, we get **per-tile** or **per-cluster** light-lists.



We then have the tiled/clustered deferred variation of deferred shading. It's designed to avoid some of the issues of the traditional deferred method – namely the repeated sampling of the G-Buffers.

In these methods we again start by rendering the scene to G-Buffers. Using the depth-component from the G-Buffers, we then perform light assignment to tiles or clusters. From the light-assignment, we get ***per-tile*** *or* ***per-cluster*** *light-lists.*

Later, during a full-screen pass, we compute lighting. For each pixel, we read the G-Buffers once, and then find which tile or cluster it belongs to. The tile/cluster tells us which lights might be affecting pixels in that tile/cluster, so we simply loop over those lights and accumulate the result in the shader. At the end, we write the results to our output color buffer once.

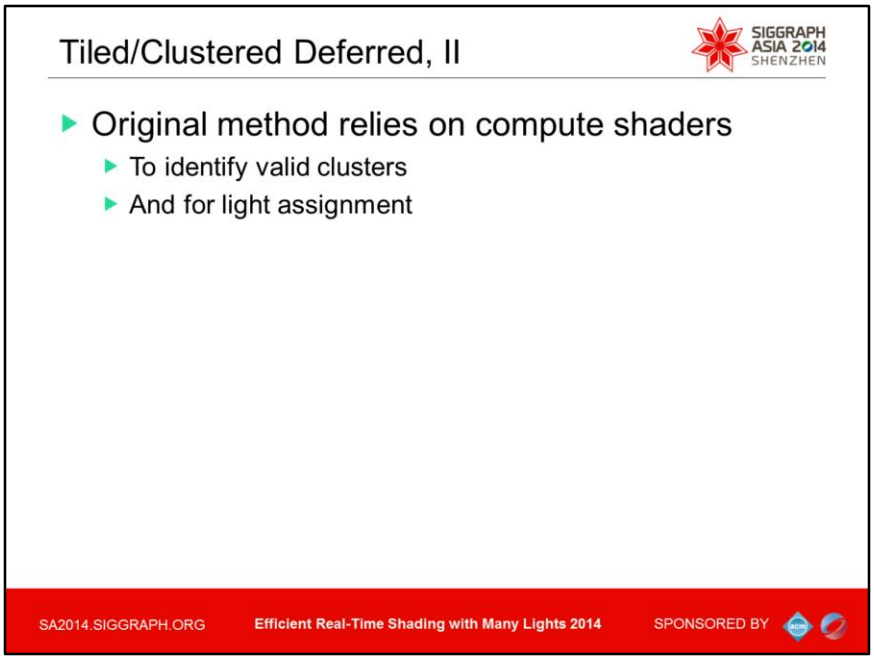


We then have the tiled/clustered deferred variation of deferred shading. It's designed to avoid some of the issues of the traditional deferred method – namely the repeated sampling of the G-Buffers.

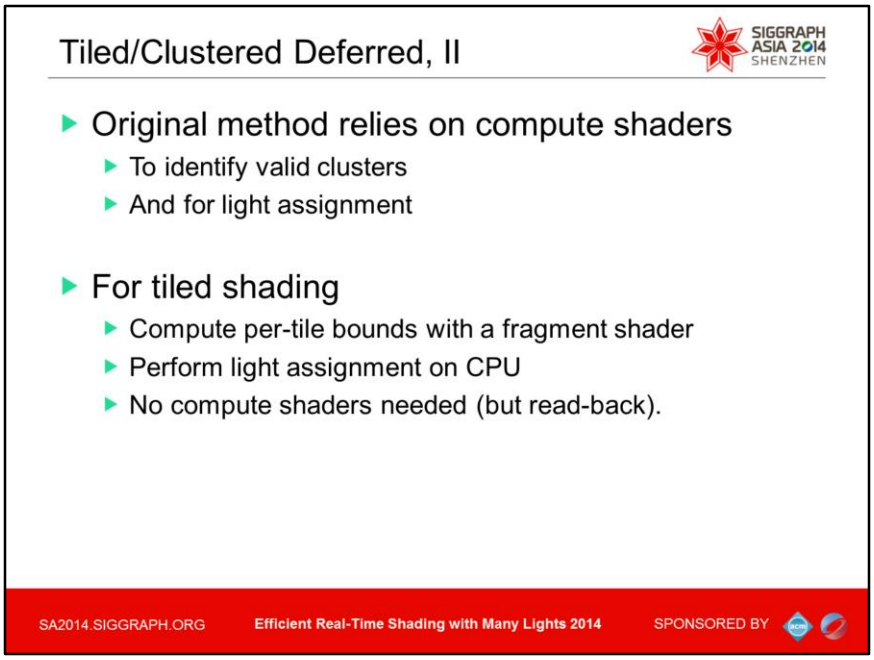
In these methods we again start by rendering the scene to G-Buffers. Using the depth-component from the G-Buffers, we then perform light assignment to tiles or clusters. From the light-assignment, we get ***per-tile*** *or* ***per-cluster*** *light-lists.*

Later, during a full-screen pass, we compute lighting. For each pixel, we read the G-Buffers once, and then find which tile or cluster it belongs to. The tile/cluster tells us which lights might be affecting pixels in that tile/cluster, so we simply loop over those lights and accumulate the result in the shader. At the end, we write the results to our output color buffer once.

We've avoided the multiple reads from the G-Buffer and the multiple writes to the output color buffer, and thereby reduced the required bandwidth quite significantly. However, we still need support for MRT and G-Buffer data is still transferred off-chip

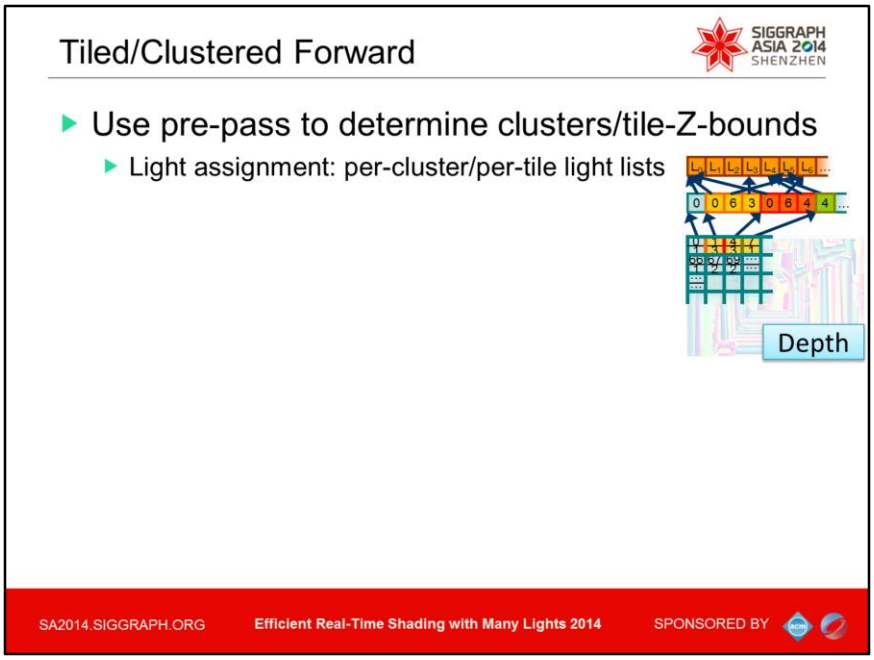


Our original clustered-shading method further relies quite heavily on compute shaders. We use compute shaders to first identify valid clusters from the G-Buffer data, and later for light assignment.



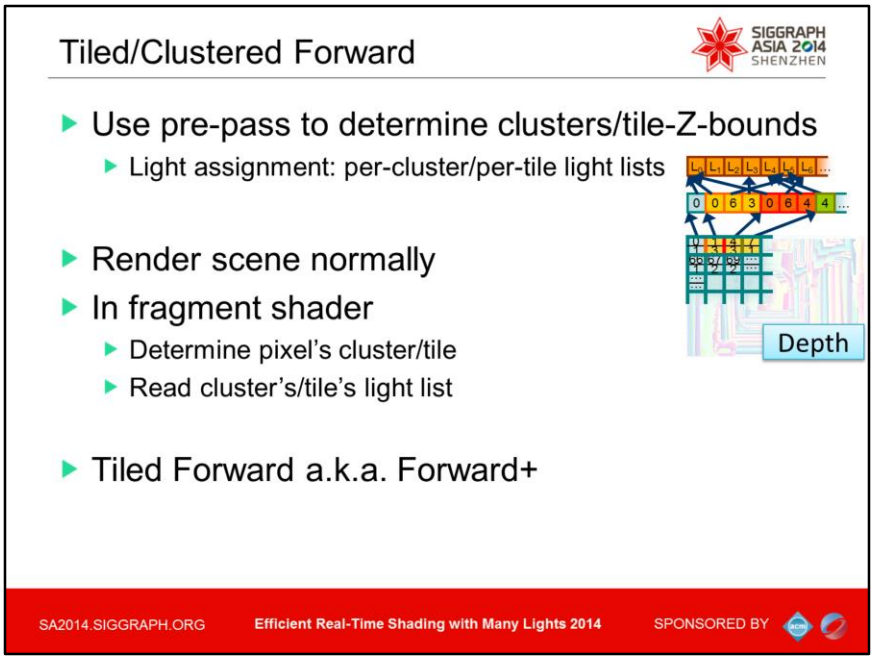
Our original clustered-shading method further relies quite heavily on compute shaders. We use compute shaders to first identify valid clusters from the G-Buffer data, and later for light assignment.

For tiled shading the situation is a bit different. You can compute the per-tile Z-bounds in a fragment shader, and then perform the light assignment on the CPU. This eliminates the need for compute shaders, but instead requires transfer of data from OpenGL|ES objects to CPU-accessible system memory.



Next up, we have the forward variation of the tiled/clustered method.

Here, instead of rendering full G-Buffers, we perform a depth-only pre-pass. From this, we can find active clusters, or in the case of tiled, find the per-tile depth bounds. Again, performing the light assignment gives us **per-tile** or **per-cluster** light lists.

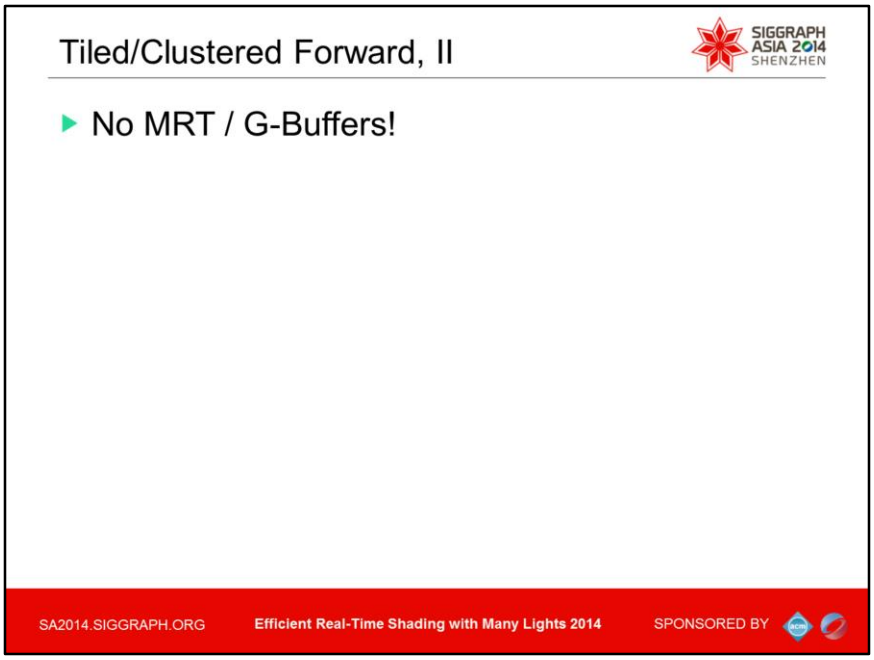


Next up, we have the forward variation of the tiled/clustered method.

Here, instead of rendering full G-Buffers, we perform a depth-only pre-pass. From this, we can find active clusters, or in the case of tiled, find the per-tile depth bounds. Again, performing the light assignment gives us ***per-tile*** *or* ***per-cluster*** *light lists.*

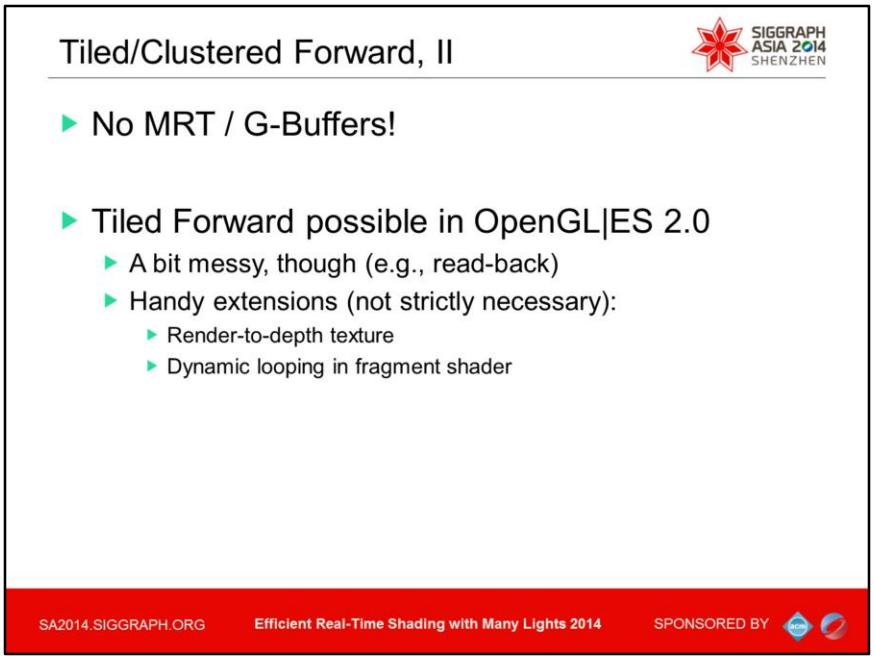
After light assignment, we render the scene "normally", in a forward fashion. Here, for each generated fragment, we determine which tile/cluster it belongs to, and access the light lists of that tile/cluster to compute shading.

Note: in the literature, tiled forward is also known as "Forward+", so if you look for information on this method, make sure to search for that term as well.



One of the key properties here is that we no longer need support for multiple render targets, and there are no more heavy G-Buffers. (*)

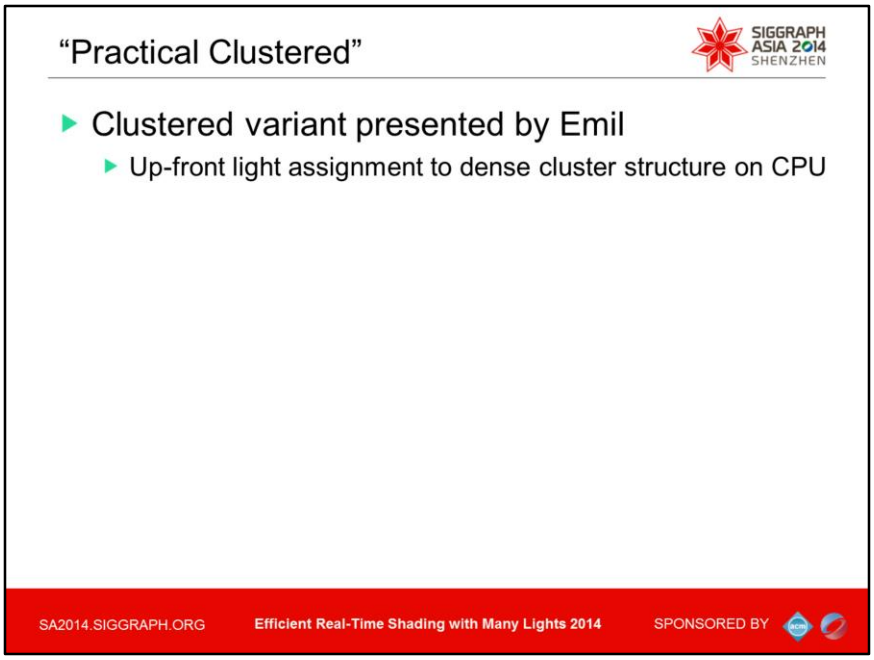
(*) Note – added after presentation: We still need to copy the depth buffer to off-chip memory, since it's needed when finding clusters/tile-Z-bounds.



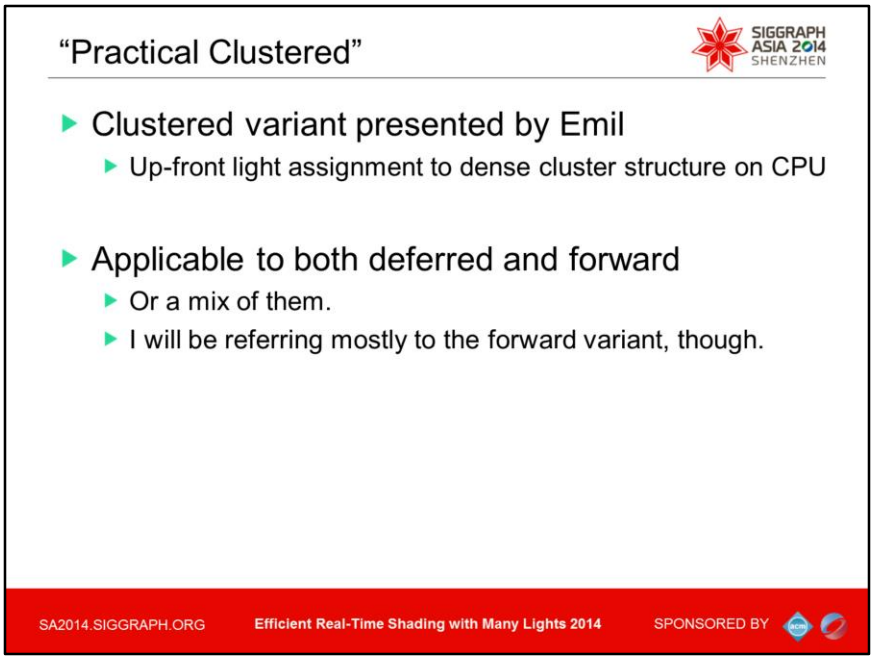
One of the key properties here is that we no longer need support for multiple render targets, and there are no more heavy G-Buffers. (*)

It's possible to implement Tiled Forward in a pure OpenGL|ES 2.0 environment. This is a bit messy. It requires a read-back from OpenGL to system memory for the light assignment. There's also a few extensions that make life a bit easier: render-to-depth texture and being able to perform dynamic looping in the fragment shader.

(*) Note – added after presentation: We still need to copy the depth buffer to off-chip memory, since it's needed when finding clusters/tile-Z-bounds.

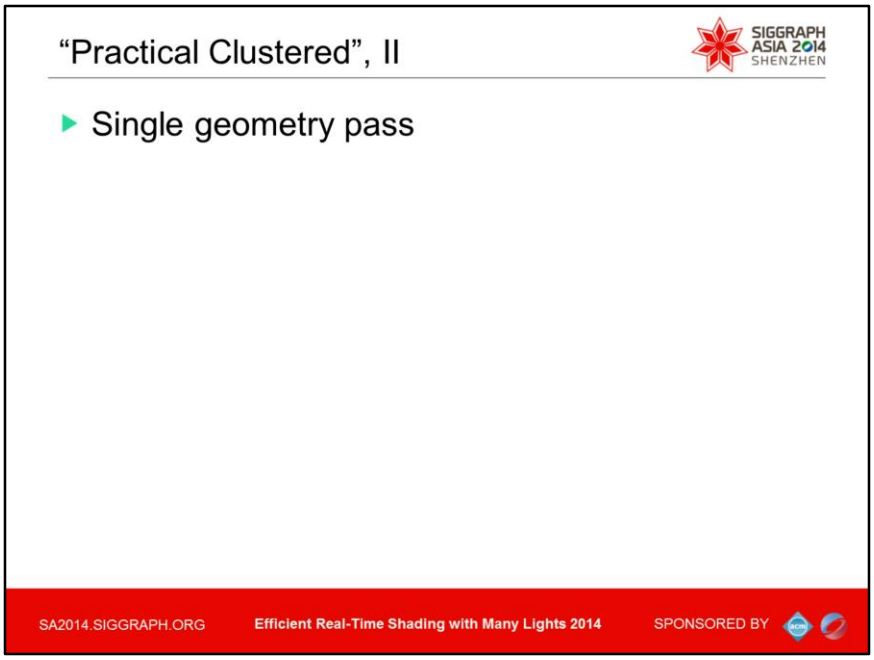


The next method is the clustered-shading variation that Emil presented in this talk. The key difference is that light-assignment is performed up-front to a dense grid / dense cluster structure on the CPU.

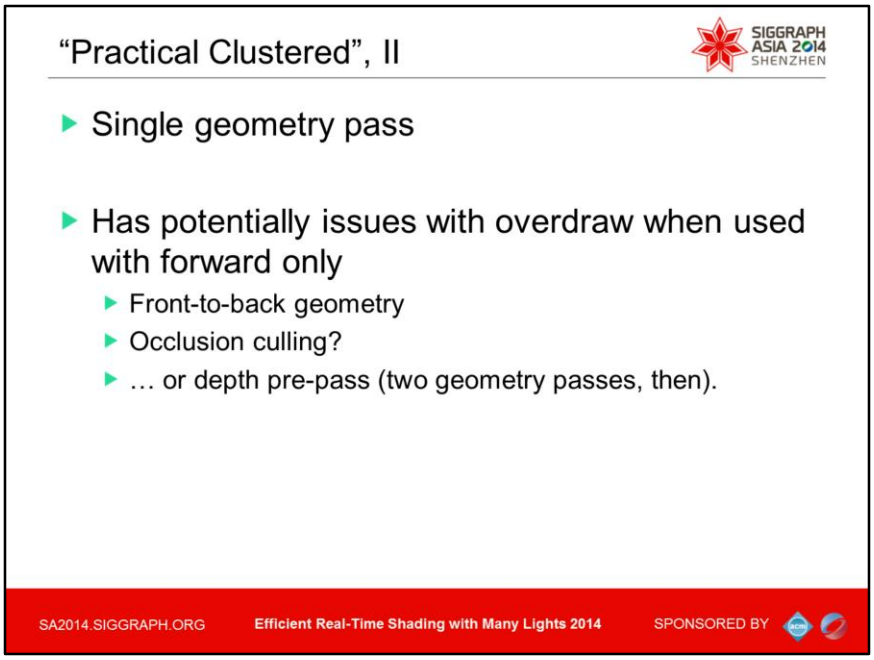


The next method is the clustered-shading variation that Emil presented in this talk. The key difference is that light-assignment is performed up-front to a dense grid / dense cluster structure on the CPU.

As Emil mentioned, it's applicable to both deferred and forward rendering, or a mix of the two. In my presentation, I'll be mostly referring to a strictly forward variant, though, since the forward variant avoids G-Buffers completely.

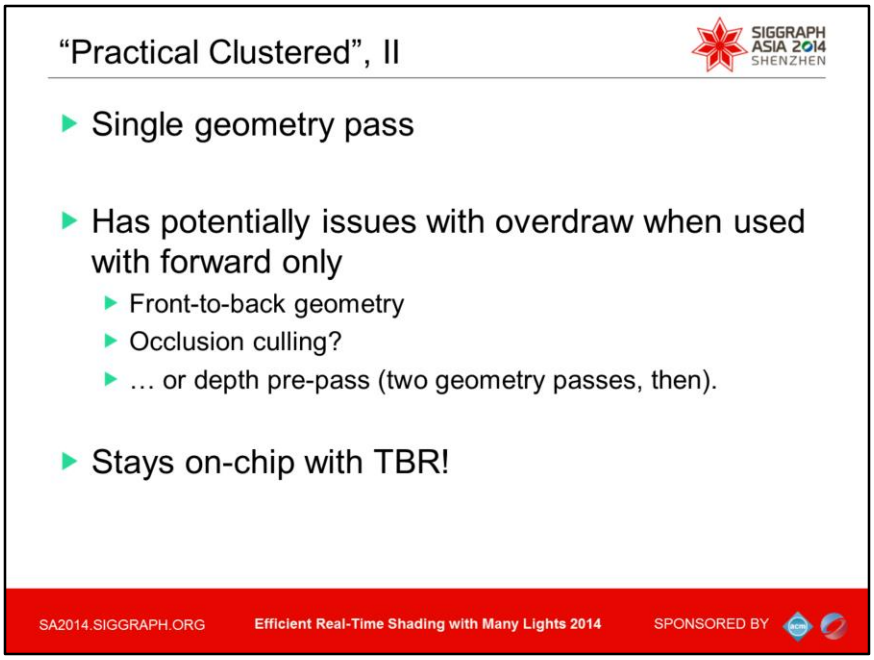


The main feature of the practical clustered forward method is that it can be done in a single geometry pass, and without G-Buffers.



The main feature of the practical clustered forward method is that it can be done in a single geometry pass, and without G-Buffers.

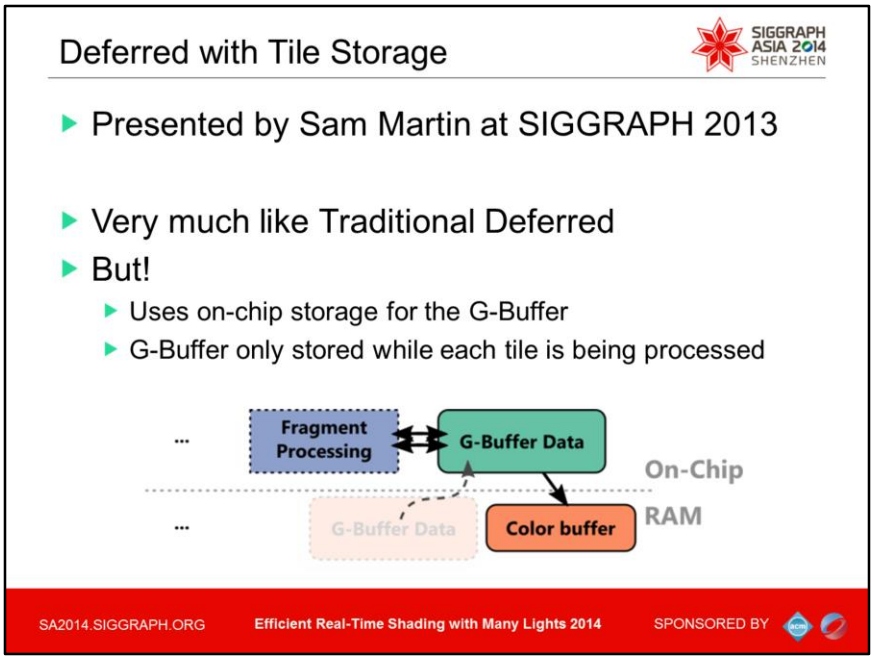
The downside is that it has, in the forward-only setting, problems with overdraw. This can be mitigated somewhat by drawing geometry front-to-back and, if possible, by using occlusion culling. Overshading can also be largely eliminated by performing a depth-only pre-pass, but at the cost of requiring two geometry passes.



The main feature of the practical clustered forward method is that it can be done in a single geometry pass, and without G-Buffers.

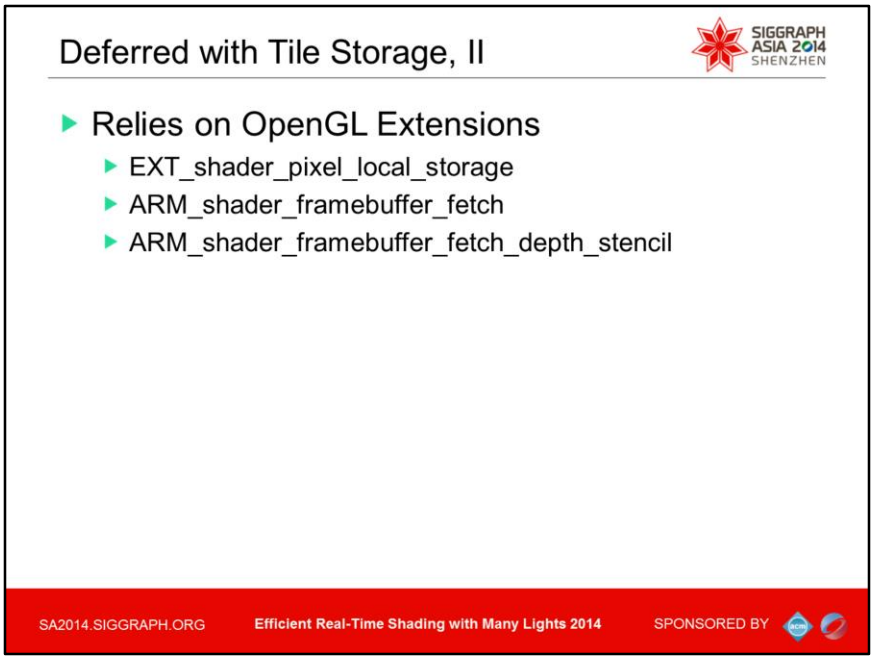
The downside is that it has, in the forward-only setting, problems with overdraw. This can be mitigated somewhat by drawing geometry front-to-back and, if possible, by using occlusion culling. Overshading can also be largely eliminated by performing a depth-only pre-pass, but at the cost of requiring two geometry passes.

The upside is that the frame buffer can stay on-chip on a TBR platform (even if a depth-only) pre-pass is employed.

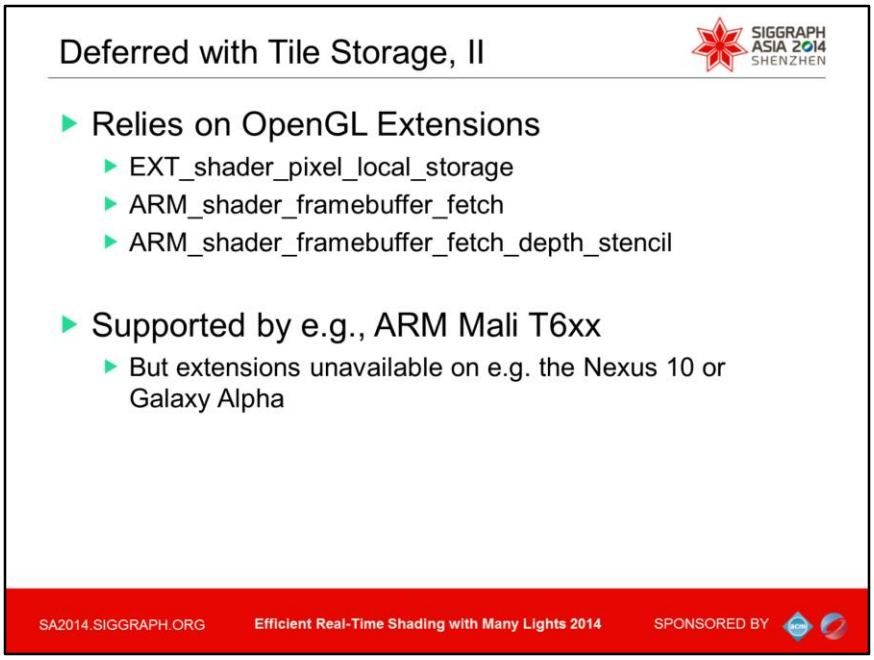


The next method I'm quickly summarizing is the method presented by Sam Martin at SIGGRAPH 2013, referred to as "Deferred with Tile Storage".

The method works very much like traditional deferred shading, but uses OpenGL|ES extensions to keep the G-Buffer data on-chip. A consequence of this is that the G-Buffer data is only available during the processing of each tile, but that's also when we primarily need that data.

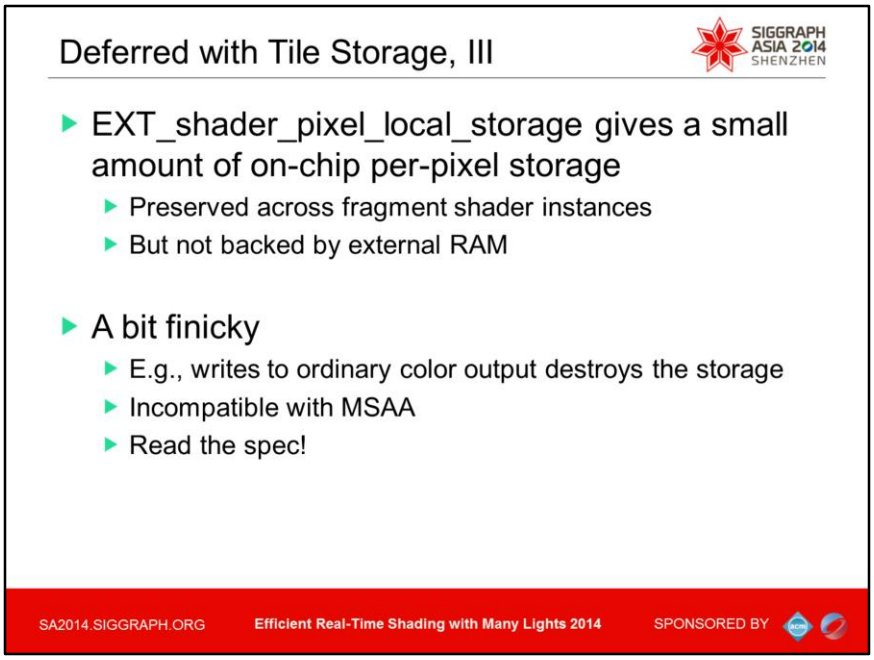


As mentioned, the method relies on some OpenGL|ES extensions, listed here.



As mentioned, the method relies on some OpenGL|ES extensions, listed here.

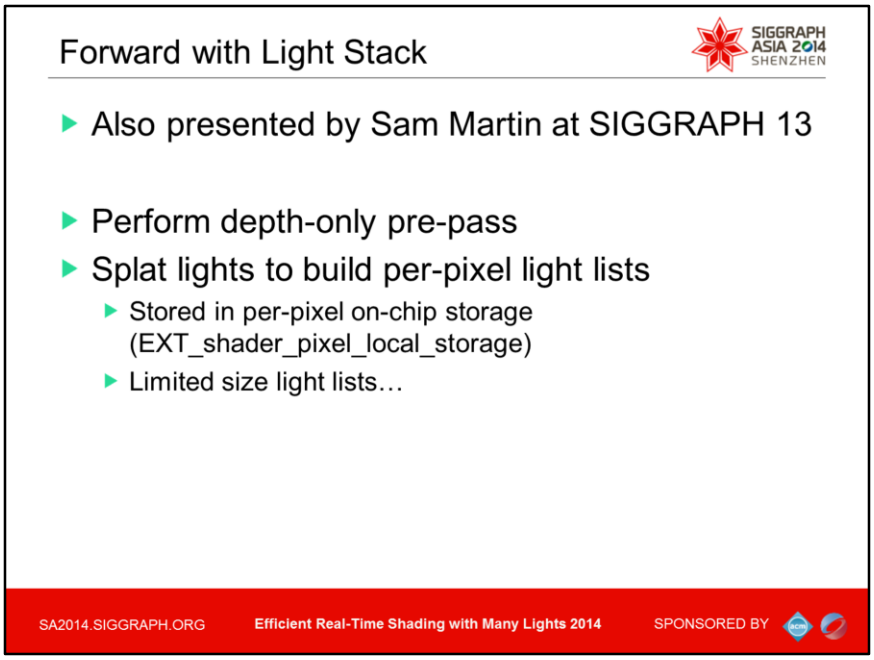
These extensions are supposedly supported by e.g., the ARM Mali T6somethingsomething GPUs, but on my devices, i.e., the Nexus 10 tablet and the Galaxy Alpha smartphone, they are unavailable at the moment. The original presentations mentions using a dev-board…



The key extension for this technique is the EXT_shader_pixel_local_storage one. It gives a small amount of on-chip per-pixel storage that is preserved across fragment shader invocations. So, the fragment shader can allocate this storage, and when generating a fragment with a set of pixel coordinates, it can write to the storage. Later, a fragment with the same pixel coordinates can read the data stored by the previous fragment. The storage is not backed by external RAM, however.

Using the extension can be a bit finicky. So, under certain conditions, the stored data is lost. For instance, writing to the ordinary color output would overwrite the local-storage of that pixel. It's also, for now, incompatible with certain features like MSAA.

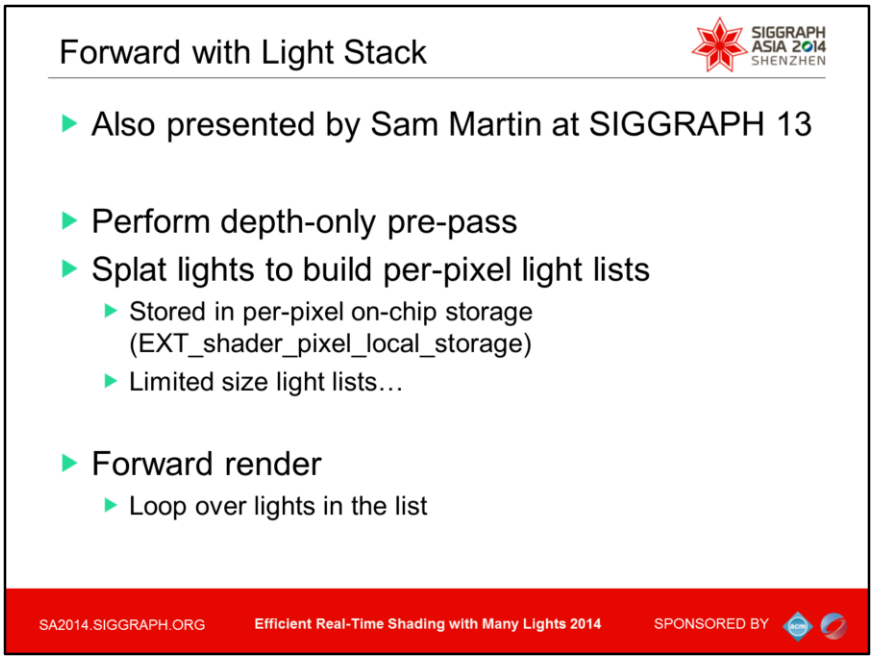
If you want to know more about this, be sure to check the talk by Sam Martin, and/or read the extension spec. There's a lot of valuable information there.



The next method was presented at the same talk. It performs rendering in a forward-fashion.

First, a depth-only pre-pass is performed. After this pass, light sources are rendered, using proxy-geometry, on top of the depth buffer. In the fragment shader, per-pixel light lists are built into the local on-chip storage provided by the extensions that I just discussed, i.e., the EXT_shader_pixel_local_storage extension.

Note: the storage provided by this extension is limited to e.g., 16 bytes on current GPUs, which limits the size of the light lists.

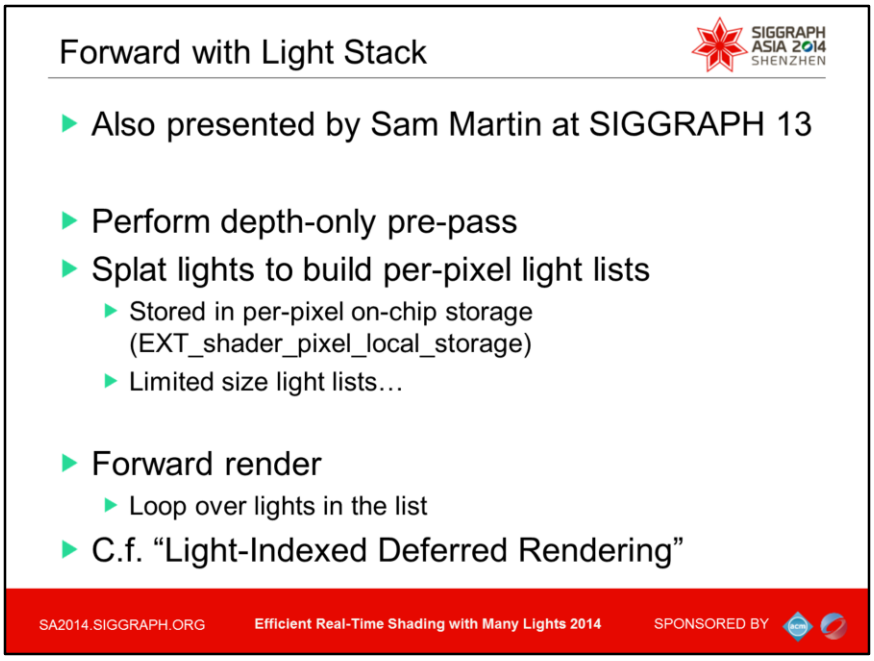


The next method was presented at the same talk. It performs rendering in a forward-fashion.

First, a depth-only pre-pass is performed. After this pass, light sources are rendered, using proxy-geometry, on top of the depth buffer. In the fragment shader, per-pixel light lists are built into the local on-chip storage provided by the extensions that I just discussed, i.e., the EXT_shader_pixel_local_storage extension.

Note: the storage provided by this extension is limited to e.g., 16 bytes on current GPUs, which limits the size of the light lists.

After building the per-pixel light lists, the geometry is rendered in a forward pass. Here, the light lists are in the local storage of that pixel, so they can be accessed quite efficiently.



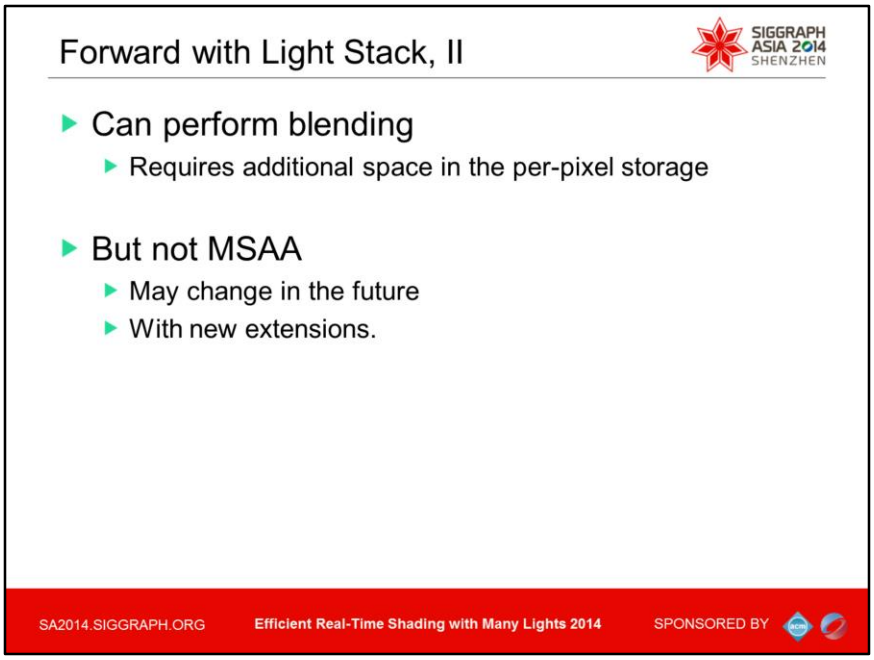
The next method was presented at the same talk. It performs rendering in a forward-fashion.

First, a depth-only pre-pass is performed. After this pass, light sources are rendered, using proxy-geometry, on top of the depth buffer. In the fragment shader, per-pixel light lists are built into the local on-chip storage provided by the extensions that I just discussed, i.e., the EXT_shader_pixel_local_storage extension.

Note: the storage provided by this extension is limited to e.g., 16 bytes on current GPUs, which limits the size of the light lists.

After building the per-pixel light lists, the geometry is rendered in a forward pass. Here, the light lists are in the local storage of that pixel, so they can be accessed quite efficiently.

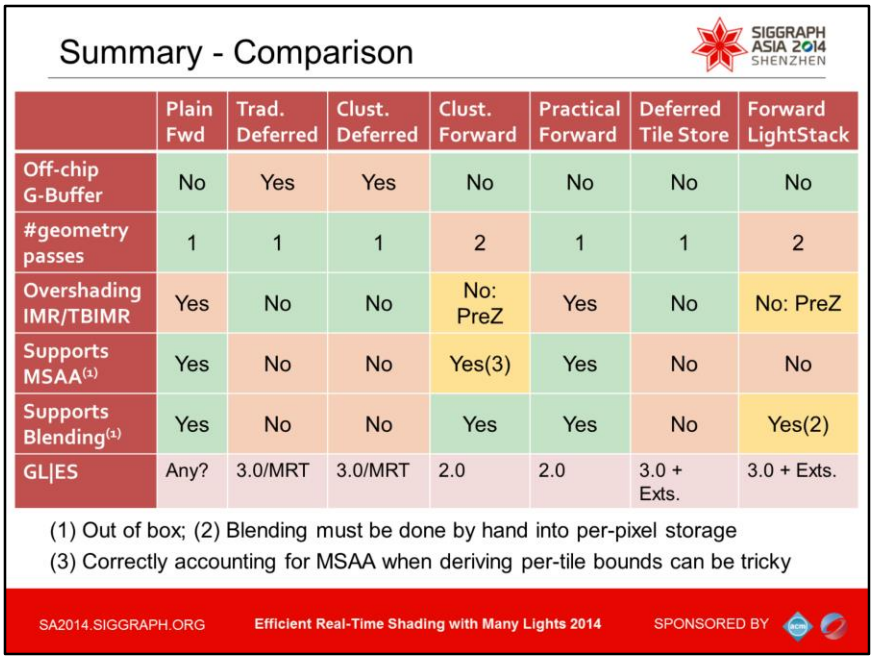
This method is quite similar to the one presented in the "Light-Indexed Deferred Rendering" article by Treblico in 2007. That method also builds per-pixel light lists.



This forward method can support blending. However, the blending must be computed manually in the fragment shader and stored in the per-pixel local storage - writing to the ordinary color output in the fragment shader would destroy the contents of the local storage, since the memory of the color output is shared with the local storage.

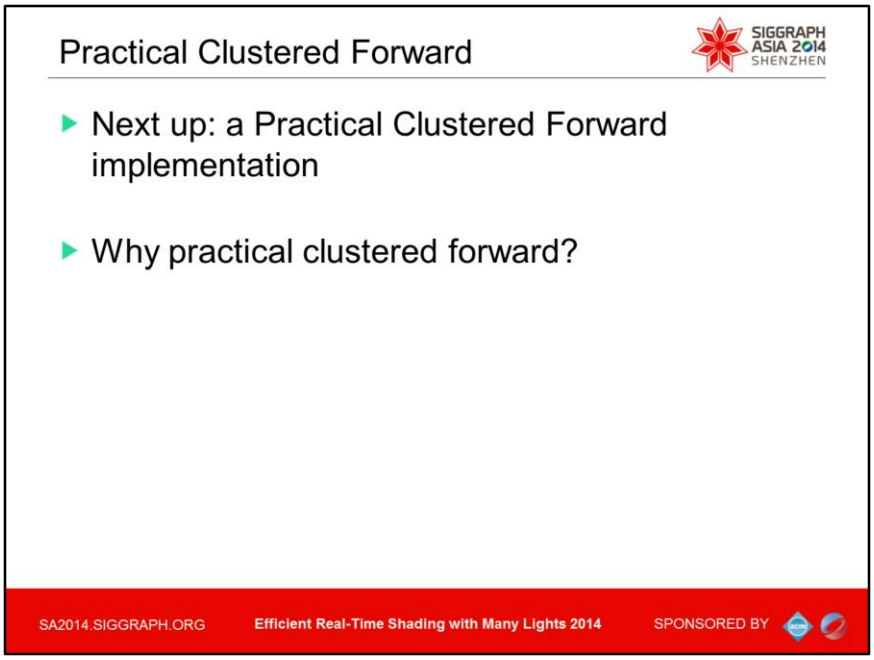
Storing the color in the local storage requires some additional space, so the per-pixel light lists are further limited in length.

Combining the per-pixel local-storage with MSAA is currently impossible, as per extension spec. This might however change in the future, with a new, improved, extension.



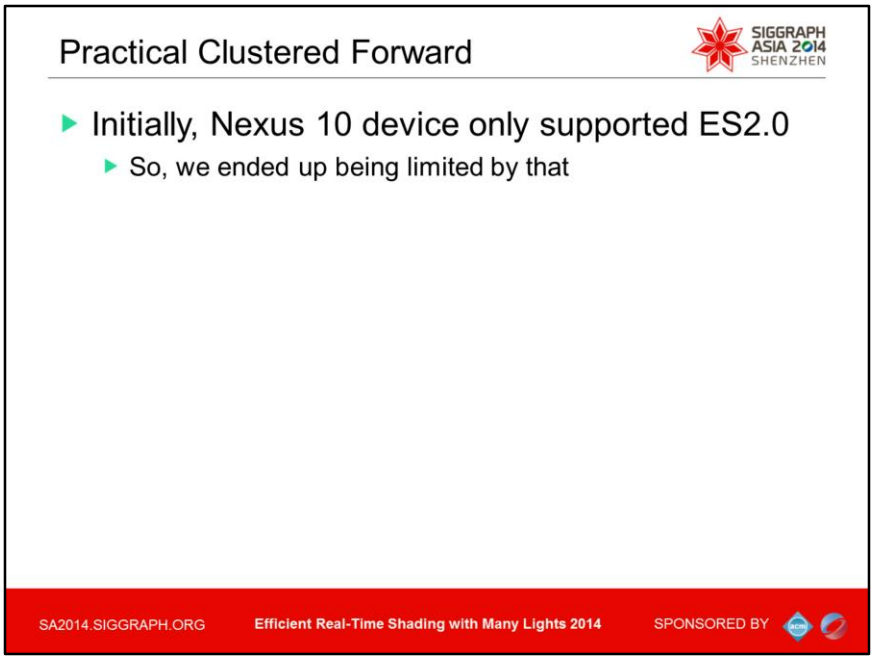
Here's a comparison of all the methods that I've discussed so far.

I'm not going to talk too much about this table right now, we can get back to it if there are any questions.

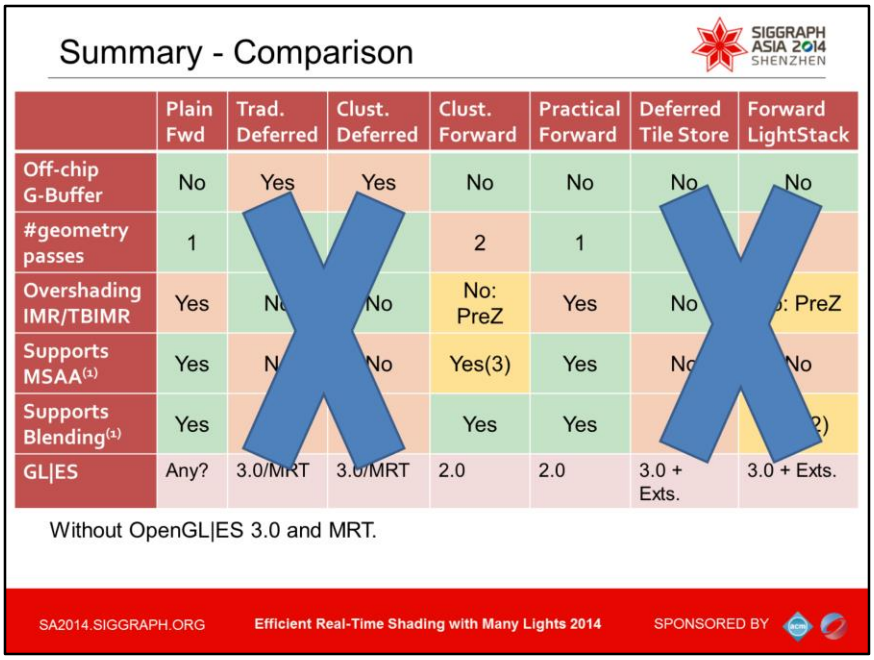


With this, I'm getting to the end of this second part of the presentation. The next part is about a practical clustered forward implantation targeting mobile (Android) devices.

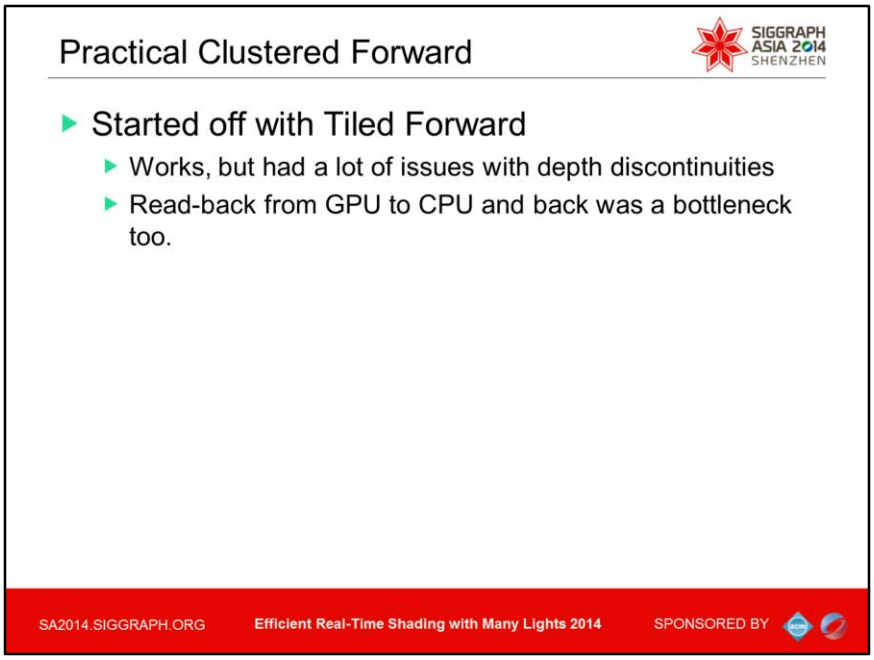
But let's quickly take a step back and look why I ended up choosing practical clustered forward for this.



A part of this is historic – when I started working on mobile rendering, we had a Nexus 10 device as our target. At that time, it only supported OpenGL|ES 2.0.



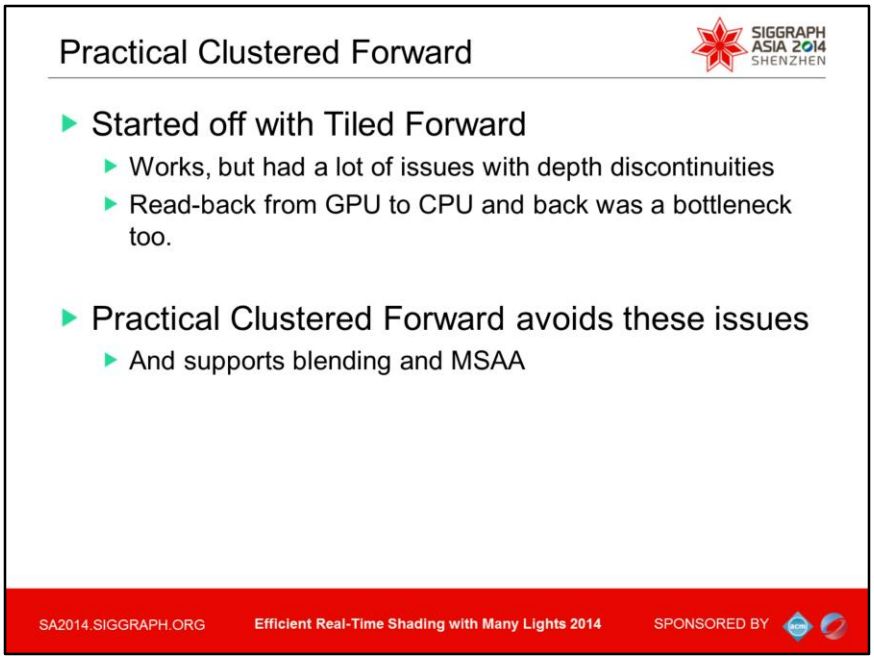
With only OpenGL|ES 2.0, the situation looks like this. Additionally, Emil hadn't presented his practical variation of clustered shading yet, so…



… I started off by implementing a tiled forward renderer.

This definitely works.

However, we had a lot of issues with depth discontinuities. Additionally, the read-back from OpenGL to system memory presented some interesting synchronization issues that would stall the system for long periods of time – and that proved to be somewhat hard to work around.

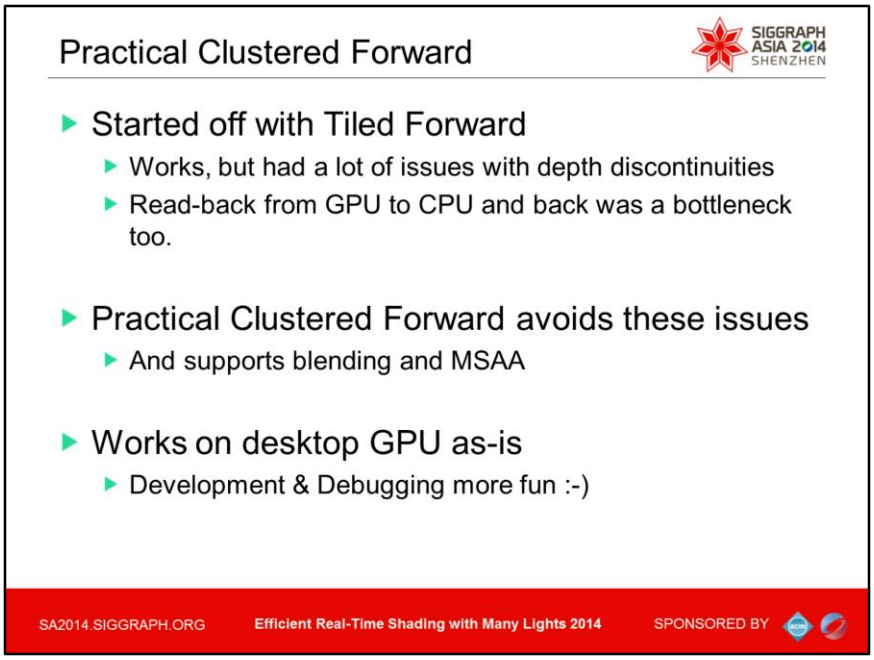


… I started off by implementing a tiled forward renderer.

This definitely works.

However, we had a lot of issues with depth discontinuities. Additionally, the read-back from OpenGL to system memory presented some interesting synchronization issues that would stall the system for long periods of time – and that proved to be somewhat hard to work around.

It turns out that the practical clustered forward method avoids all of these issues, and also supports blending and MSAA nicely. Tiled forward can also support these, but MSAA can be tricky to get right, since we cannot access the unresolved depth buffer.



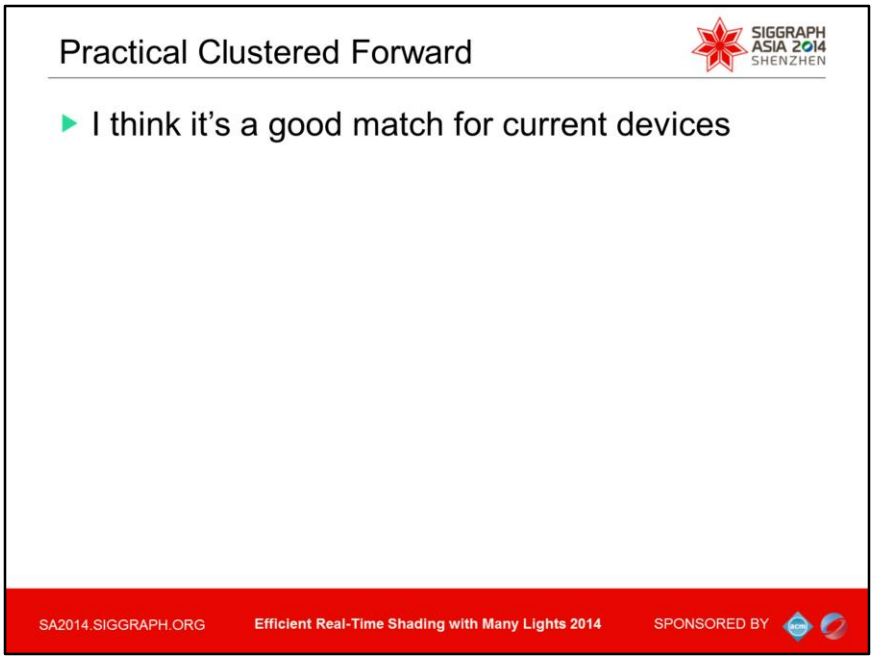
… I started off by implementing a tiled forward renderer.

This definitely works.

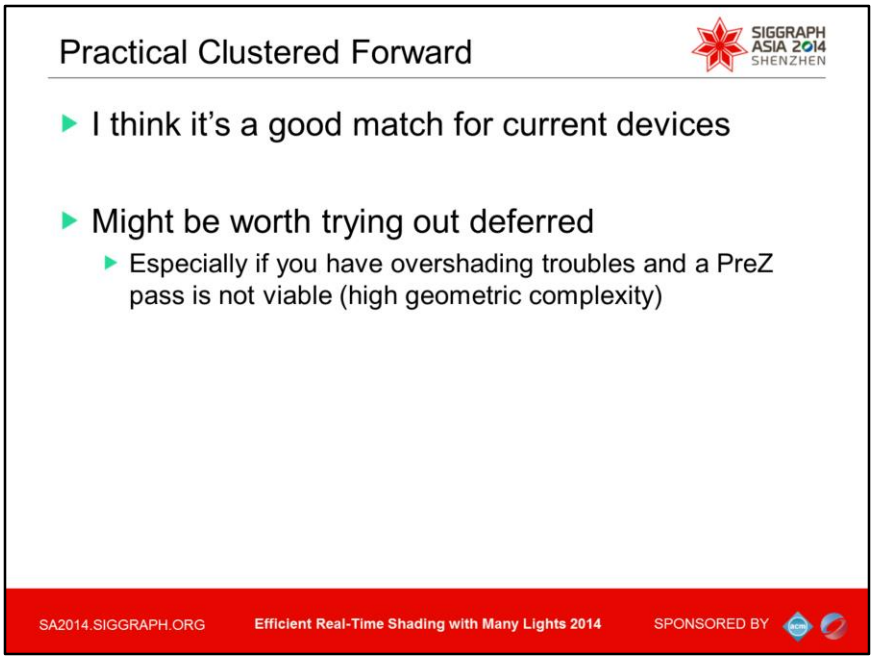
However, we had a lot of issues with depth discontinuities. Additionally, the read-back from OpenGL to system memory presented some interesting synchronization issues that would stall the system for long periods of time – and that proved to be somewhat hard to work around.

It turns out that the practical clustered forward method avoids all of these issues, and also supports blending and MSAA nicely. Tiled forward can also support these, but MSAA can be tricky to get right, since we cannot access the unresolved depth buffer.

As a bonus, the rendering works pretty much as-is on an desktop GPU, which makes development and debugging quite a bit easier. (This is actually true for both the tiled and the clustered methods.)

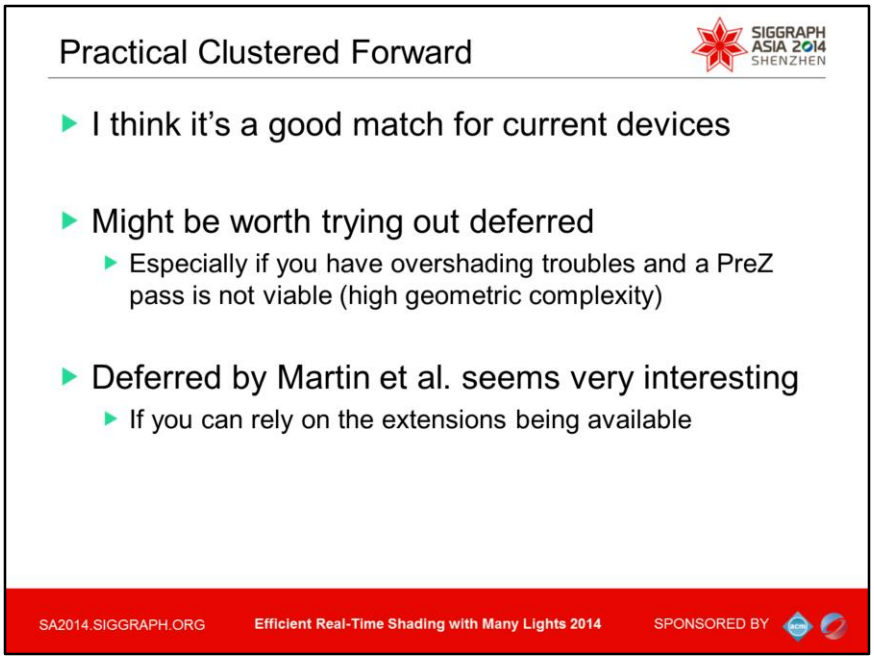


For now, I think the practical clustered forward method is a good match for current and upcoming devices.



For now, I think the practical clustered forward method is a good match for current and upcoming devices.

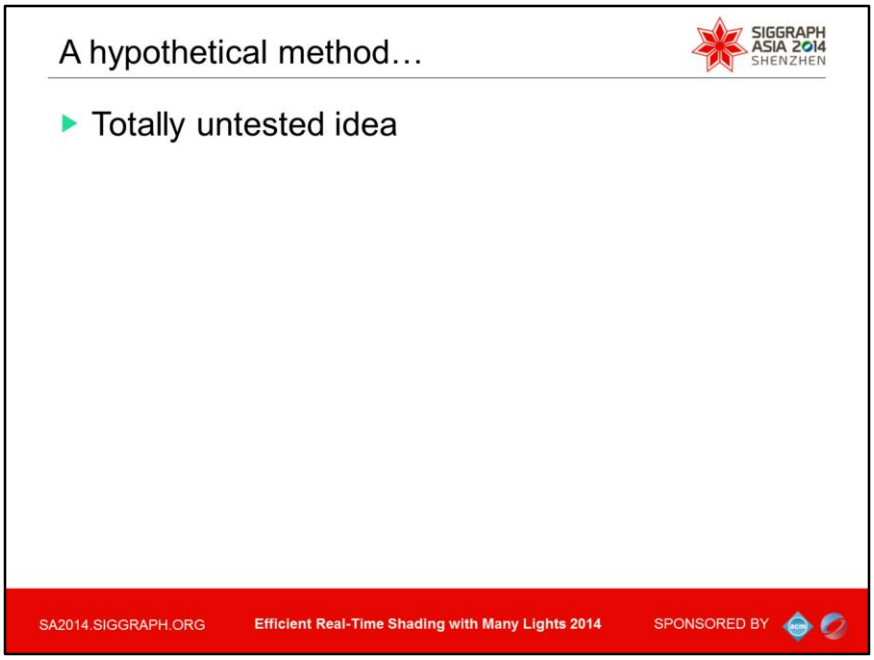
Depending on the situation, it might be worth trying out a deferred variant of this, especially if you have problems with overshading and a depth-only pre-pass is not viable, for instance, because of high geometric complexity.



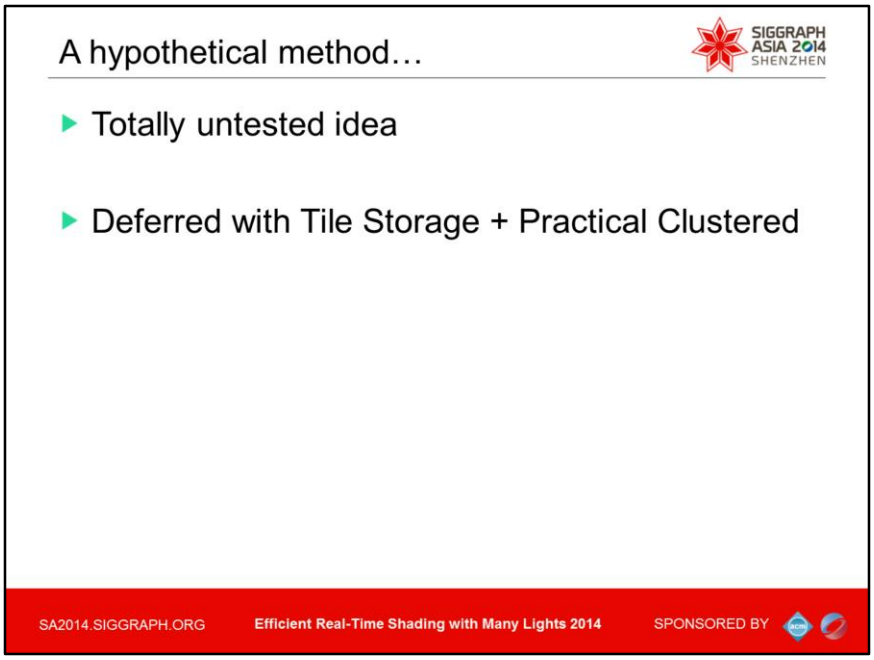
For now, I think the practical clustered forward method is a good match for current and upcoming devices.

Depending on the situation, it might be worth trying out a deferred variant of this, especially if you have problems with overshading and a depth-only pre-pass is not viable, for instance, because of high geometric complexity.

If you can rely on the required extensions being available, the deferred method by Martin et al. also seems very interesting. However, whether or not the extensions ever become available on non-TBR platforms is questionable, so you also might end up having to support multiple rendering pipes.

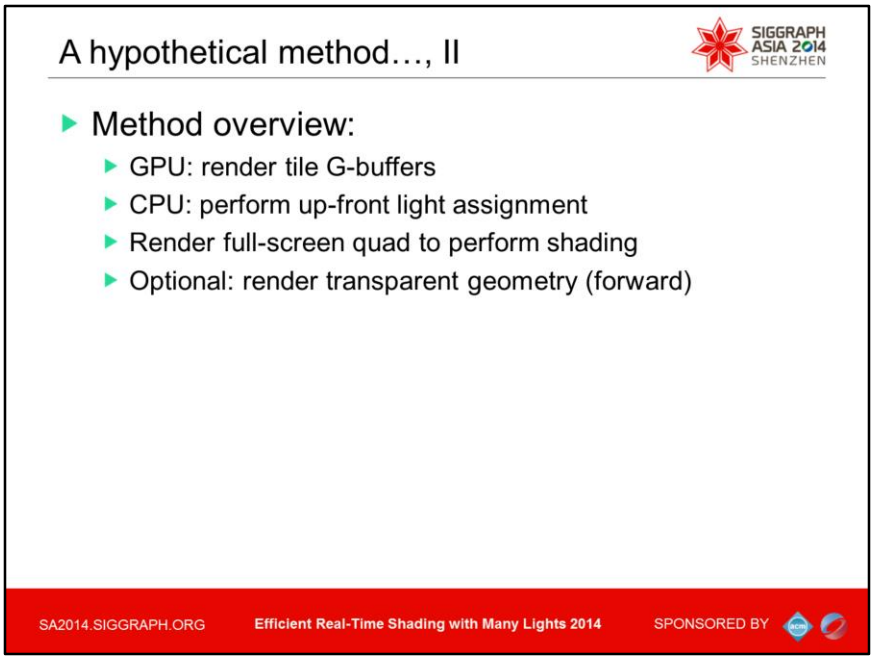


I'm quickly going to squeeze in a totally untested and hypothetical method here – it's also not been presented before, as far as I'm aware.



I'm quickly going to squeeze in a totally untested and hypothetical method here – it's also not been presented before, as far as I'm aware.

The idea is to combine the *Deferred with Tile Storage*-method by Martin et al. with the Practical Clustered shading method.

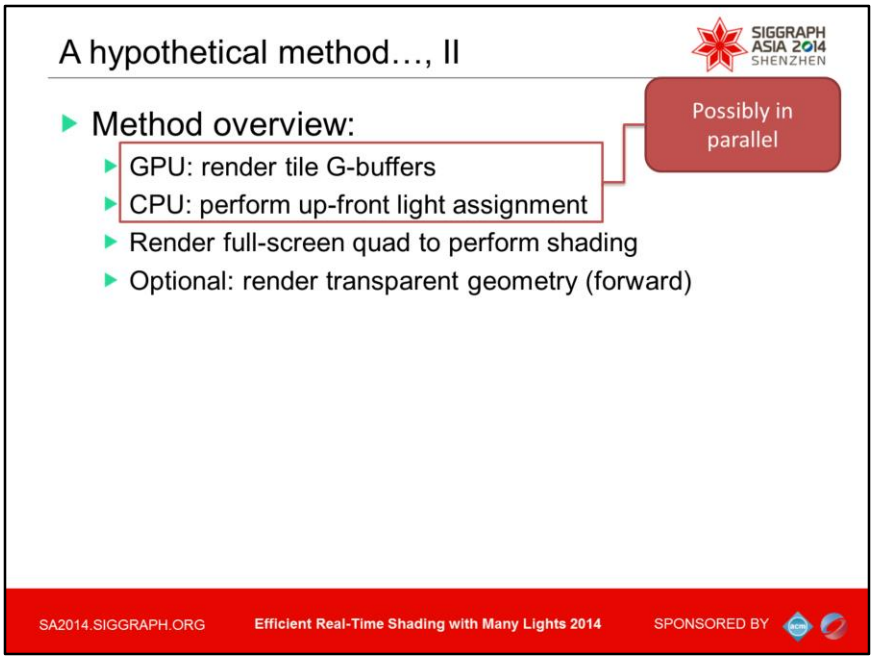


The method looks then roughly like follows:

We start by rendering the geometry to on-chip G-Buffers, like the original in the original method by Martin et al.. Light assignment is performed up-front into a dense grid, as with the practical clustered method.

Then we render a full-screen quad to perform the shading. Each fragment fetches its associated information from the per-pixel tile storage, and then computes which cluster it belongs to so that it can access the per-cluster light list.

Optionally, after the full screen shading pass, we can render transparent geometry in a forward fashion. Here, we can use standard blending, since we do no longer need the information from the per-pixel local storage.



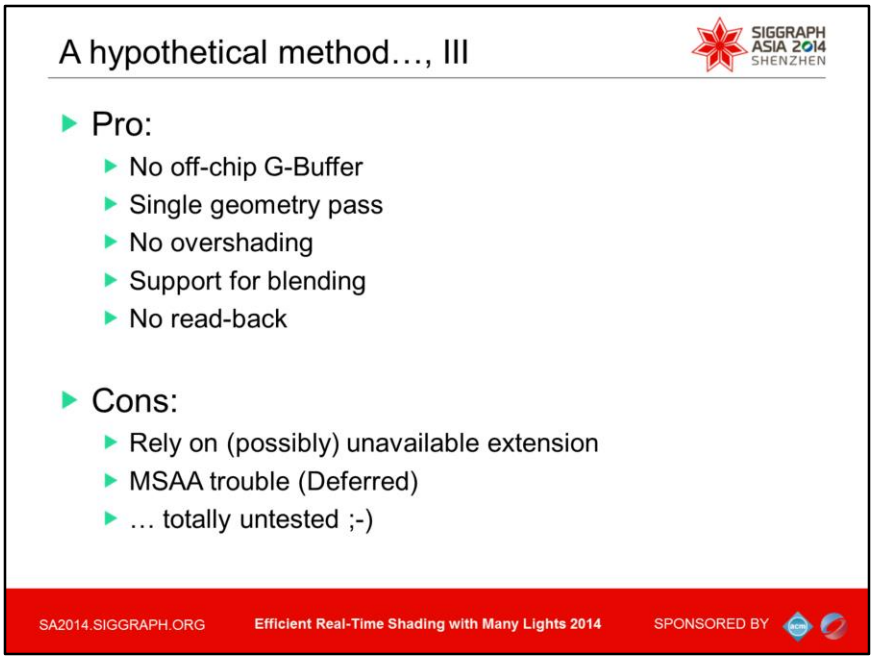
The method looks then roughly like follows:

We start by rendering the geometry to on-chip G-Buffers, like the original in the original method by Martin et al.. Light assignment is performed up-front into a dense grid, as with the practical clustered method.

Then we render a full-screen quad to perform the shading. Each fragment fetches its associated information from the per-pixel tile storage, and then computes which cluster it belongs to so that it can access the per-cluster light list.

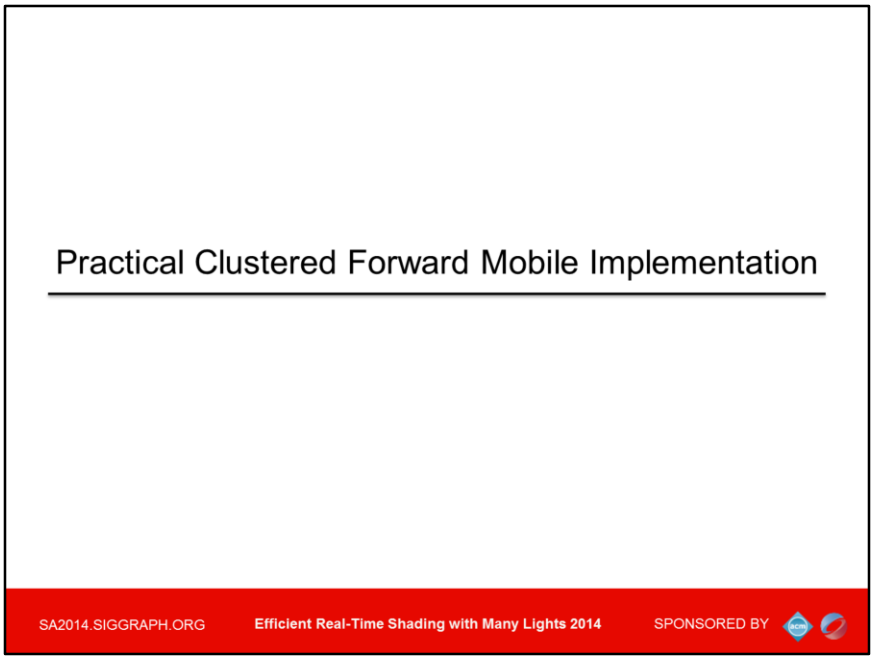
Optionally, after the full screen shading pass, we can render transparent geometry in a forward fashion. Here, we can use standard blending, since we do no longer need the information from the per-pixel local storage.

The first two steps can be performed in parallel, since they are independent form each other. The rendering to G-Buffers doesn't need information about the light sources, and the up-front light assignment doesn't rely on depth-buffers or such.

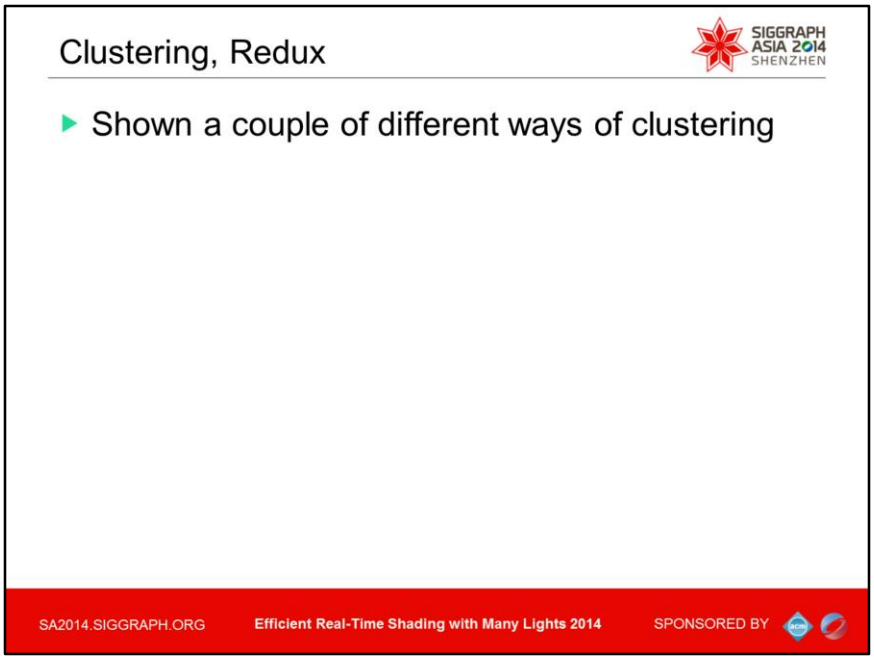


On the up-side, this method wouldn't require off-chip G-Buffers. It only needs a single geometry pass, and there's no overshading. It also supports blending. There are also no read-backs from OpenGL, which could stall rendering.

On the down-side, it still relies on the possibly unavailable extensions for the per-pixel local storage. Being a deferred method, and using the local storage, it's not possible to combine with MSAA. Also, it's totally untested – I'd be curious about trying it out, but currently don't have access to a device with the required extensions.

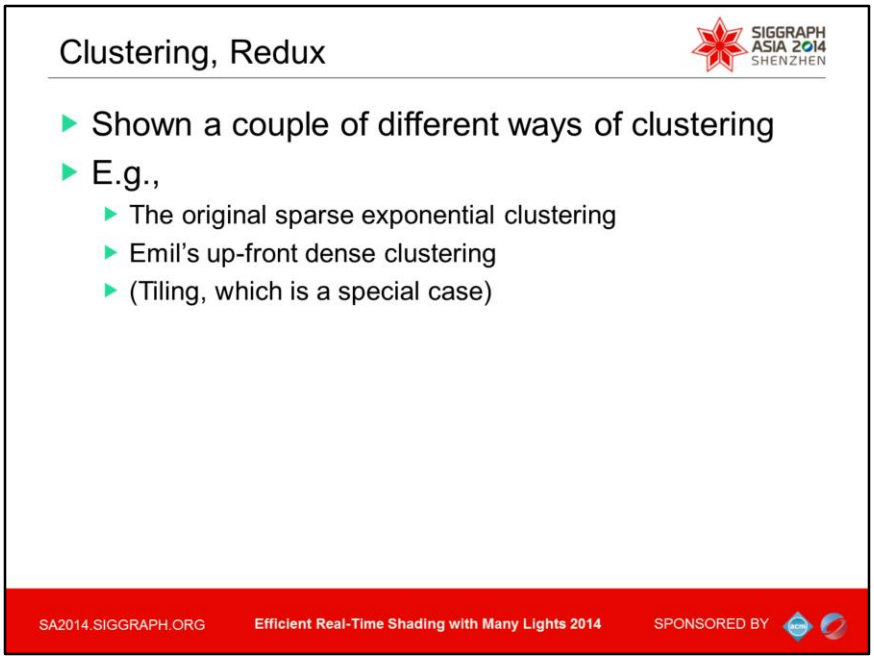


This gets us to the last part of my presentation – the practical clustered forward implementation that targets Android devices.



Compared to the other clustering method that you've heard about so far, I do the clustering slightly differently.

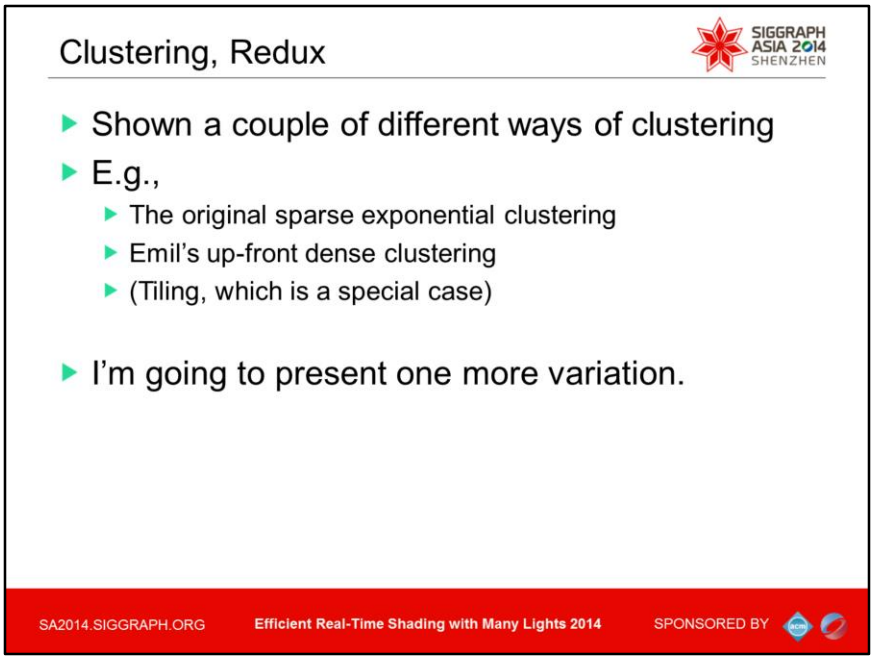
You've seen a couple of different methods already…



Compared to the other clustering method that you've heard about so far, I do the clustering slightly differently.

You've seen a couple of different methods already…

… specifically, we've shown you the original sparse exponential clustering. You've seen Emil's up-front clustering to a dense grid. And we've also discussed tiling, which can be considered a special case of the clustering.

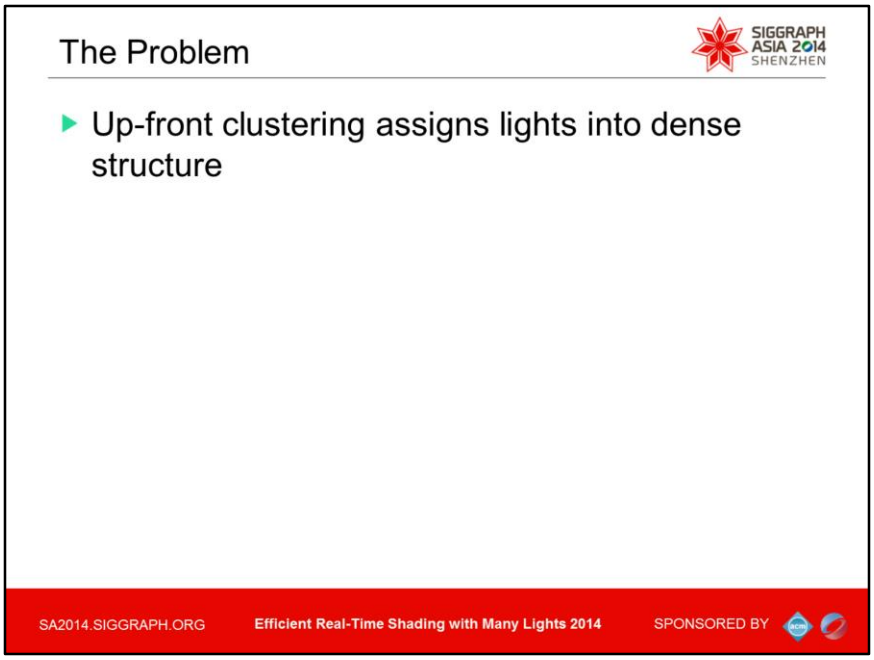


Compared to the other clustering method that you've heard about so far, I do the clustering slightly differently.

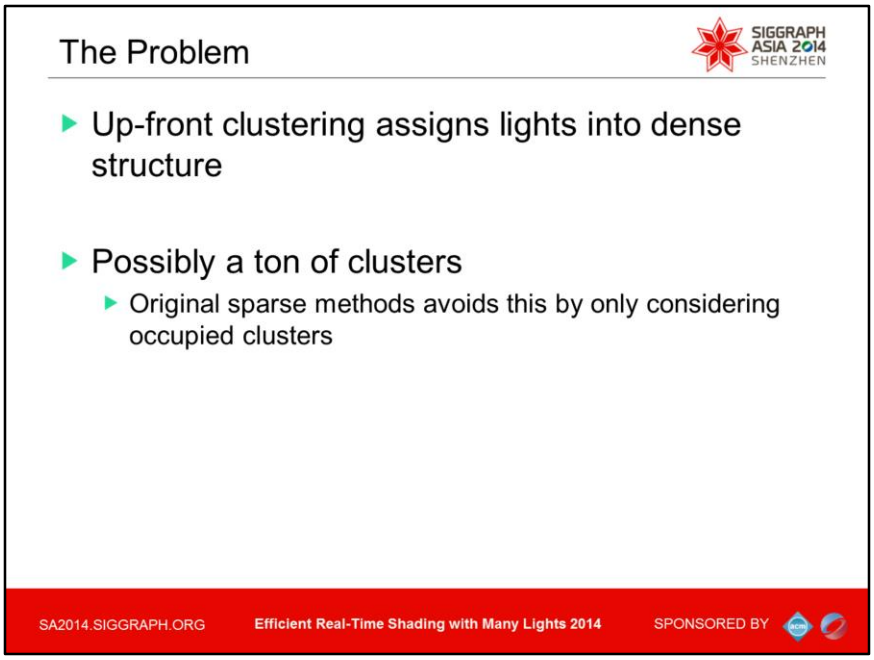
You've seen a couple of different methods already…

… specifically, we've shown you the original sparse exponential clustering. You've seen Emil's up-front clustering to a dense grid. And we've also discussed tiling, which can be considered a special case of the clustering.

I'm going to quickly present one more variation.



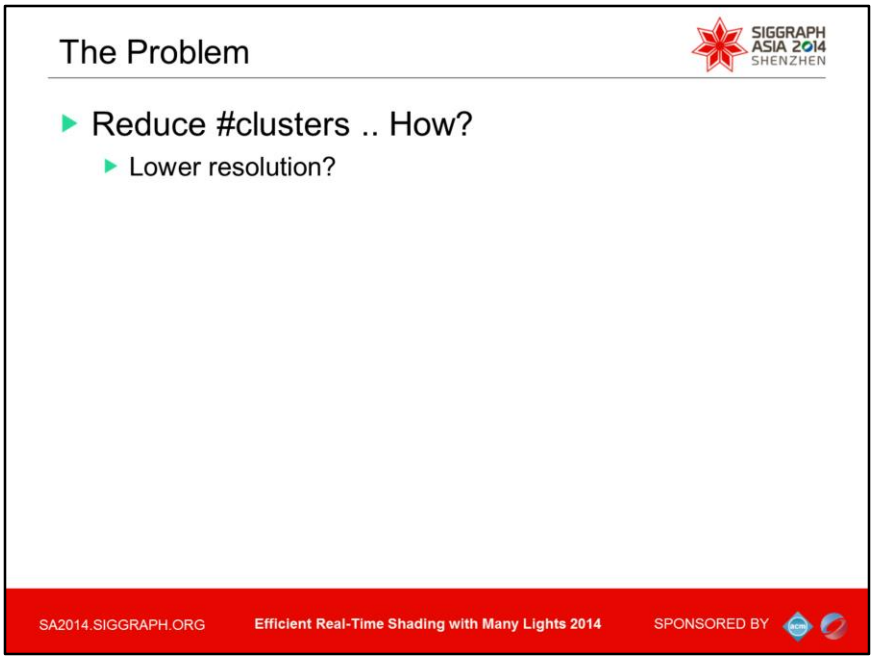
One of the problems with the up-front clustering is that it assigns lights into a dense 3D-grid.



One of the problems with the up-front clustering is that it assigns lights into a dense 3D-grid.

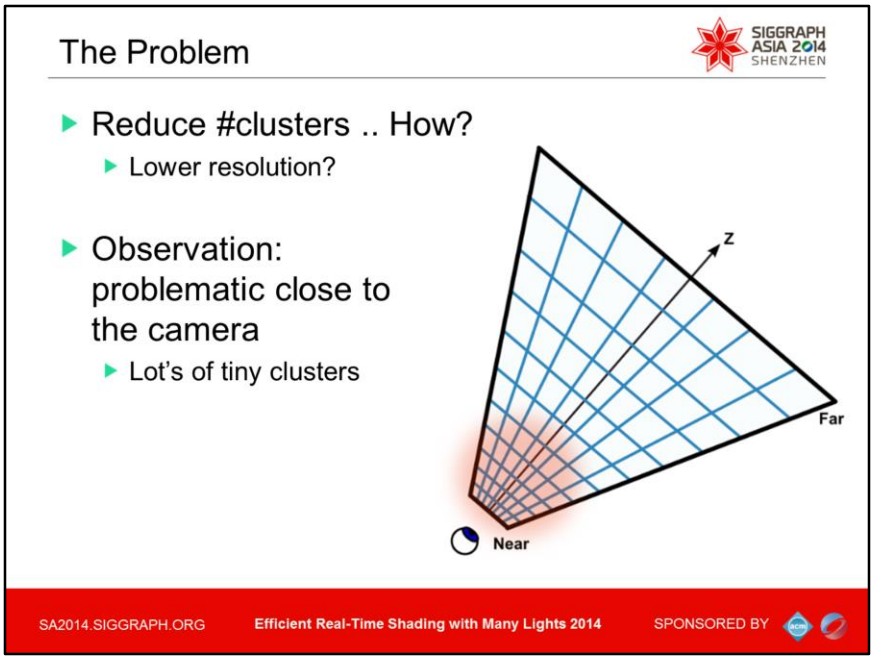
A dense 3D-grid potentially has a lot of cells/clusters. In the original method, we avoid this problem by using a sparse grid, i.e., by only considering clusters that actually contain fragments that need to be shaded.

We can't do this with the up-front variants, since we don't know what clusters we'll end up accessing.



A solution is to reduce the number of cells/clusters. But how should this be done?

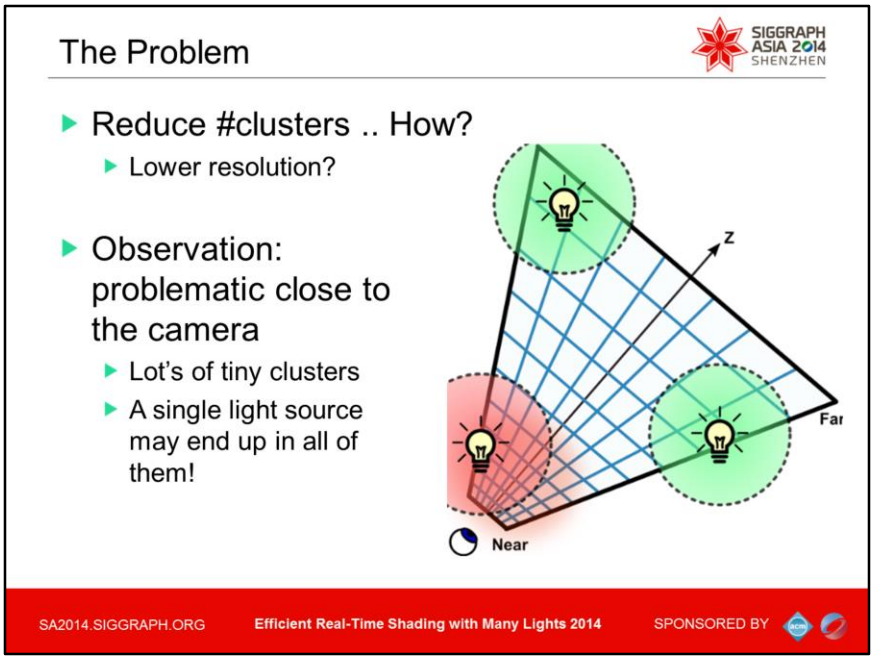
Simply lowering the overall resolution negatively impacts the accuracy of the light assignment, which in turn generates more false positives, and increases the shading costs unnecessarily.



A solution is to reduce the number of cells/clusters. But how should this be done?

Simply lowering the overall resolution negatively impacts the accuracy of the light assignment, which in turn generates more false positives, and increases the shading costs unnecessarily.

An observation at this point is, that this is mainly a problem close to the camera. There's a lot of tiny clusters – highlighted in red – near to the camera.

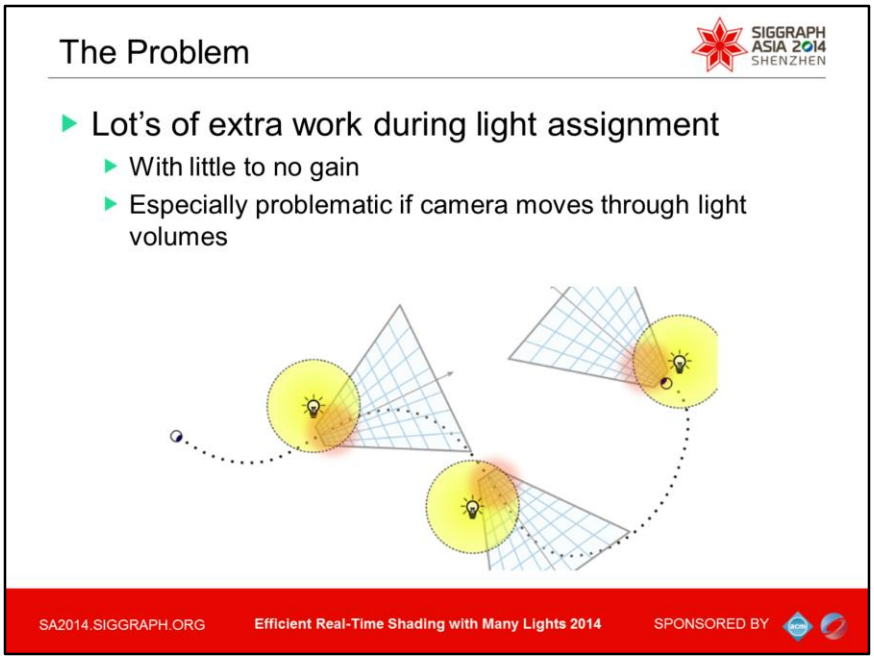


A solution is to reduce the number of cells/clusters. But how should this be done?

Simply lowering the overall resolution negatively impacts the accuracy of the light assignment, which in turn generates more false positives, and increases the shading costs unnecessarily.

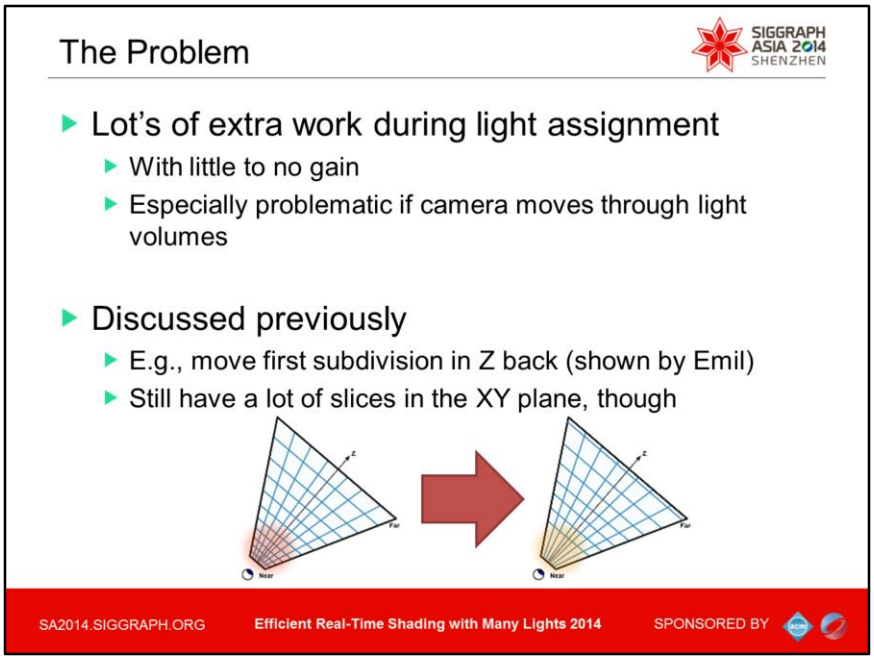
An observation at this point is, that this is mainly a problem close to the camera. There's a lot of tiny clusters – highlighted in red – near to the camera.

Adding a few light sources further illustrates the problem: a light source close to the camera might end up in all those tiny clusters.



Adding light sources to all those tiny clusters is a significant amount of work, from which we do not gain a lot. All the tiny clusters will have almost identical light lists, so there's a lot of redundant information being stored.

This problem is almost guaranteed to happen if we allow the camera to move freely – whenever the camera passes through the light volumes, this will occur.

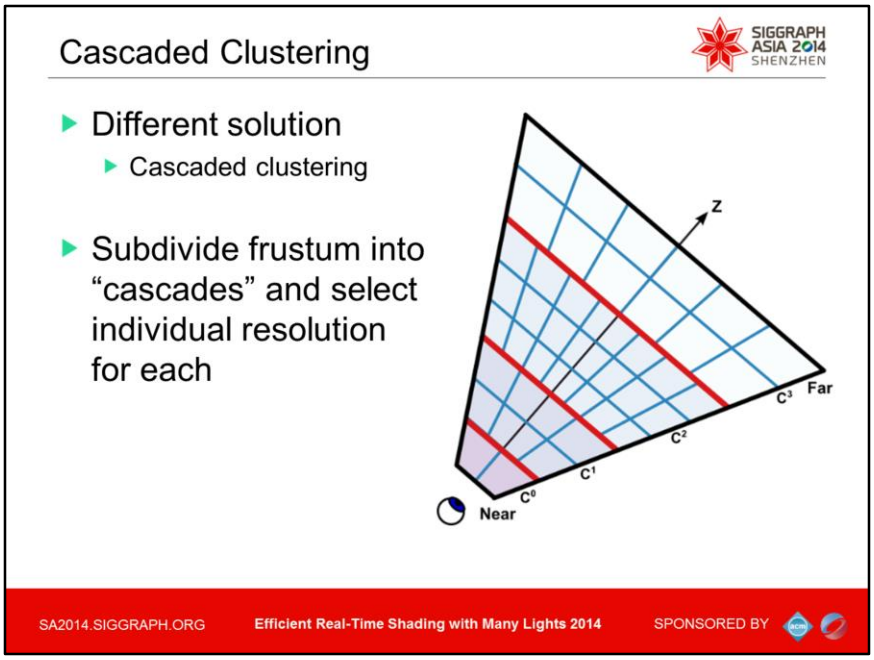


Adding light sources to all those tiny clusters is a significant amount of work, from which we do not gain a lot. All the tiny clusters will have almost identical light lists, so there's a lot of redundant information being stored.

This problem is almost guaranteed to happen if we allow the camera to move freely – whenever the camera passes through the light volumes, this will occur.

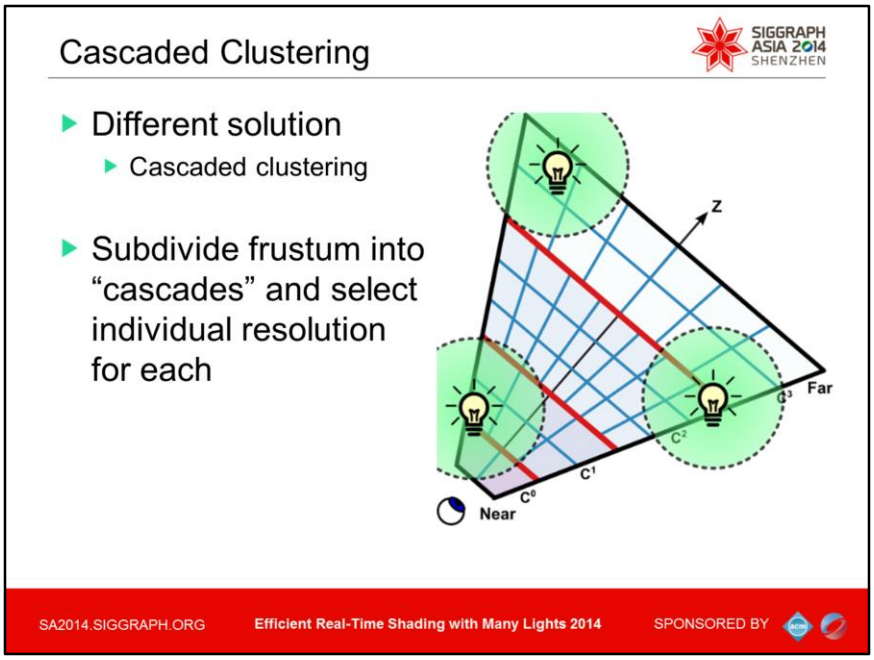
The problem has been mentioned previously by Emil. His solution is to move the first subdivision back in the depth direction. This reduces the number of tiny clusters close to the camera, but it still leaves a lot of slices in the XY direction.

I'm decided to take different approach to solving this problem, to which I'm referring as *"cascaded clustering".*



I'm decided to take different approach to solving this problem, to which I'm referring as "cascaded clustering".

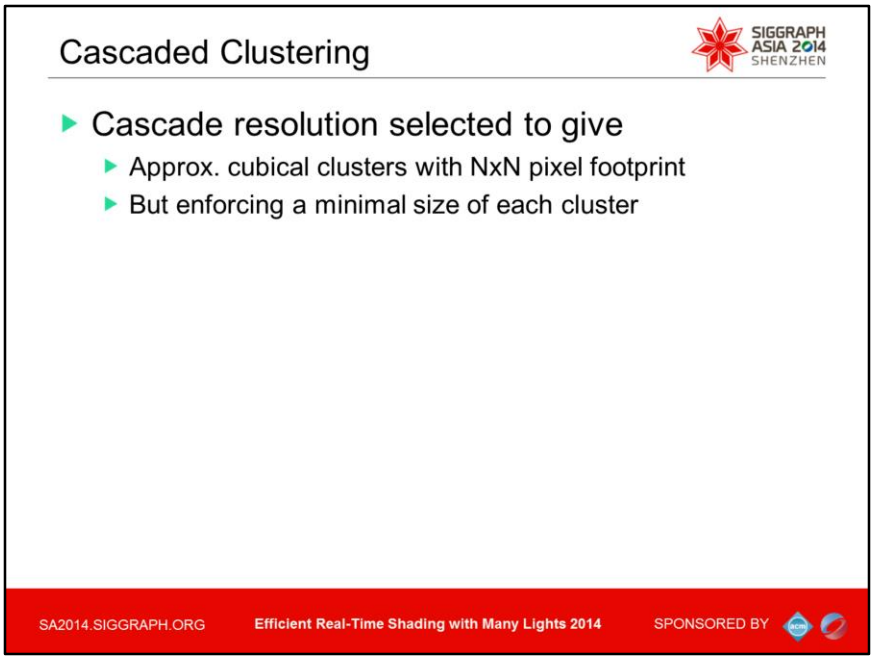
The basic idea is to subdivide the frustum into cascades and select an individual resolution for each cascade. Each cascade functions much like a separate frustum, but with a limited depth range.



I'm decided to take different approach to solving this problem, to which I'm referring as "cascaded clustering".

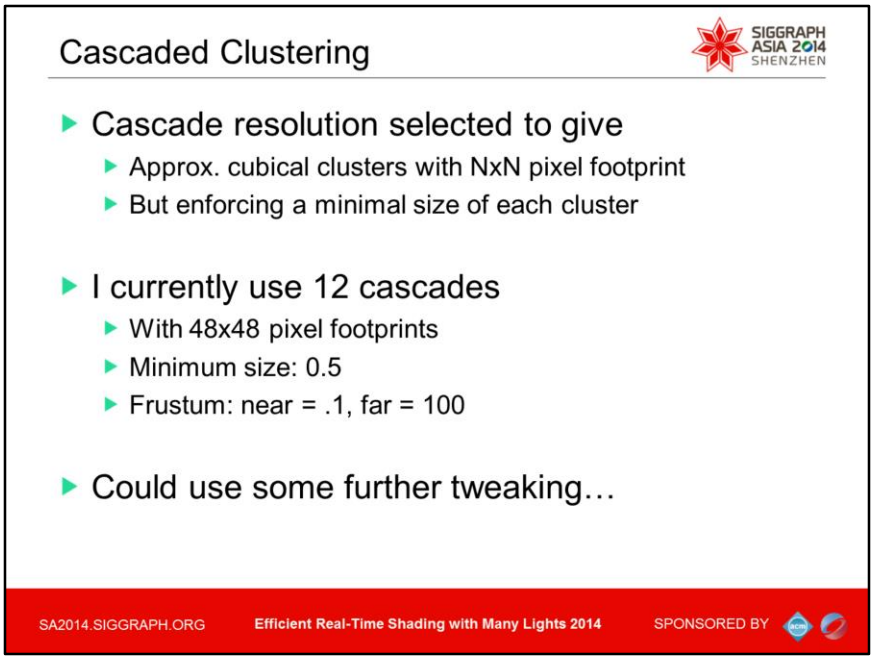
The basic idea is to subdivide the frustum into cascades and select an individual resolution for each cascade. Each cascade functions much like a separate frustum, but with a limited depth range.

To illustrate the situation in this new setup, I've added the three light sources again. Here, each light source ends up overlapping similar amounts of clusters, which is good.



When selecting the resolution for each cascade, I end up using two criteria. First, I again want approximately cubical clusters with an N-by-N pixel footprint, much like our original sparse implementation.

The second criterion is that I do not want clusters that are smaller than a certain world-space size. Whenever the first criterion would result in clusters smaller than this size, I simply clamp the size to this minimum when determining cascade resolution.



When selecting the resolution for each cascade, I end up using two criteria. First, I again want approximately cubical clusters with an N-by-N pixel footprint, much like our original sparse implementation.

The second criterion is that I do not want clusters that are smaller than a certain world-space size. Whenever the first criterion would result in clusters smaller than this size, I simply clamp the size to this minimum when determining cascade resolution.

In my setup, I ended up using 12 cascades for a frustum extending from .1 to 100. I target 48-by-48 pixel footprints and use a minimal size of 0.5 world-space units.

These settings are determined empirically, and could certainly use some additional tweaking – especially if you have scenes that significantly differ from what I experimented with.

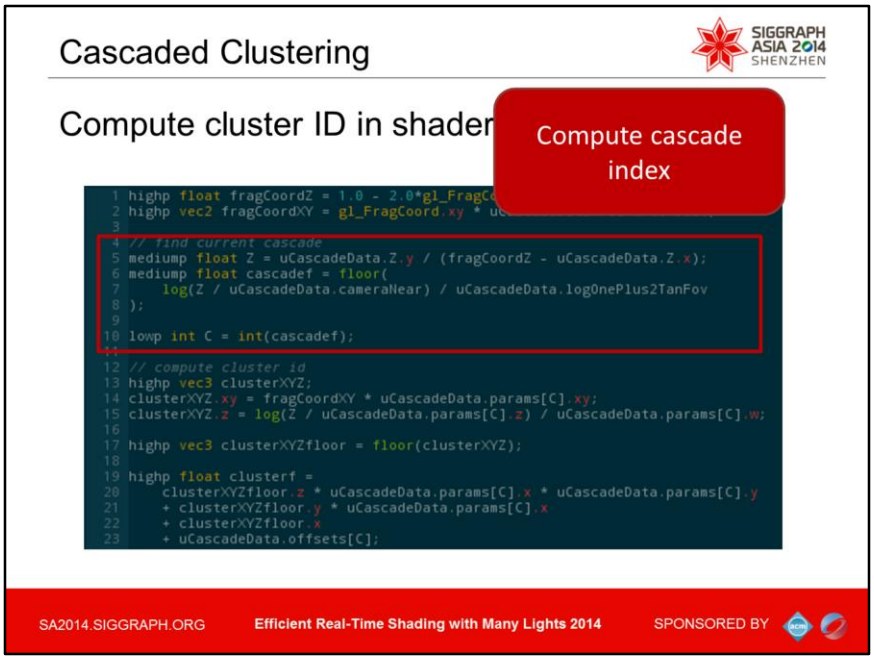
Cascaded Clustering

SIGGRAPH ASIA 2014 SHENZHEN

Compute cluster ID in shader: 2 steps..

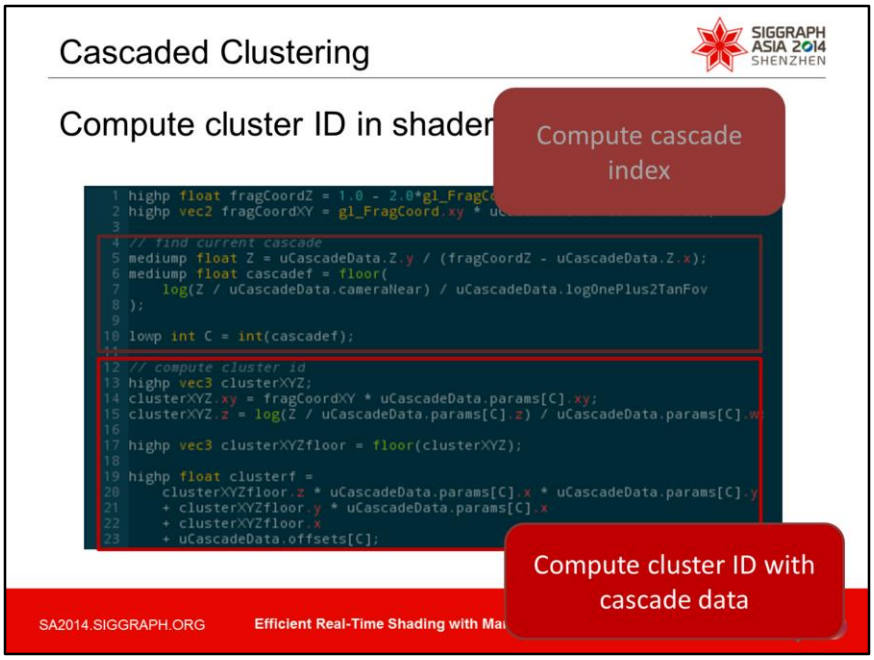
```
highp float fragCoordZ = 1.0 - 2.0*gl_FragCoord.z;
highp vec2 fragCoordXY = gl_FragCoord.xy * uCascadeData.recWindowSize;

// find current cascade
mediump float Z = uCascadeData.Z.y / (fragCoordZ - uCascadeData.Z.x);
mediump float cascadef = floor(
    log(Z / uCascadeData.cameraNear) / uCascadeData.logOnePlus2TanFov
);

lowp int C = int(cascadef);

// compute cluster id
highp vec3 clusterXYZ;
clusterXYZ.xy = fragCoordXY * uCascadeData.params[C].xy;
clusterXYZ.z = log(Z / uCascadeData.params[C].z) / uCascadeData.params[C].w;

highp vec3 clusterXYZfloor = floor(clusterXYZ);

highp float clusterf =
    clusterXYZfloor.z * uCascadeData.params[C].x * uCascadeData.params[C].y
    + clusterXYZfloor.y * uCascadeData.params[C].x
    + clusterXYZfloor.x
    + uCascadeData.offsets[C];
```


Computing the cluster ID now requires two steps.



Computing the cluster ID now requires two steps.

First, given the fragment's depth, the cascade index can be computed.

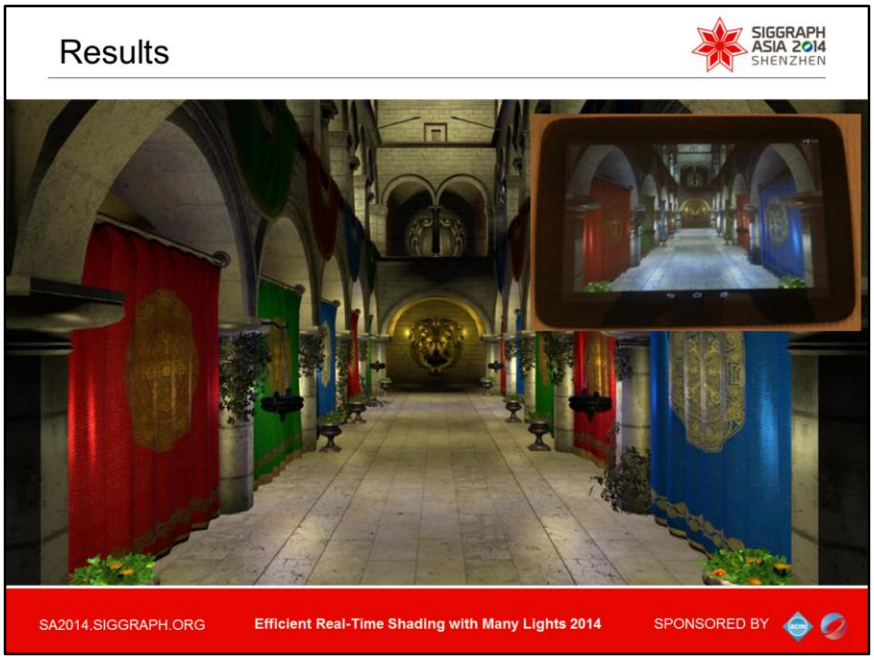


Computing the cluster ID now requires two steps.

First, given the fragment's depth, the cascade index can be computed.

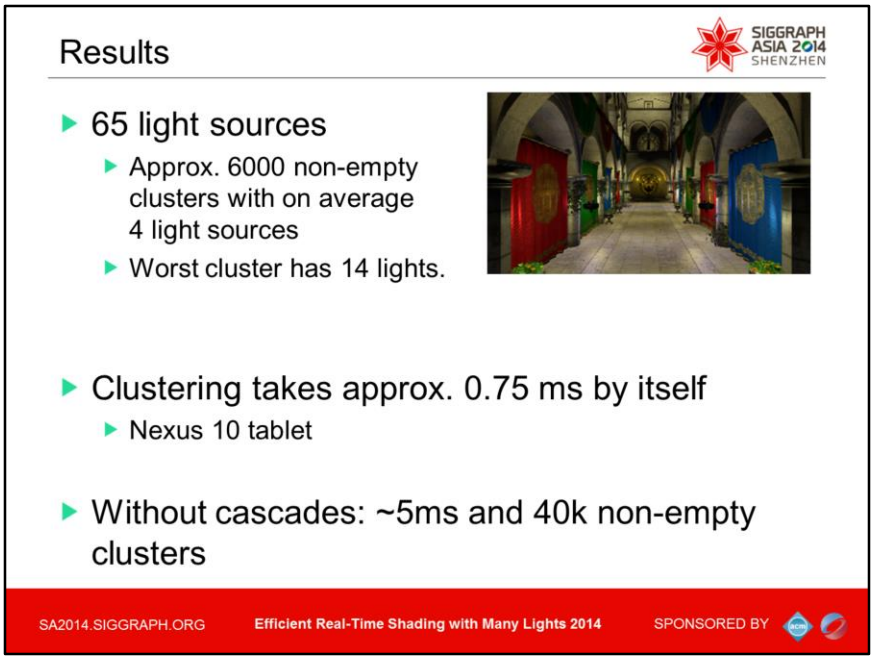
Using the cascade index, the properties of that cascade are retrieved. Using the cascade properties, we can find the cluster's ID in that cascade.

This second computation is largely equivalent to computing the cluster ID in our original method and to computing the cluster ID in Emil's practical method, so really only this first computation is new, additional code that needs to be executed in the fragment shader.



Let’s quickly look at some results.

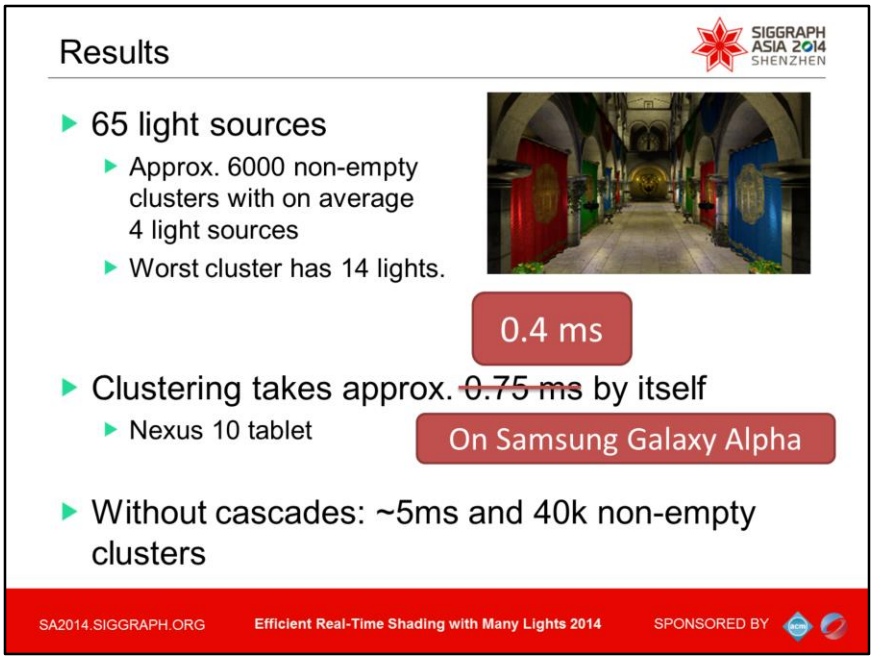
You’ve already seen this scene once, in Ola’s previous presentation. This time, I’m running it on a mobile device, as shown in the top right, albeit without the shadows.



The scene contains 65 light sources, and, using the cascaded clustering, it results in approximately 6000 non-empty clusters with on average 4 light sources. The worst case cluster contains 14 lights, however.

Computing the clustering takes approximately 0.75 ms on the an Nexus 10 device.

Performing the clustering without the cascades would take approximately 5 ms, so it's quite a bit more expensive. It results in approximately 40k non-empty clusters, without significantly improving the light assignment.

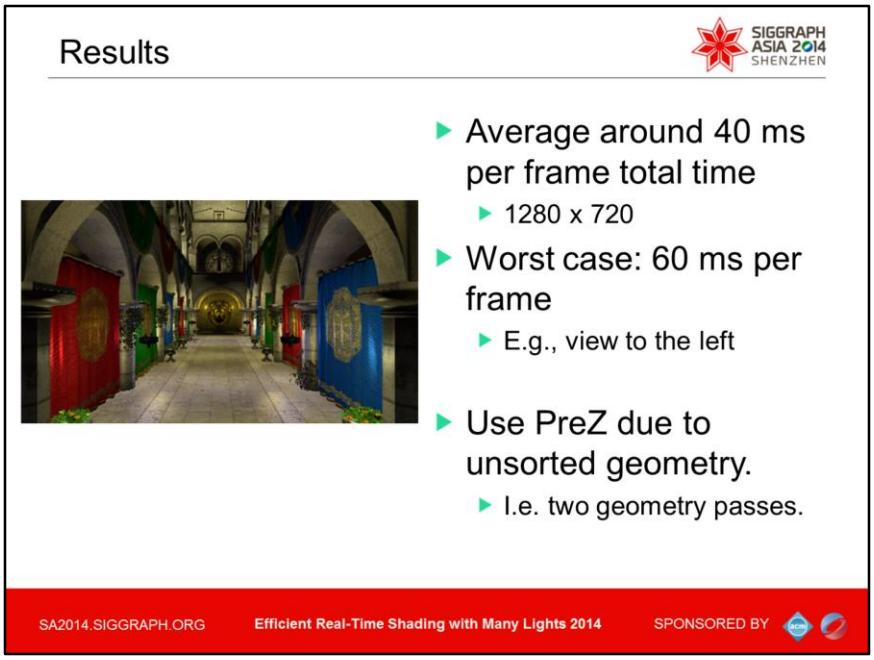


The scene contains 65 light sources, and, using the cascaded clustering, it results in approximately 6000 non-empty clusters with on average 4 light sources. The worst case cluster contains 14 lights, however.

Computing the clustering takes approximately 0.75 ms on the an Nexus 10 device.

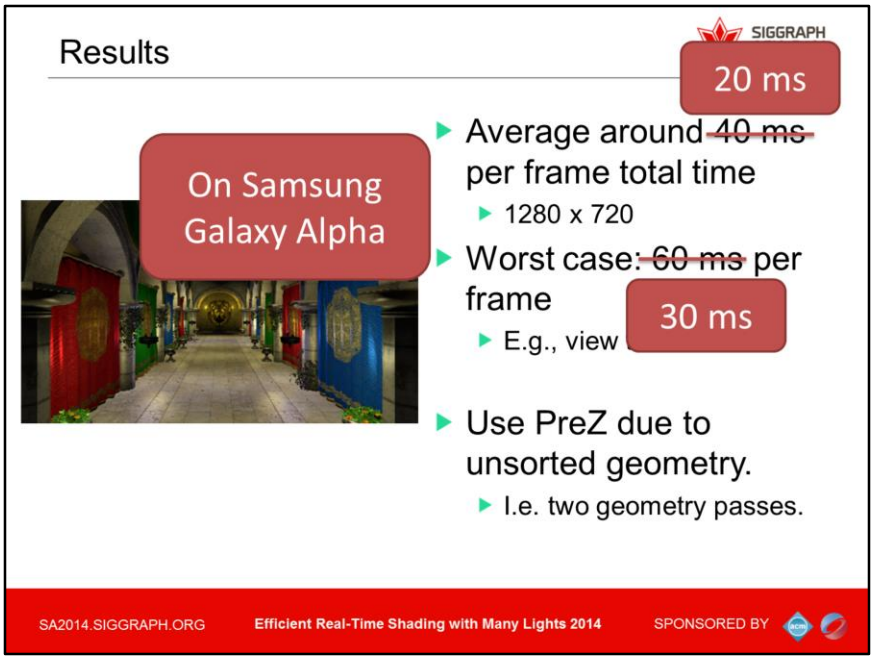
Performing the clustering without the cascades would take approximately 5 ms, so it's quite a bit more expensive. It results in approximately 40k non-empty clusters, without significantly improving the light assignment.

A few days ago, I finally tried this out on my Samsung Galaxy Alpha device, which is a bit newer, and sports a bit nicer hardware. On that device, the clustering takes 0.4 ms.



On the Nexus 10, rendering takes around 40 ms per frame, at a 720p resolution. In the worst case, it takes a bit longer – 60 ms. A view close to the worst case is illustrated to the left. Here, almost the whole scene is visible, with several large overlapping lights close to the viewer, and thereby affecting much of the screen.

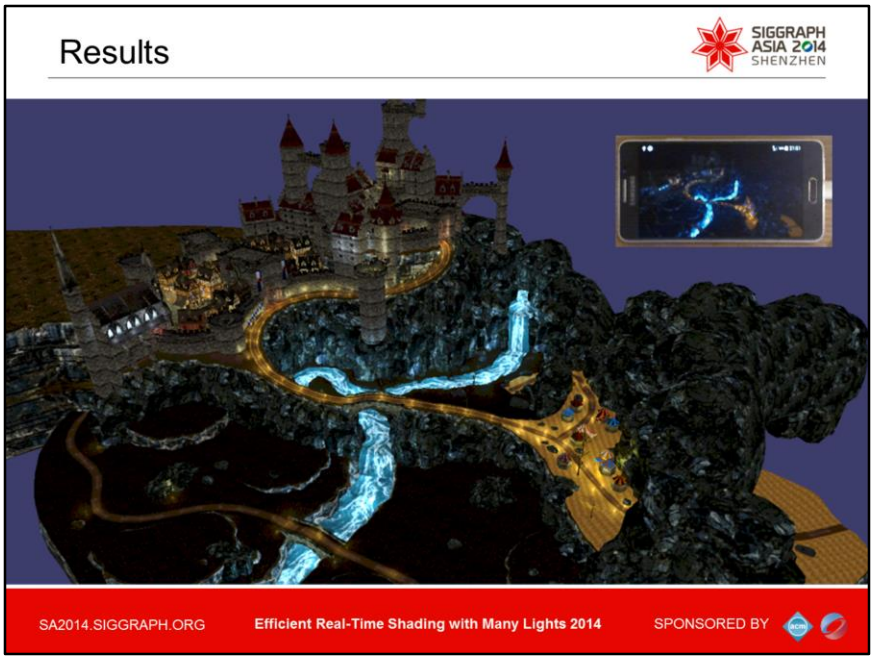
Also, I'm not submitting the geometry front-to-back, since I'm a bad person. I use a depth-only pre-pass to prime the Z-buffer and avoid overshading. In this scene, it turns out to be worth the extra geometry pass.



On the Nexus 10, rendering takes around 40 ms per frame, at a 720p resolution. In the worst case, it takes a bit longer – 60 ms. A view close to the worst case is illustrated to the left. Here, almost the whole scene is visible, with several large overlapping lights close to the viewer, and thereby affecting much of the screen.

Also, I'm not submitting the geometry front-to-back, since I'm a bad person. I use a depth-only pre-pass to prime the Z-buffer and avoid overshading. In this scene, it turns out to be worth the extra geometry pass.

On the Galaxy Alpha device performance improves again. Rendering takes 20 ms on average per frame, and the worst case is now around 30 ms.

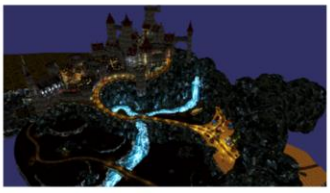


The second scene is the Epic Citadel, extracted from the Unreal SDK.

Results

SIGGRAPH ASIA 2014 SHENZHEN

- 192 light sources
  - < 3500 non-empty clusters with on average 3.5 light sources
  - Worst cluster has 20 lights.



- Rather view-dependent (large-ish scene)
  - Some views have ~500 active clusters only
- Clustering takes <0.35 ms by itself
  - Samsung Galaxy Alpha

Here, I placed 192 light sources – in the last two evenings, by hand, so I'm now very much aware how ridiculous 1M of hand-placed light sources would be.

Anyway, in this scene, I typically end up with less than 3.5 thousand non-empty clusters that contain 3.5 light sources on average. The worst case here is around 20 light sources in a single cluster.

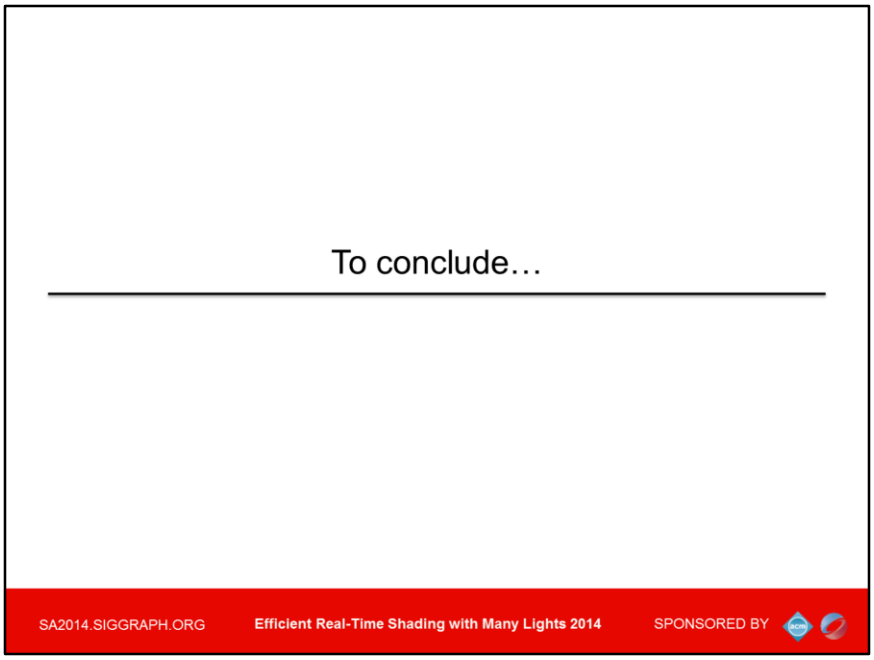
This scene is a bit different, in the sense that lights have much less overlap when compared to the lights in the Sponza scene shown earlier. The numbers presented are also rather view dependent, but close to the worst case.

The clustering typically takes less than 0.35 ms on the Galaxy Alpha device. The faster clustering is again due to the smaller light sources that end up in fewer clusters.

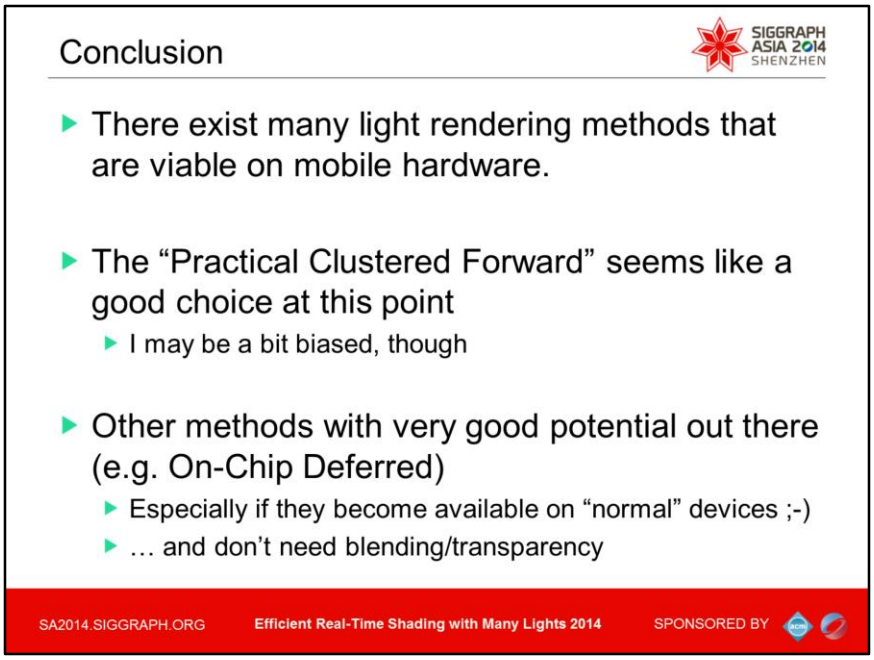
Results

- Average around 30 ms per frame total time
  - 1280 x 720
- Worst case: ~35 ms per frame
  - Limited overlap between lights
- Again, with PreZ

Rendering takes around 30ms, again at 720p resolution. The worst case is 35ms. I again use a depth-only pre-pass to prime the depth buffer.



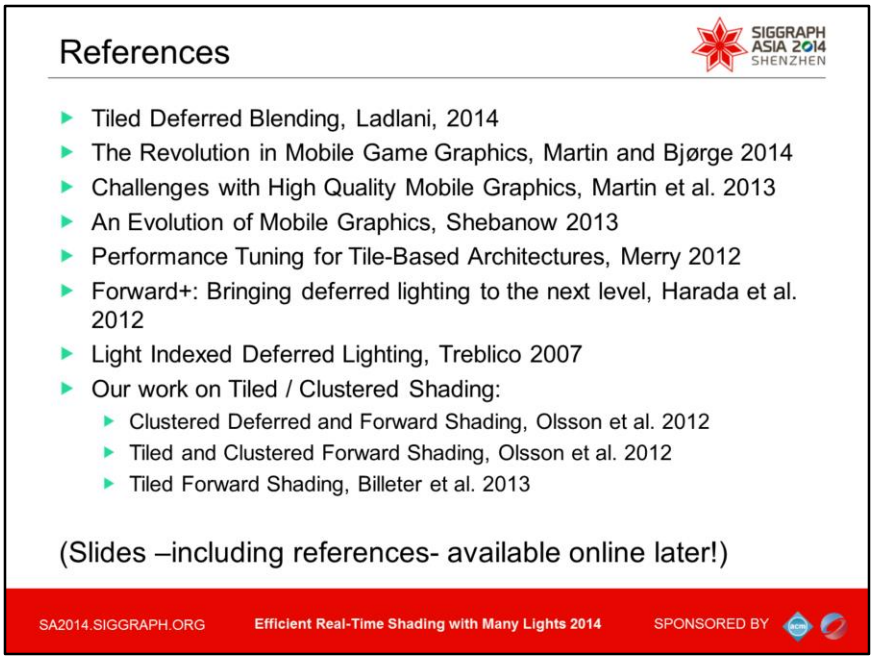
With this, I'm almost done. Let's conclude with a short summary of the most important points.



There’s a few many-light rendering methods that are viable on mobile hardware.

To me, the practical clustered forward method seems like a good choice at this point – it’s implementable on a wide variety of devices and doesn’t have a lot of special requirements. Disclaimer: I’m a bit biased, though.

Other very interesting methods include e.g., the On-Chip Deferred method by Martin et al. – this might be a good choice as the extensions become available on consumer devices (and you don’t need blending).



Finally, here's the list of references. We'll make the slides, including this reference list, available online at some point soon.