
SIGGRAPH2015
Xroads of Discovery

SIGGRAPH2015
Xroads of Discovery

The 42nd International Conference and Exhibition on Computer Graphics and Interactive Techniques

VISUALIZATION AND MULTIMEDIA LAB

University of Zurich[UZH]

Department of Informatics

DIVA

Part 4:
Many-Light Rendering on Mobile Hardware
40 min

Markus Billeter
VMML • University of Zürich

Many-Light on Mobiles

- Outline

Hello & welcome to this last part about many light rendering on mobile hardware, i.e. on devices like smart phones and tablets. ▶▶▶

Many-Light on Mobiles

- Outline
  - What's different?

<contd.>

My presentation consists of three different parts: first, I'm going to start with a short introduction about mobile hardware in general, and point out some the differences and limitations compared to the high-end systems that this course has focused on so far. ▶▶▶

Many-Light on Mobiles

- Outline
  - What's different?
  - Review with a twist towards Mobile HW

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

VISUALIZATION AND MULTIMEDIA LAB

<contd.>

Next, there's a review of different many light rendering techniques. If you've been here since the beginning, you're already familiar with quite a few of them – besides reviewing each method quickly one more time, I'm also going include some notes regarding their suitability for running on mobile hardware. If you decided to sleep in, you're in luck too: this part includes very short summaries of parts of Ola's material from earlier this morning. ▶▶▶

Many-Light on Mobiles

- Outline
  - What's different?
  - Review with a twist towards Mobile HW
  - Case study: Clustered Implementation

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

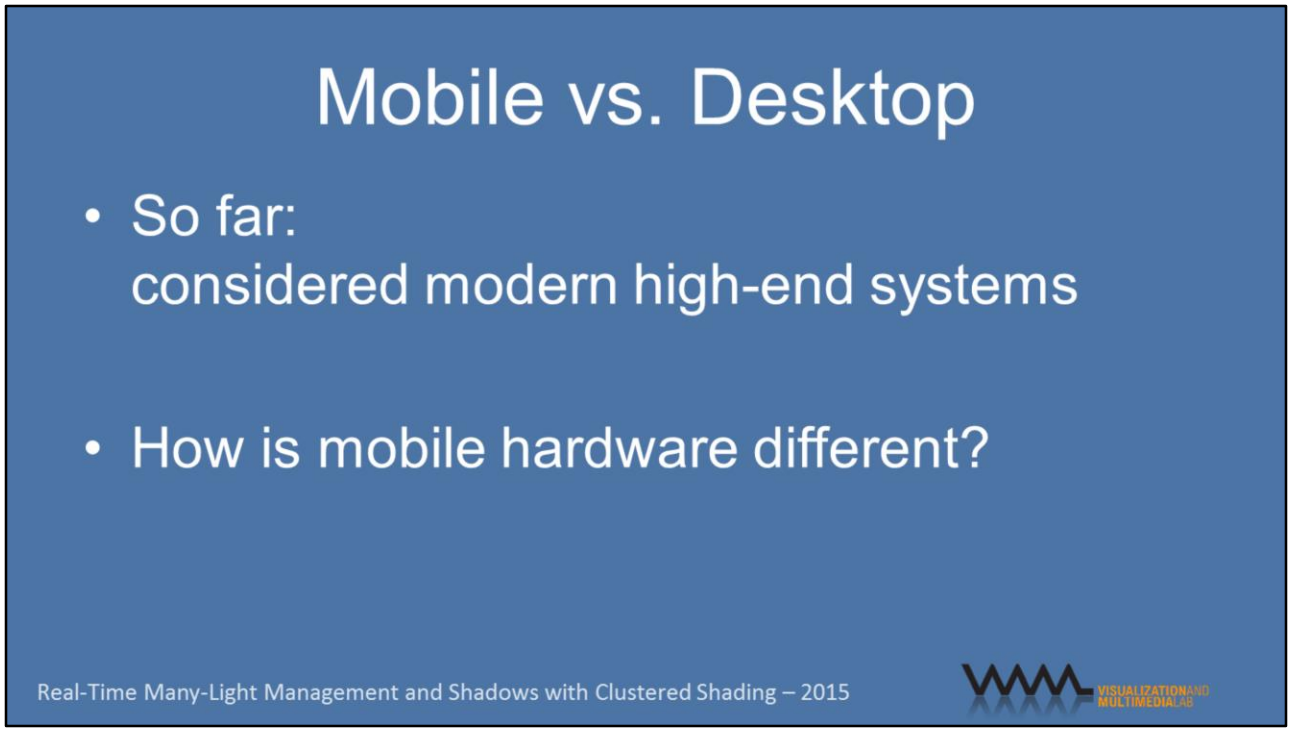
VISUALIZATION AND MULTIMEDIA LAB

<contd.>

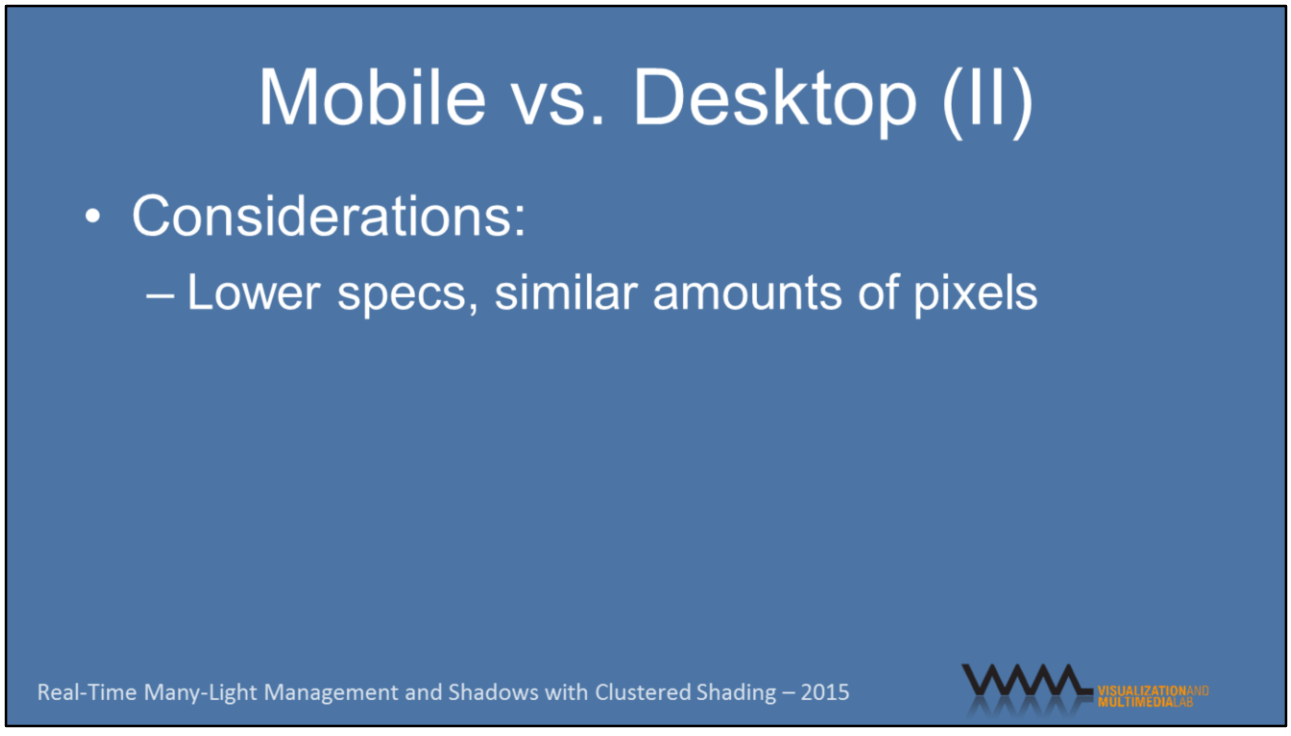
Towards the end, I'm going to talk about a clustered implementation that I developed with smart phones and tablets in mind. It performs a slightly different trade off compared to other methods you've seen today. Even though it's been developed with mobile HW in mind, it could be interesting in other areas as well. One of the secondary goals with this is to somewhat show the flexibility of the clustered-shading idea, where you can mix up the method to really match your specific use case.

Part 1

MOBILE HARDWARE

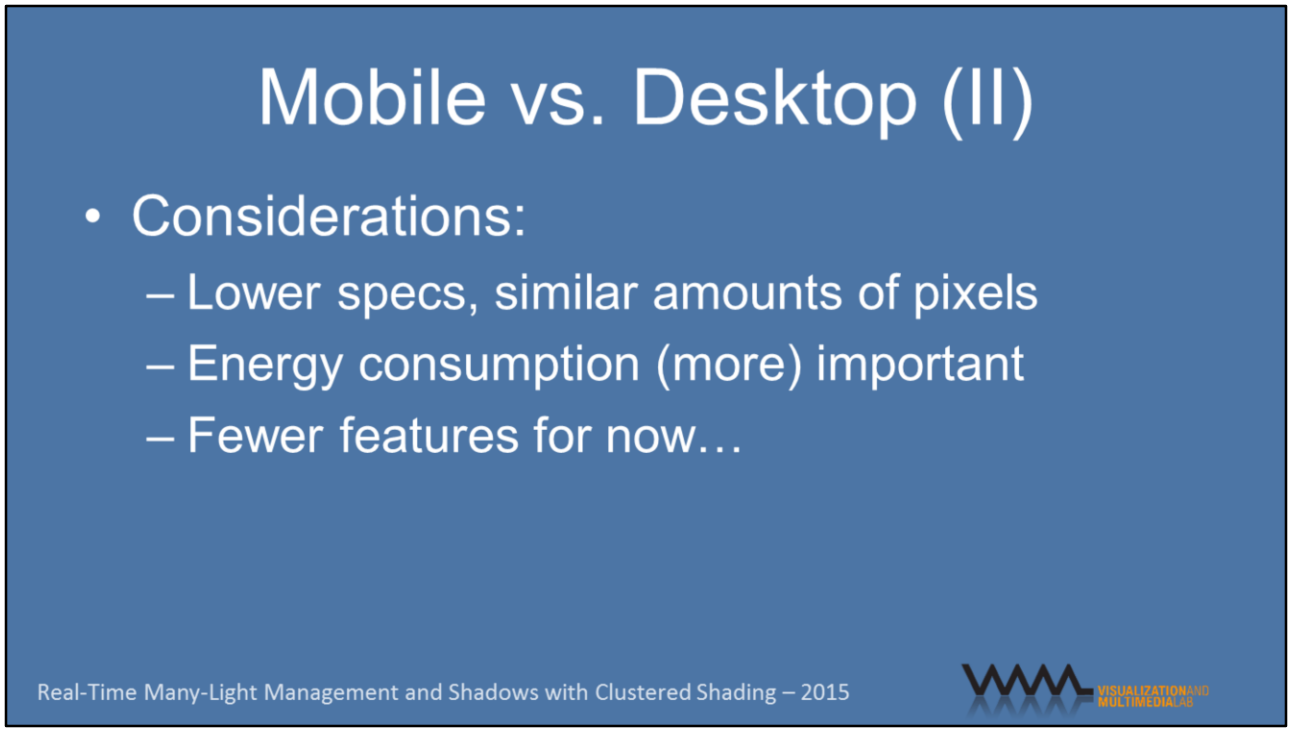


As already mentioned once, this course has mainly considered modern high-end systems – that is, dedicated desktop-class GPUs and perhaps game consoles. Now that we're looking at mobile hardware… what are the differences and challenges we have to deal with?



If we compare the absolute specs, that is e.g., the absolute computational power and memory bandwidth that's available, I don't think it's too surprising to find that mobile hardware clocks in quite a bit lower than the high-end desktop: we get about one order of magnitude less of both computational power and memory bandwidth on average.

The computational work load we have some amount of control over, we can reduce it by for example considering fewer lights per pixel during shading. Memory bandwidth is a bit trickier, so it's important to look for methods that conserve bandwidth. Furthermore, bandwidth is expected to improve more slowly than computational power. ▶▶▶



<contd.>

When talking about mobile hardware, it's hard to avoid the topic of energy consumption. If it's possible, reducing energy consumption from the software side of things is definitively something worth considering. Not only for the improved battery life, but also to avoid running into thermal limits that might cause the device to run at lower performance. Fortunately, our goal to conserve memory bandwidth helps us here, since memory transactions are quite power hungry – earlier this week, I saw a figure citing up to 16% of the total energy consumption being attributed to memory.

Finally, I'd still claim that mobile hardware lags behind in terms of features, at least if we're considering the devices currently out there. With this said, however, the gap seems to be closing quickly. Regardless, I'll show some numbers about this.

A few numbers… (I)

- Android OpenGL Version:

OpenGL ES Version	Percentage
2.0	63%
3.0	35%
3.1	2%

Data from June 2015
Source: http://developer.android.com/about/dashboards/index.html

This first table shows the support for different OpenGL ES versions as seen on “active” Android devices observed accessing the Google Play Store during some period in June this year.

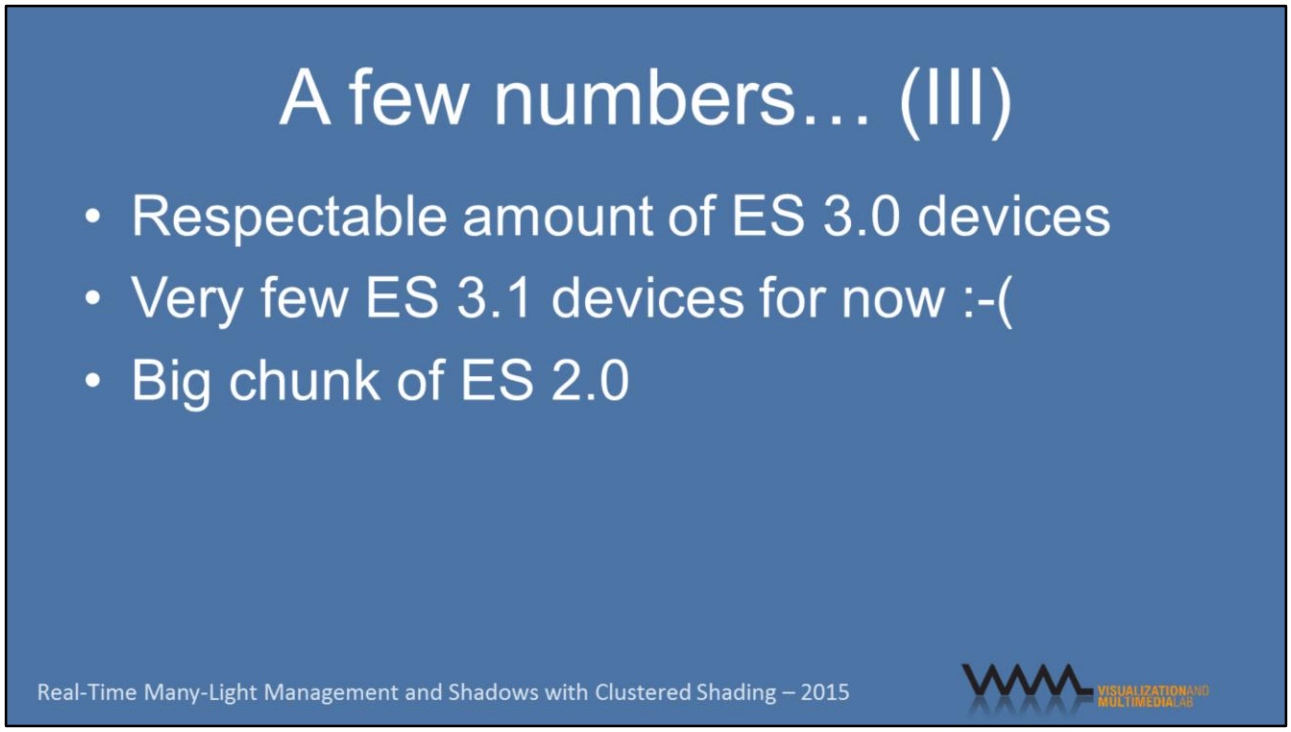
A few numbers… (II)

- Unity Mobile Stats (various platforms):

OpenGL ES Version(*)	Percentage
2.0	58%
3.0	40%
3.1	~2%

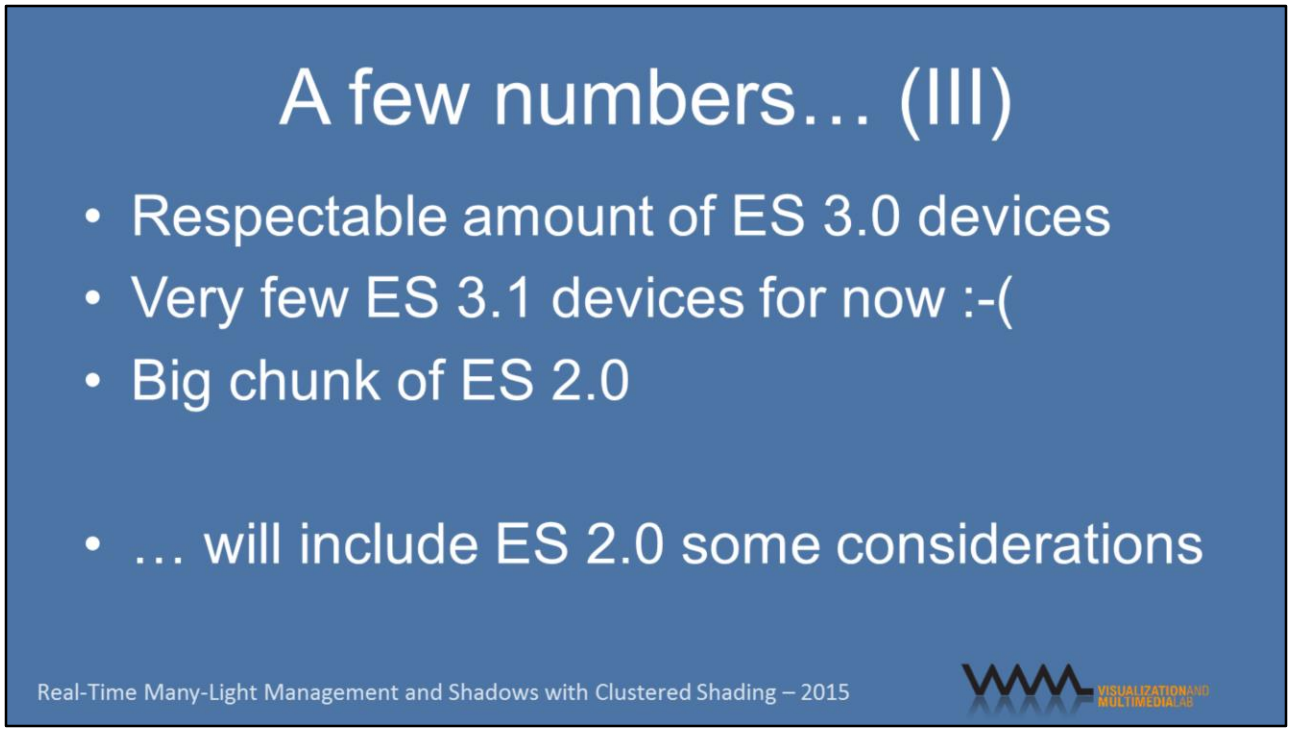
Data from April 2015; (*) = Shader Generations
Source: http://hwstats.uinty3d.com/mobile/gpu.html

This second table the same information, but this time as measured by the Unity Mobile Hardware survey. It shows rather similar figures, albeit this time including non-Android devices.



The good news is that there's a respectable chunk of ES 3.0 devices, around 40%, which is nice. ES 3.1 is still a bit scarce, though – something around 2%. And while ES 3.2 was announced a few days ago, unsurprisingly, it's adoption today is still sort of low…

Finally there's the big chunk of ES 2.0 devices, which covers the remaining close-to-60%. ▶▶▶



<contd.>

Because of this, I'm going to include some considerations that relate to ES 2.0 when discussing the different many light methods.

OpenGL ES

- ES 2.0 : shader-based and FBO support
  - But no MRT in core

One thing to point out straightaway is the lack of support for multiple render targets. So while OpenGL|ES 2.0 supports shaders and custom frame buffer objects, the core spec only provides for a single color render target. ▶▶▶

OpenGL ES

- ES 2.0 : shader-based and FBO support
  - But no MRT in core
- ES 3.0 “fixes” that

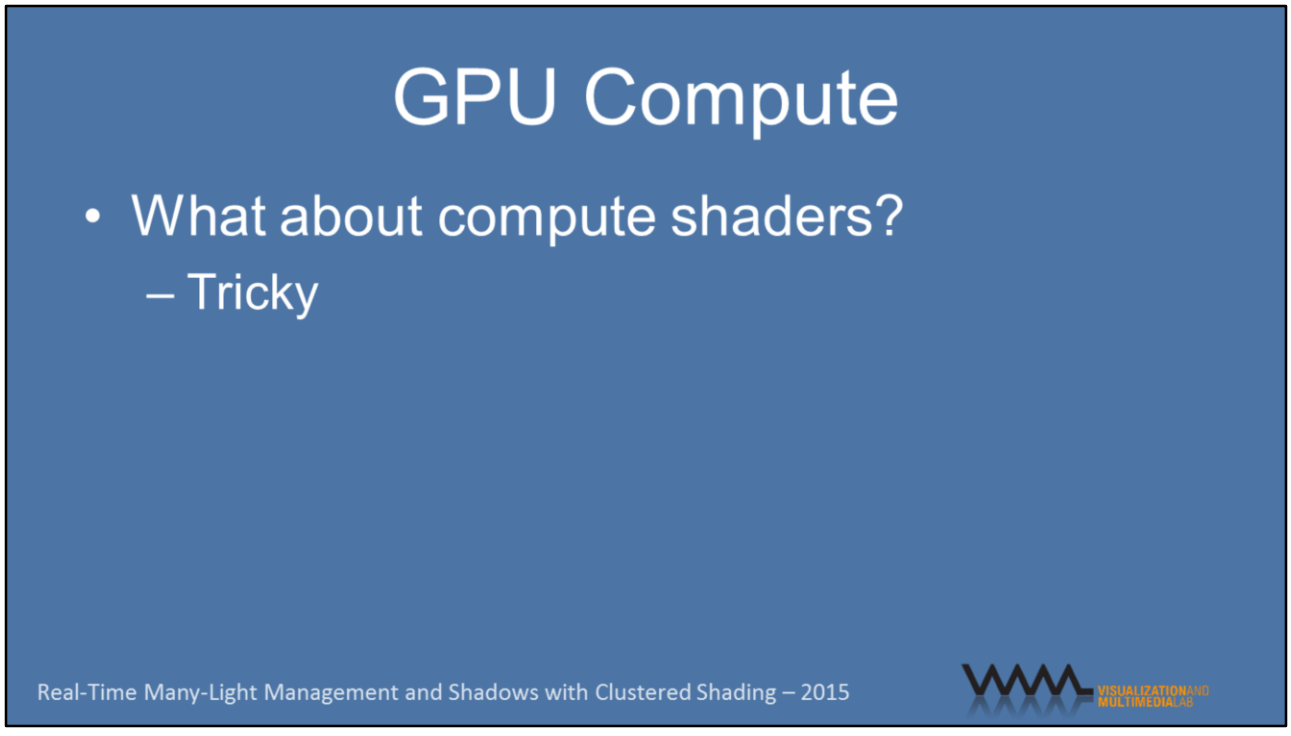
<contd.>

ES3.0 fixes that, among other things. The ES 3.0 spec provides for at least four color attachments.

GPU Compute

- What about compute shaders?

Some of the techniques and their variation you've heard about today rely somewhat heavily on GPU compute shaders. ▶▶▶



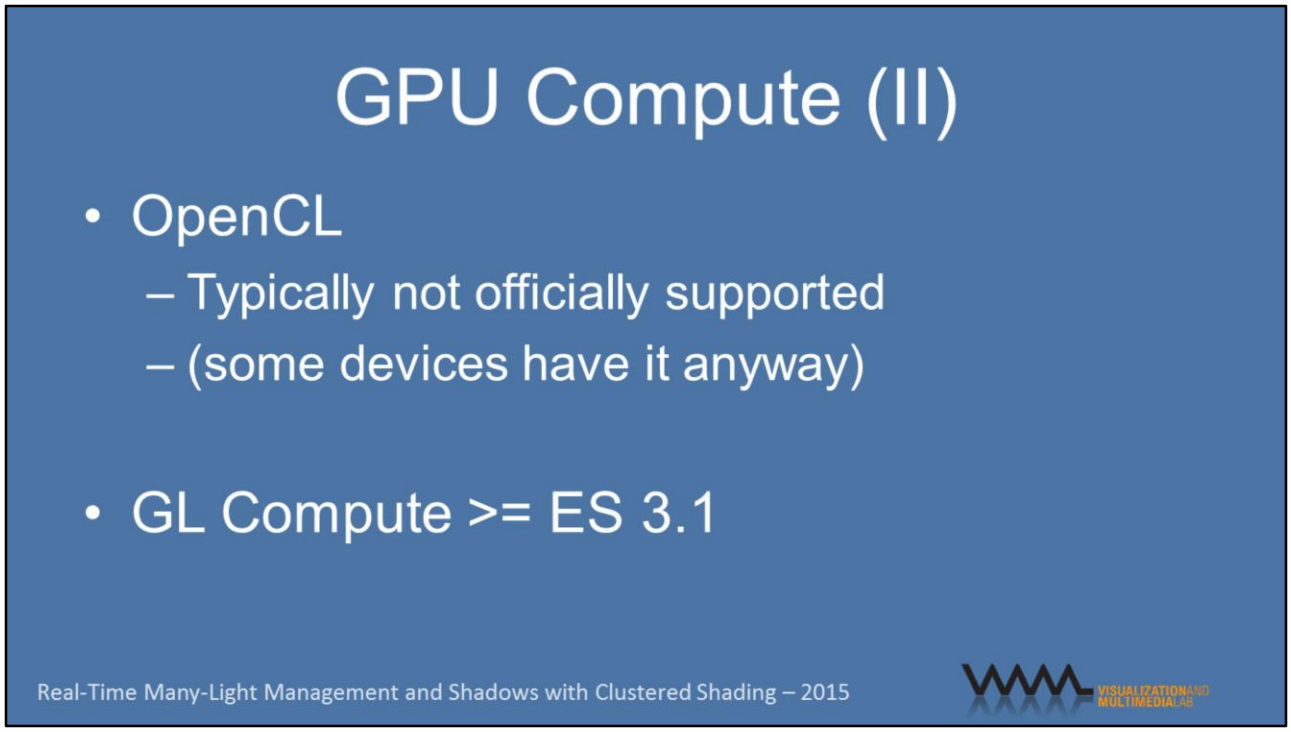
<contd.>

Unfortunately, for now, the situation doesn't look too good for that on mobile devices.

GPU Compute (II)

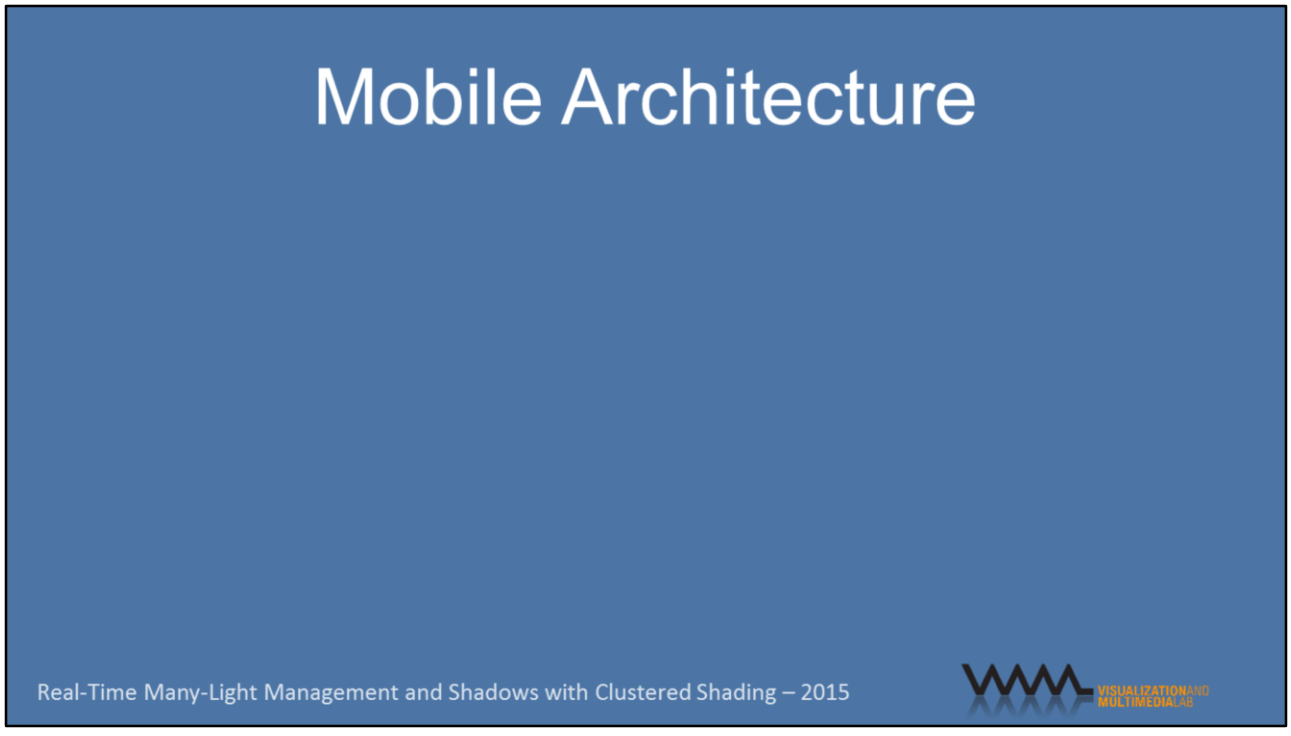
- OpenCL
  - Typically not officially supported
  - (some devices have it anyway)

Neither Android nor iOS support OpenCL officially. Some manufacturers of Android devices include support anyway, though. ▶▶▶

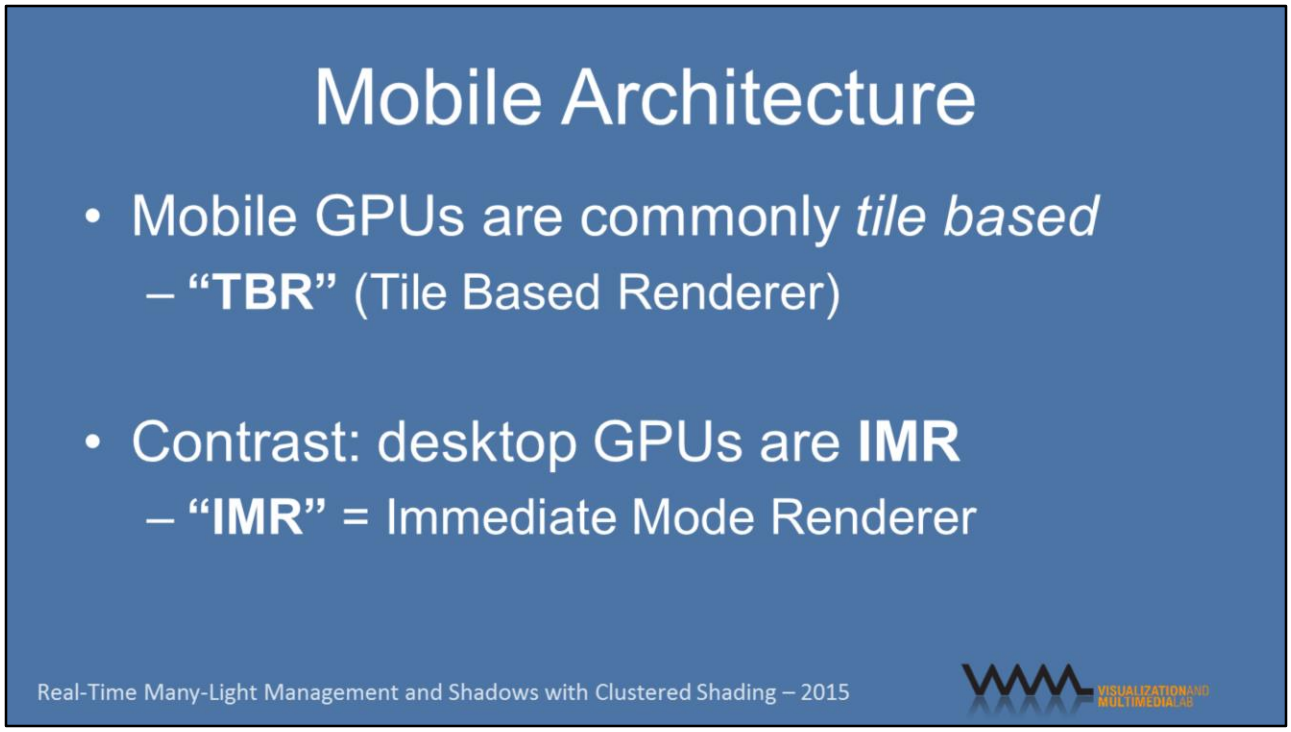


<contd.>

The other option is to use the compute shaders included in OpenGL ES 3.1, at least if or when you can rely on ES 3.1+ being available on your target devices.

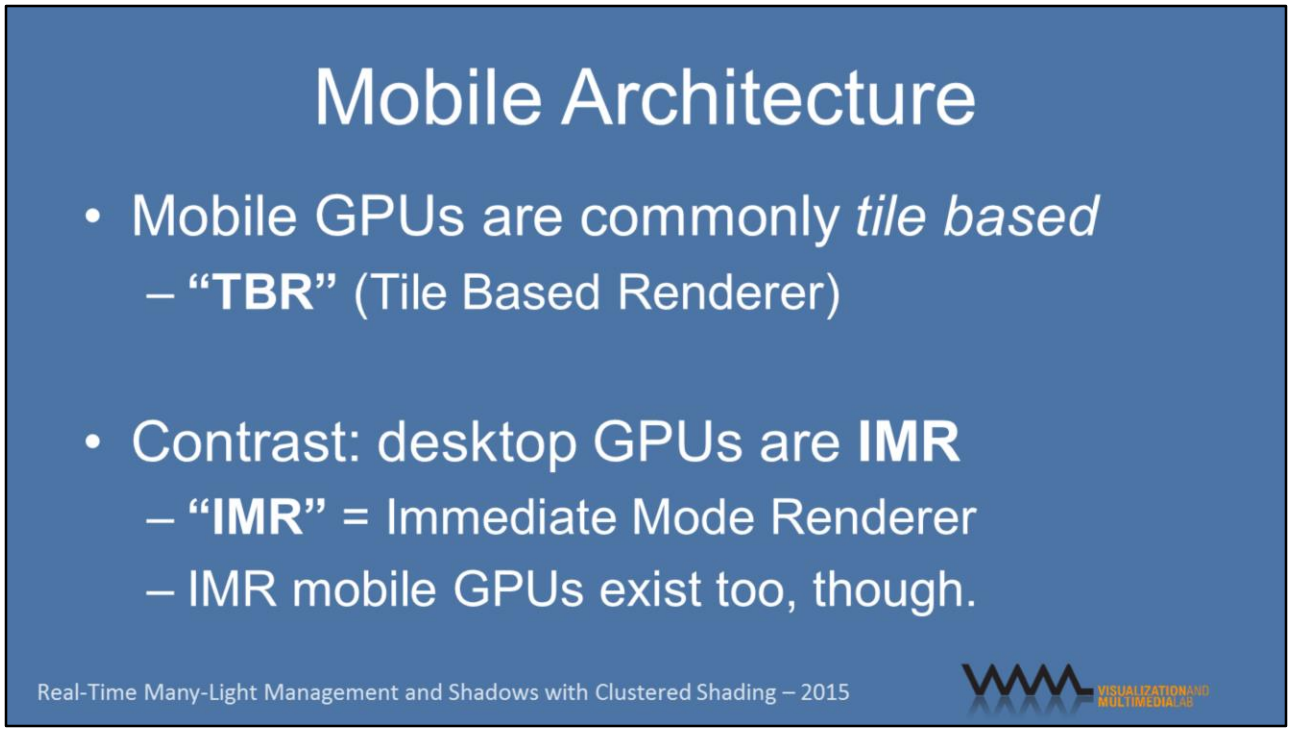


So, that was mostly about the limitations and difficulties compared to high-end GPUs.
▶▶▶



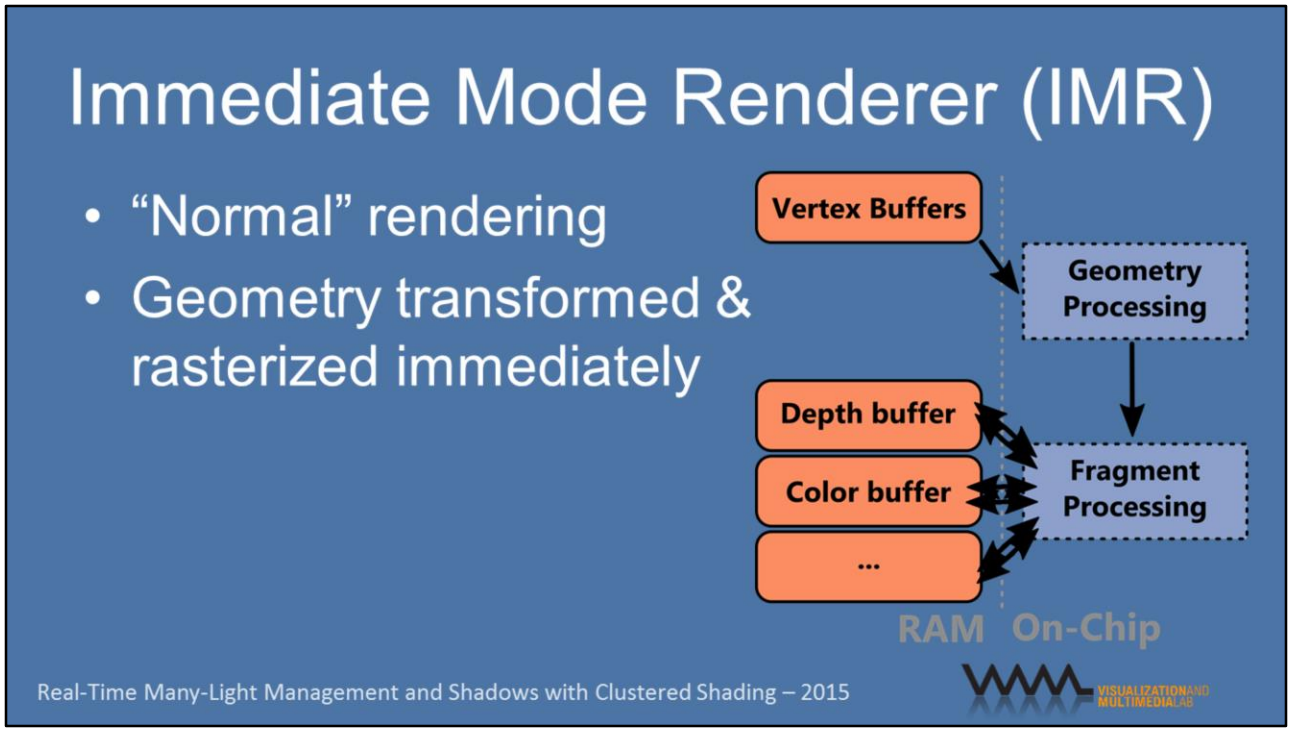
<contd.>

Now, a large percentage of mobile GPUs are tile based renderers, or TBR for short; this in contrast to the desktop GPUs that are typically Immediate Mode Renderers, or IMR for short. The main difference between these two architectures has some interesting implications, so let's look at how these two differ. ▶▶▶

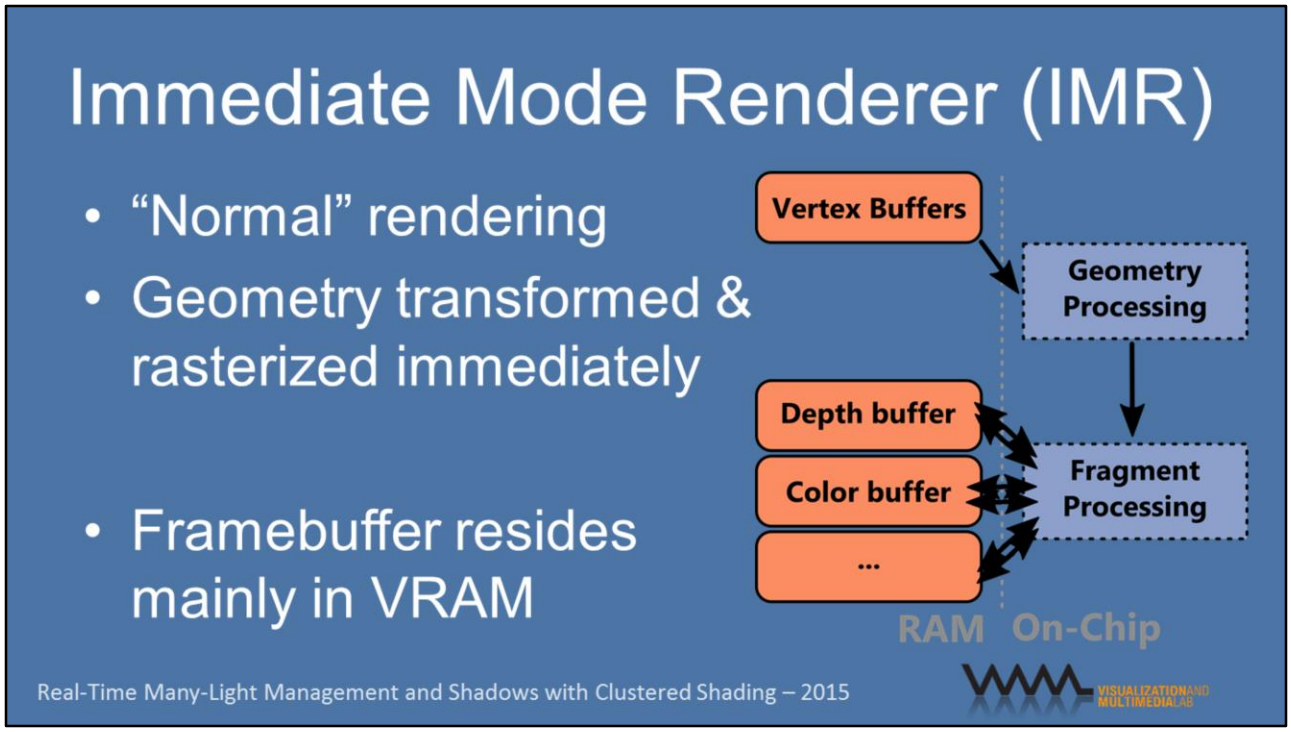


<contd.>

Small note: there are IMR based mobile GPUs too.

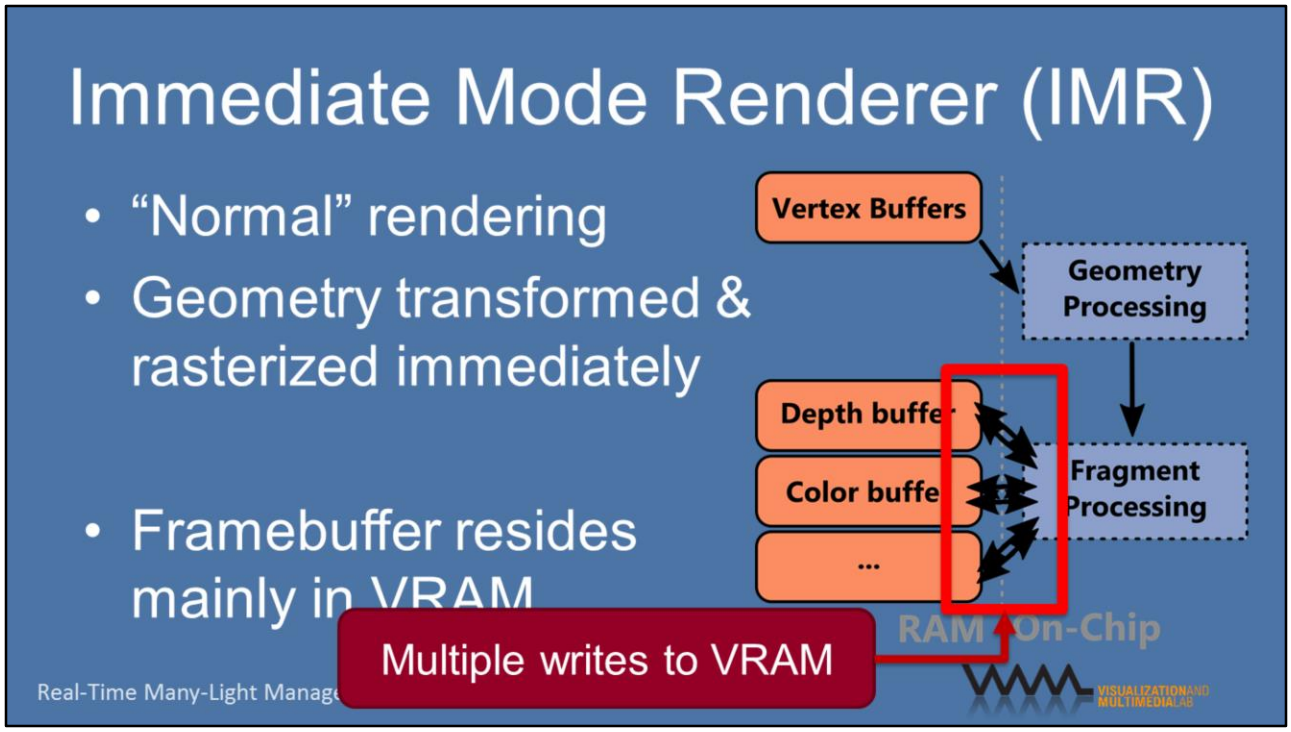


An immediate mode renderer is what I'd consider a "traditional" HW pipeline. Here, the geometry is stored in (V)RAM and submitted to the GPU in batches. The GPU transforms the geometry in the geometry processing stage, that is in the vertex shaders and so on, and then **immediately** sends the results to be rasterized and shaded via some on-chip mechanism. ▶▶▶



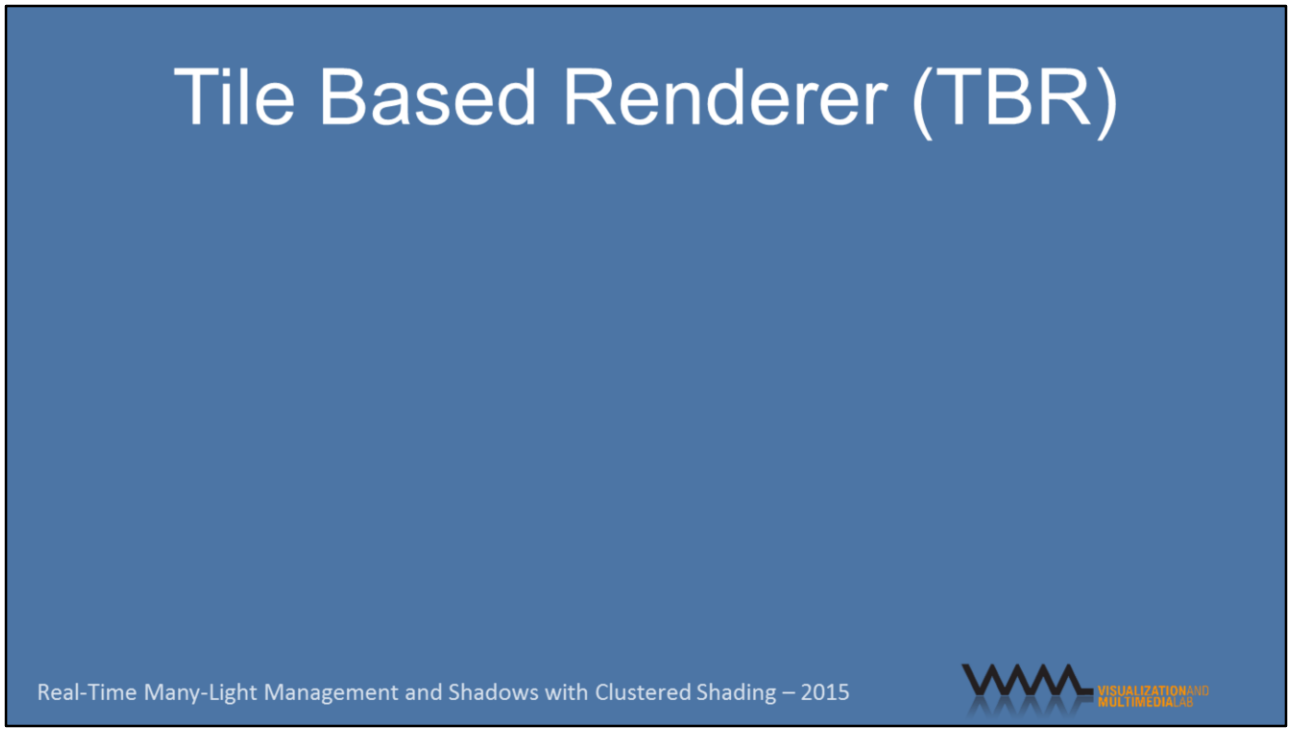
<contd.>

The results from the shading are merged into the framebuffer that here typically resides in VRAM in its entirety. ▶▶▶

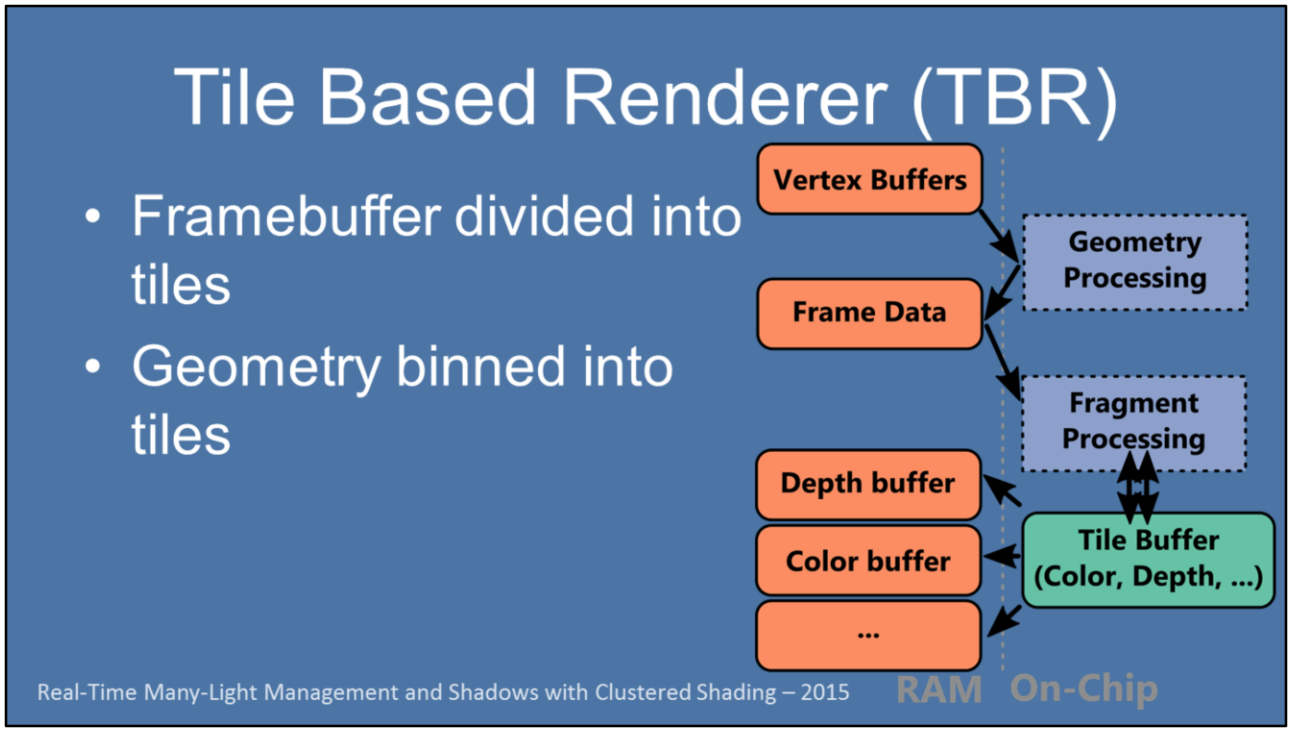


<contd.>

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

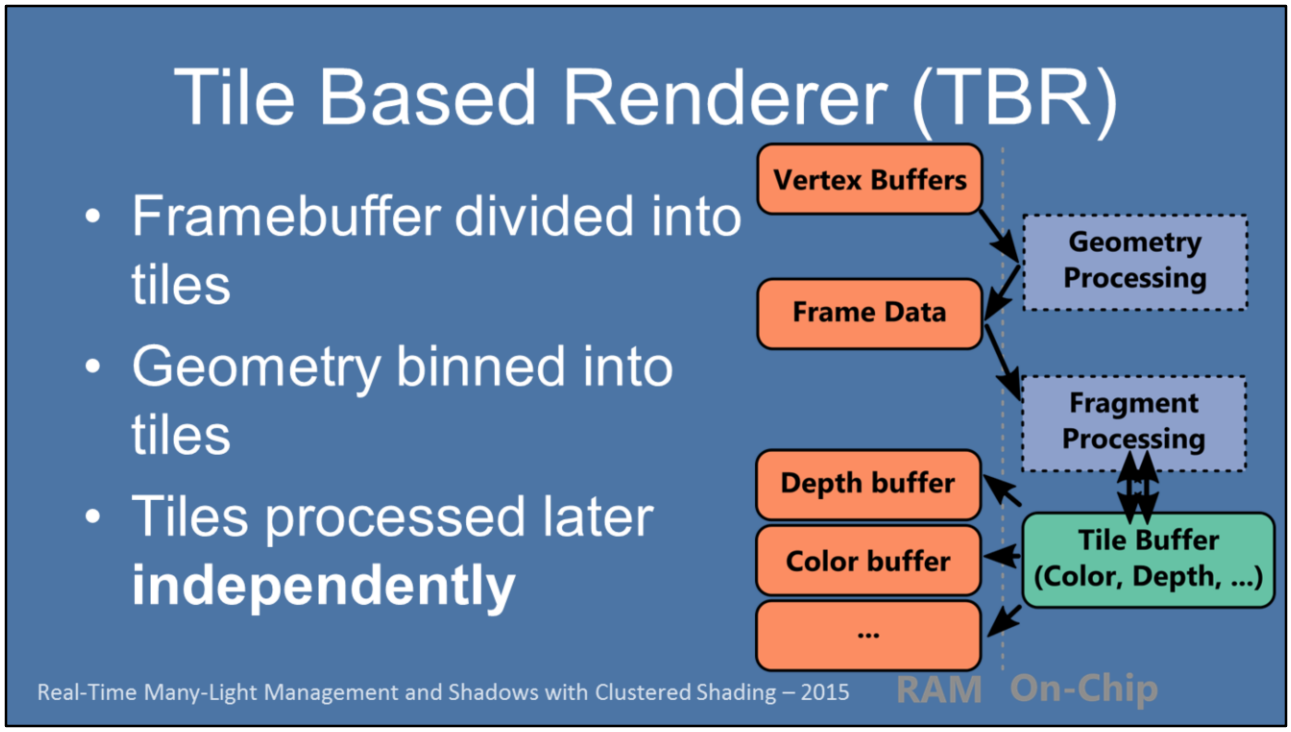


Now, let me compare this to a tile based renderer instead. ▶▶▶



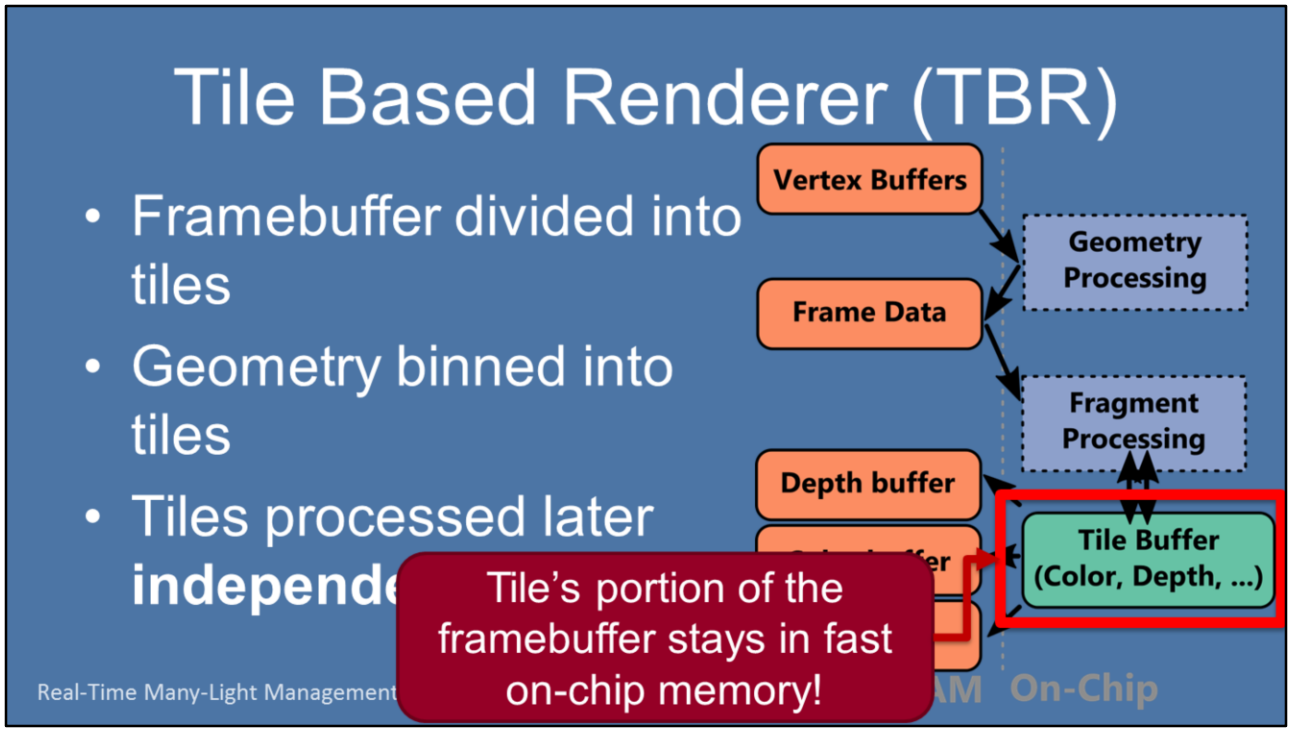
<contd.>

Tile based rendering gets its name from the fact that the framebuffer is subdivided into many tiles. When the application submits geometry, it's transformed as normally. But instead of being rasterized immediately, the transformed geometry is binned into the tiles and stored for future processing. ▶▶▶



<contd.>

At some later point, for example when all geometry has been submitted, each tile is processed. Tiles can now be processed independently. The geometry associated with each tile is rasterized and shaded. The trick here is that each tile's portion of the framebuffer can be kept in local on-chip memory for the whole duration of rasterization and shading. ▶▶▶



<contd.>

With this we avoid multiple expensive writes (and reads) to and from RAM whenever there is overdraw and instead hit the on-chip storage. Additionally, when rendering to a tile has finished, the tile's framebuffer can be compressed when it's transferred –or "resolved"– to RAM, which further reduces memory BW.

Tile Based Renderer (TBR) (II)

- Tile's FB-contents stored to RAM when needed

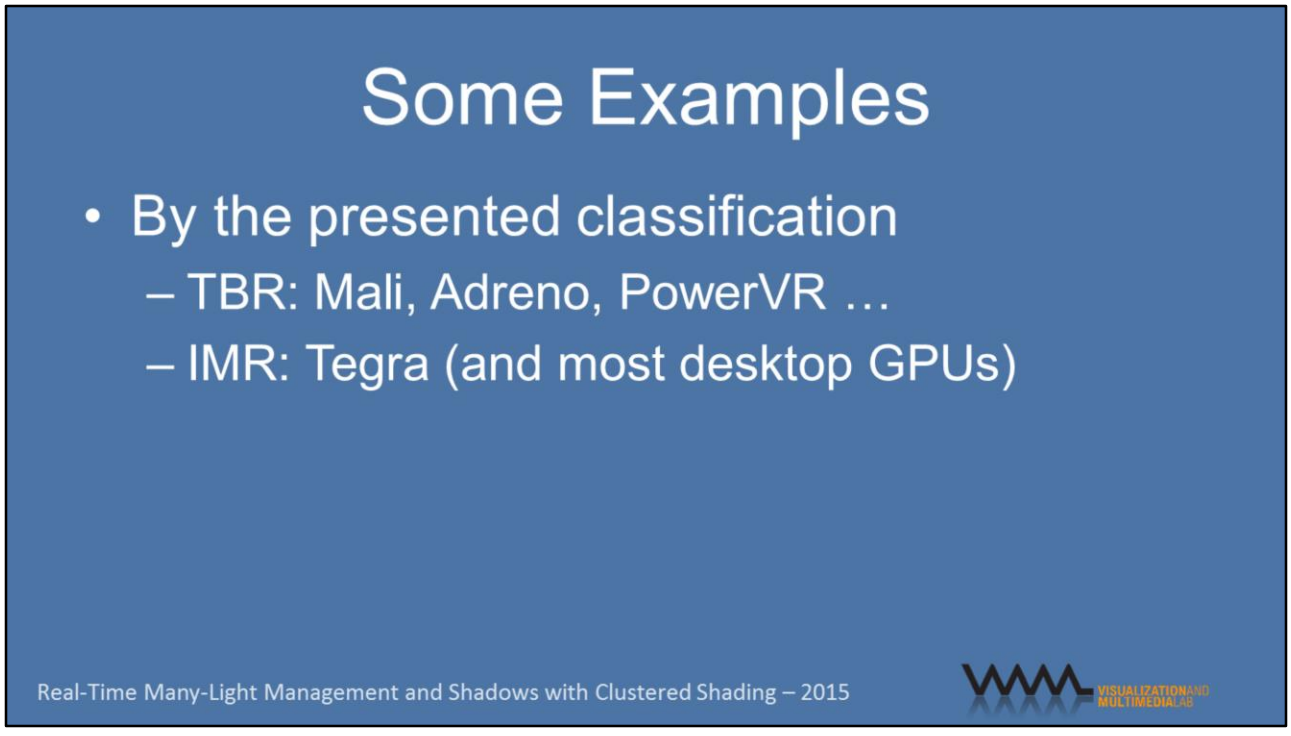
The tile's framebuffer contents are stored to RAM only when needed. ▶▶▶

Tile Based Renderer (TBR) (II)

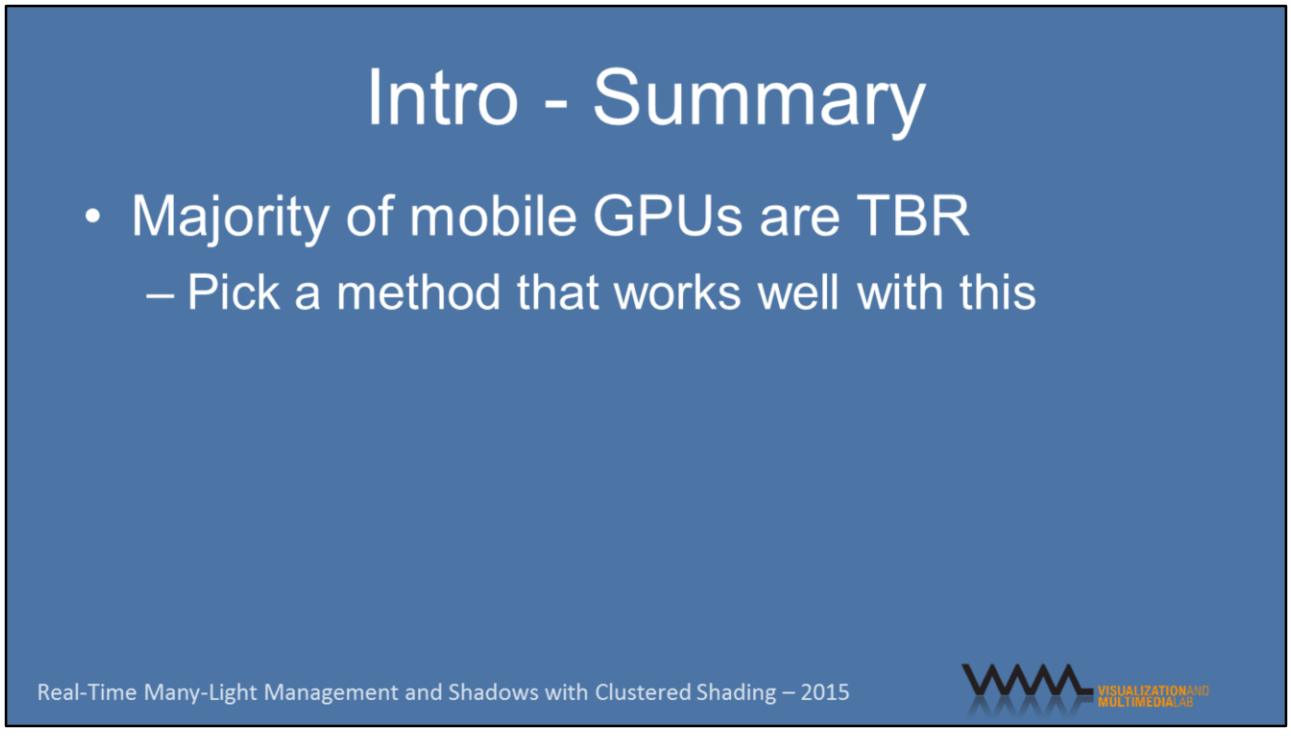
- Tile's FB-contents stored to RAM when needed
- Best case: **once**
  - When all rendering to that tile has finished

<contd.>

In the best case, this only happens once per frame, when all the rendering for that frame and tile has finished.



According to the presented classification, quite a few of the common mobile GPUs are all tile-based renderers, as you can see listed on the slide. The only mobile chips that I know that are IMR are the various NVIDIA Tegra chips, such as the K1 and the X1. Of course, most desktop GPUs are IMR as well.



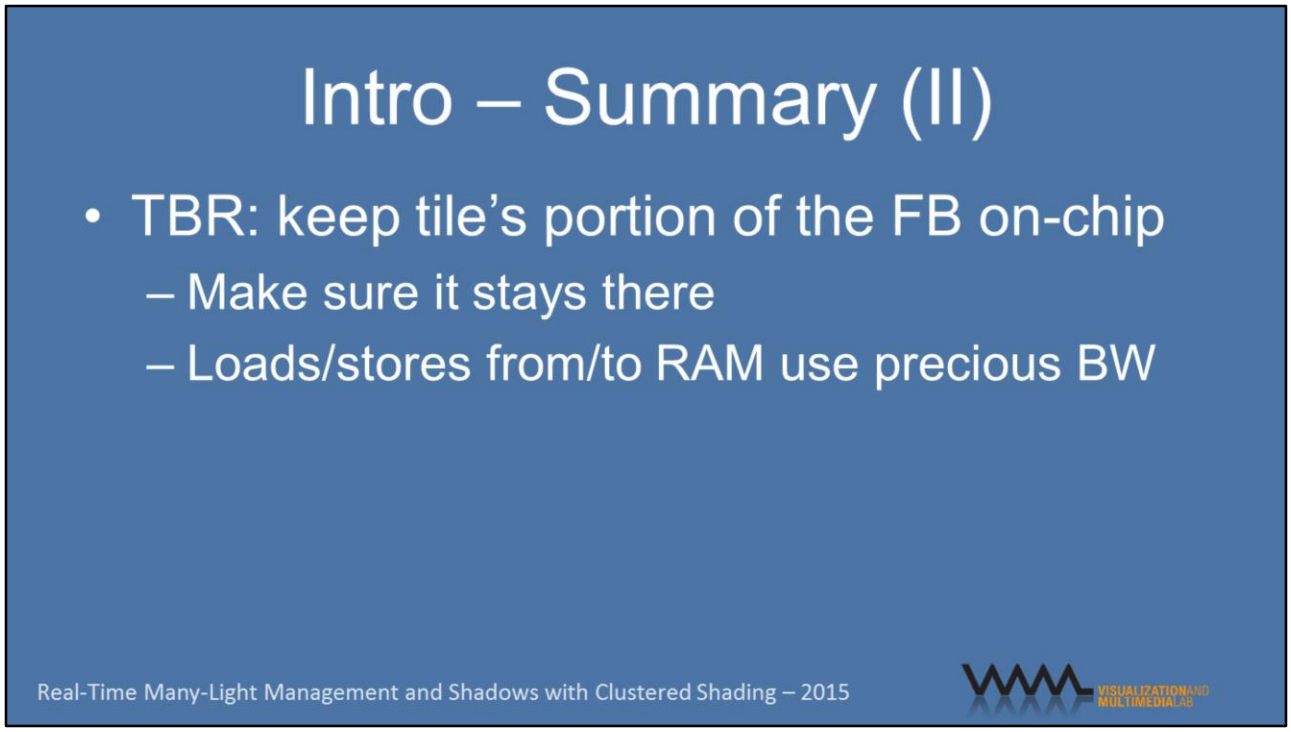
This concludes the first part of my talk. Before I move on to the many light methods, let me just quickly summarize the most important aspects of this introduction.

So, the majority of mobile GPUs are tile-based renderers. We definitively want to pick a method that maps well to this hardware architecture.

Intro – Summary (II)

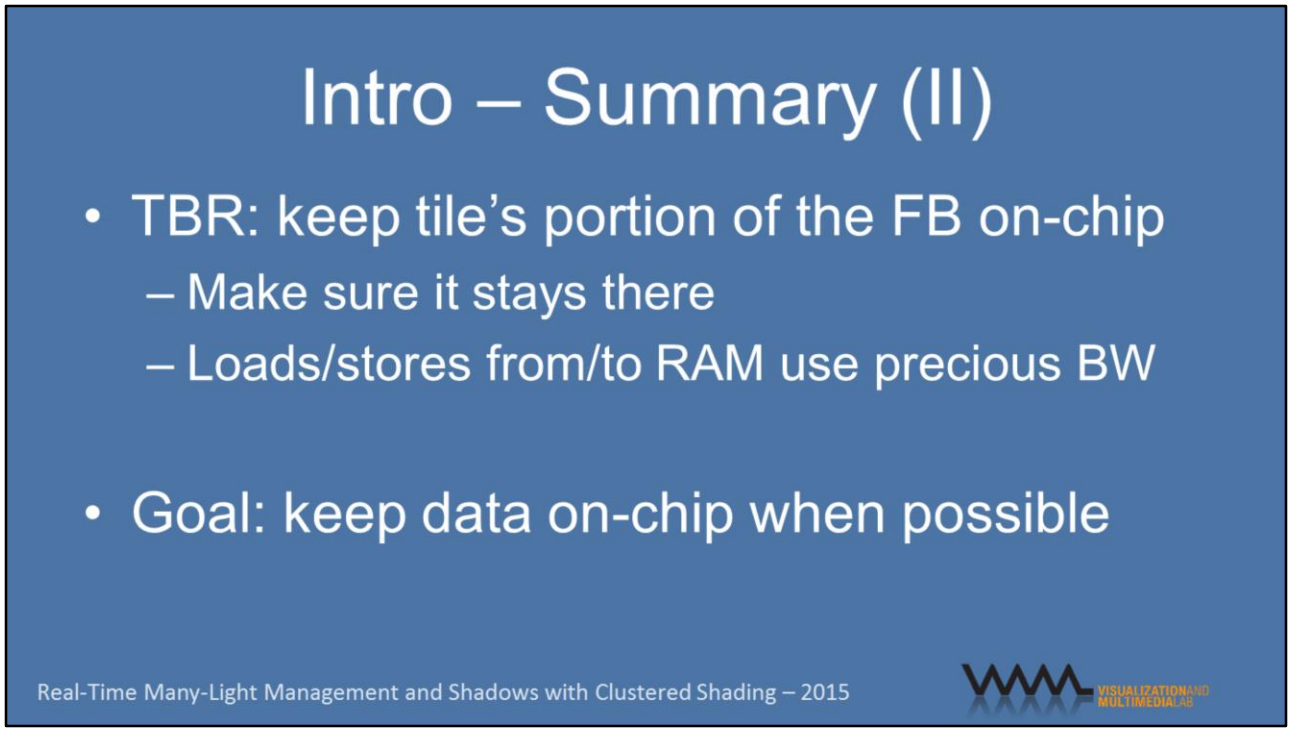
- TBR: keep tile's portion of the FB on-chip

The key feature of a tile-based renderer is that it keeps each tile's portion of the framebuffer in fast on-chip memory during shading. ▶▶▶



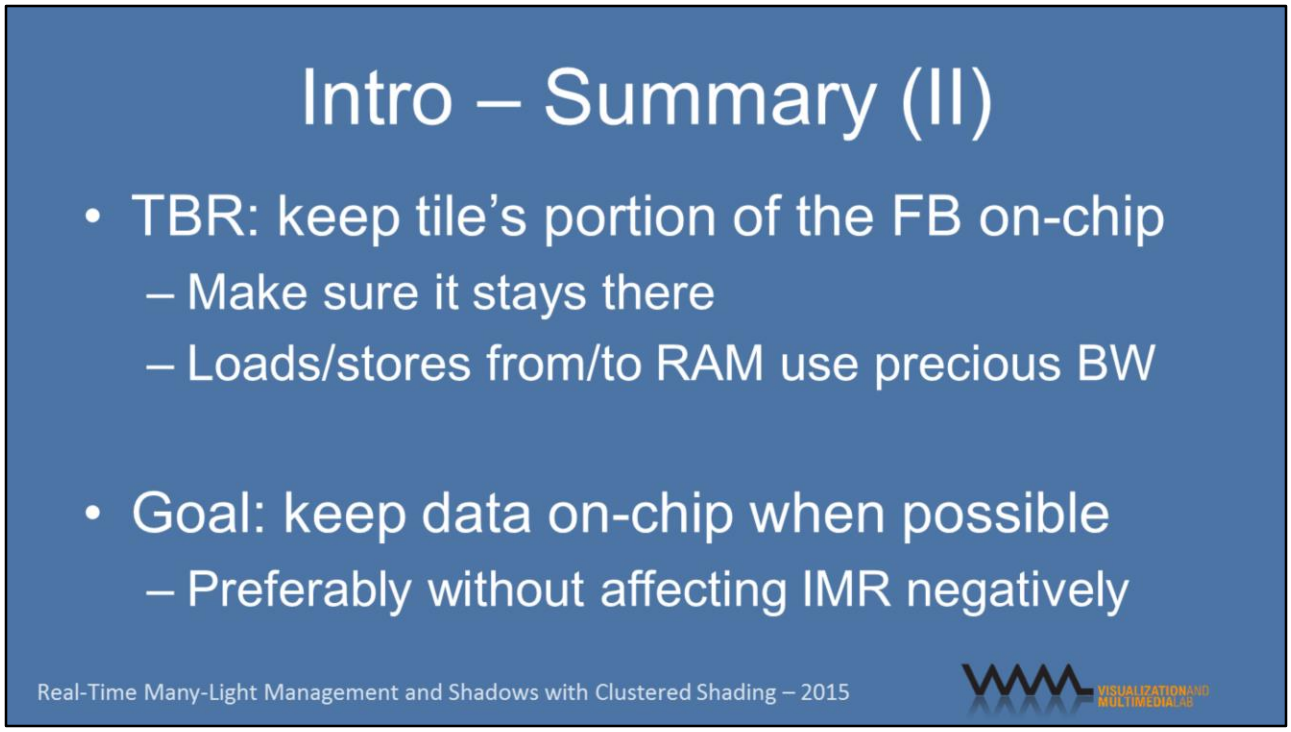
<contd.>

Our goal is to make sure that it can stay there, since storing it to RAM and later loading it back from RAM uses precious memory bandwidth, which is costly in terms of both performance and power consumption. ▶▶▶



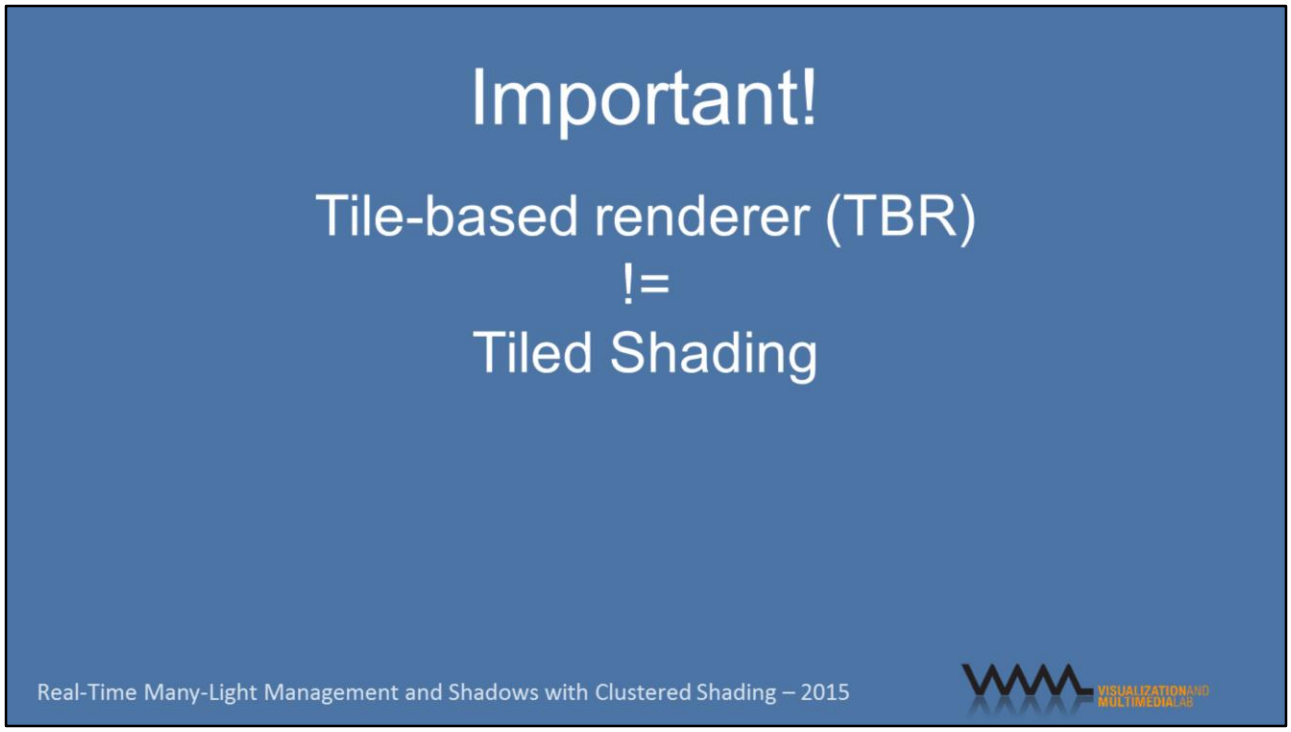
<contd.>

So, yeah, our goal is to find a method that allows us to keep as much data on-chip as long as possible.



<contd.>

And … we'd like to do this without negatively affecting performance on a normal immediate-mode rendering architecture too much.

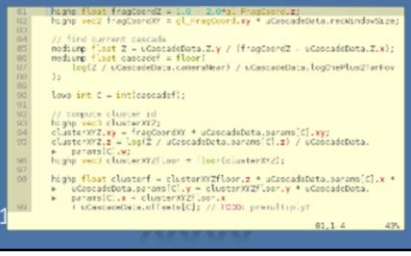


Important note: the tile-based renderer that I've been talking about so far is **not** the same as the tiled shading method that you've heard about earlier in the course.

Important!

- TBR: hardware property
  - It's out of your hands
- Tiled Shading: software algorithm
  - Your choice

Src: Wikipedia/Exynos, by User:Köf3, CC-SA3



Real-Time Many-Light Management and Shadows with Clustered Shading – 201

The tile-based renderer is a hardware property, and in that sense, it's largely out of your hands – at least if you want to support a wide variety of different devices.

Tiled shading on the other hand is a software algorithm, and it's up to you to implement it (or not).

Important!

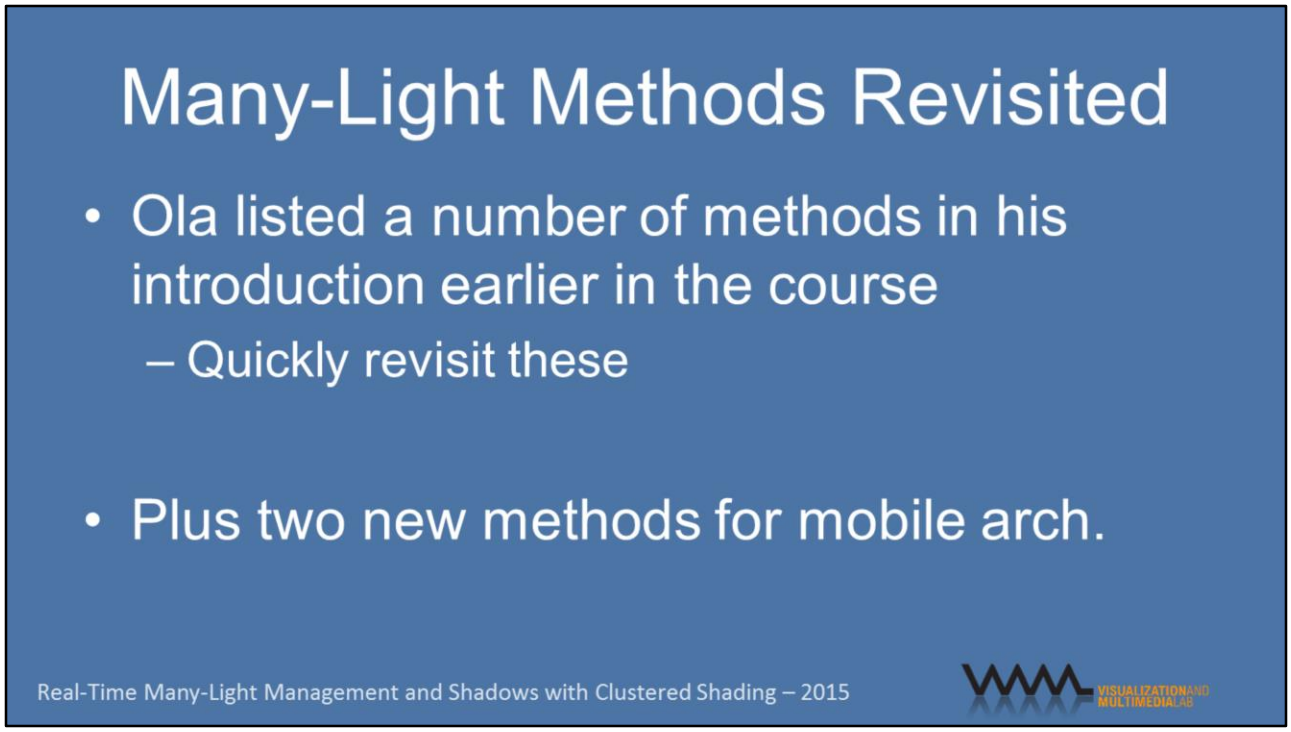
- It's perfectly valid to use *Tiled Shading* on a *TBR* platform
  - As we'll see shortly.

It's perfectly valid use tiled shading on a tile-based renderer. More about this in the upcoming, second part.

Part 2

MANY-LIGHT METHODS REVISITED

We now know a bit about the mobile hardware that we're targeting, and with this, what properties we'd like to see in our algorithms. So now we can revisit the many-light rendering methods and reason a bit about their suitability for mobile hardware.



In the first part of the course, Ola listed a number of many-light methods. I'm going to quickly revisit some of these. I'm also including two new methods that really focus on mobile architectures and, in this case, take special advantage of the on-chip storage.

The Methods

- Plain Forward
- Traditional Deferred
- Clustered Deferred
- Clustered Forward
- Practical Forward
- Deferred Tile Store
- Forward Light Stack

Here's a list of the methods.

The first, plain forward, serves as a sort-of base line method to compare against. Next, there's two deferred methods. I'll then transition to look at the different clustered methods, including the practical clustered that Emil presented earlier. Finally, there's the two new methods that both were presented by Martin et al. at SIGGRAPH 2013. As mentioned, these specifically target TBR-like architectures.

A small note. For the sake of brevity, I'm not going to make any distinction between tiled and clustered methods in this review. ▶▶▶

Tiled & Clustered

- I'll lump tiled and clustered shading together for now
- Basic idea is similar enough
  - Pick the one that suits your use case better
  - E.g., tiled for 2D-ish settings, clustered for 3D..

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

<contd.>

Tiled and clustered shading are very similar in spirit, in fact, you could consider tiled shading to be a special case of clustering, where the dimensionality of the clustering has been reduced from 3D to 2D.

When picking between these methods, you should anyway pick the one that matches your use-case better. So, for a setting with very little depth-complexity and discontinuities, such as for example a top-down view, the 2D tiling may be sufficient and will be easier to implement.

For a full 3D first- or third-person view, clustering, on the other hand, may be a better choice, since it's more robust with respect to varying views and results in a more accurate light assignment.

Even if you opt for the clustering, it might worth to see if it's possible to adapt the method to your use case. Depending on your needs, you might get away with a simpler clustering with fewer depth-layers. Or you might want to perform the clustering in a different space. In the final part of this presentation, I will quickly present one such adaption.

The Methods

- Plain Forward
- Traditional Deferred
- Clustered Deferred
- Clustered Forward
- Practical Forward
- Deferred Tile Store
- Forward Light Stack

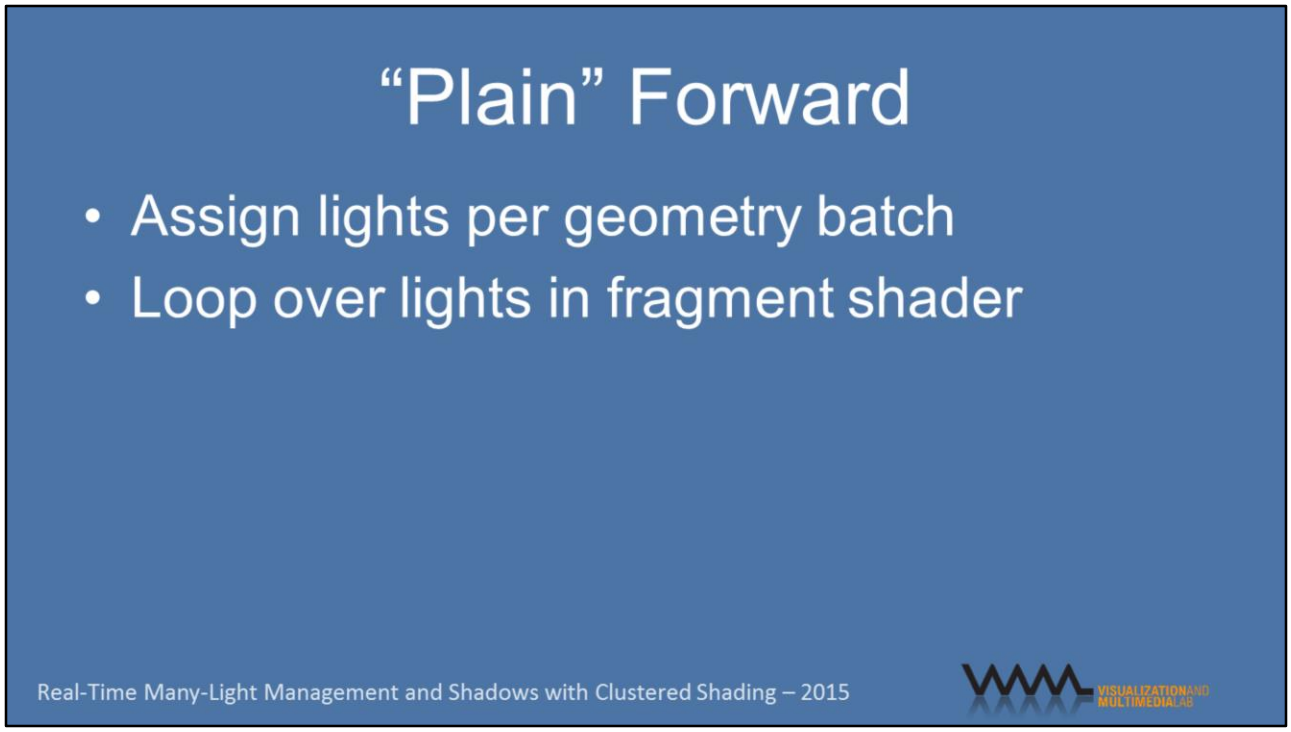
Anyway, I'll get started with the review of the different methods. ▶▶▶

The Methods

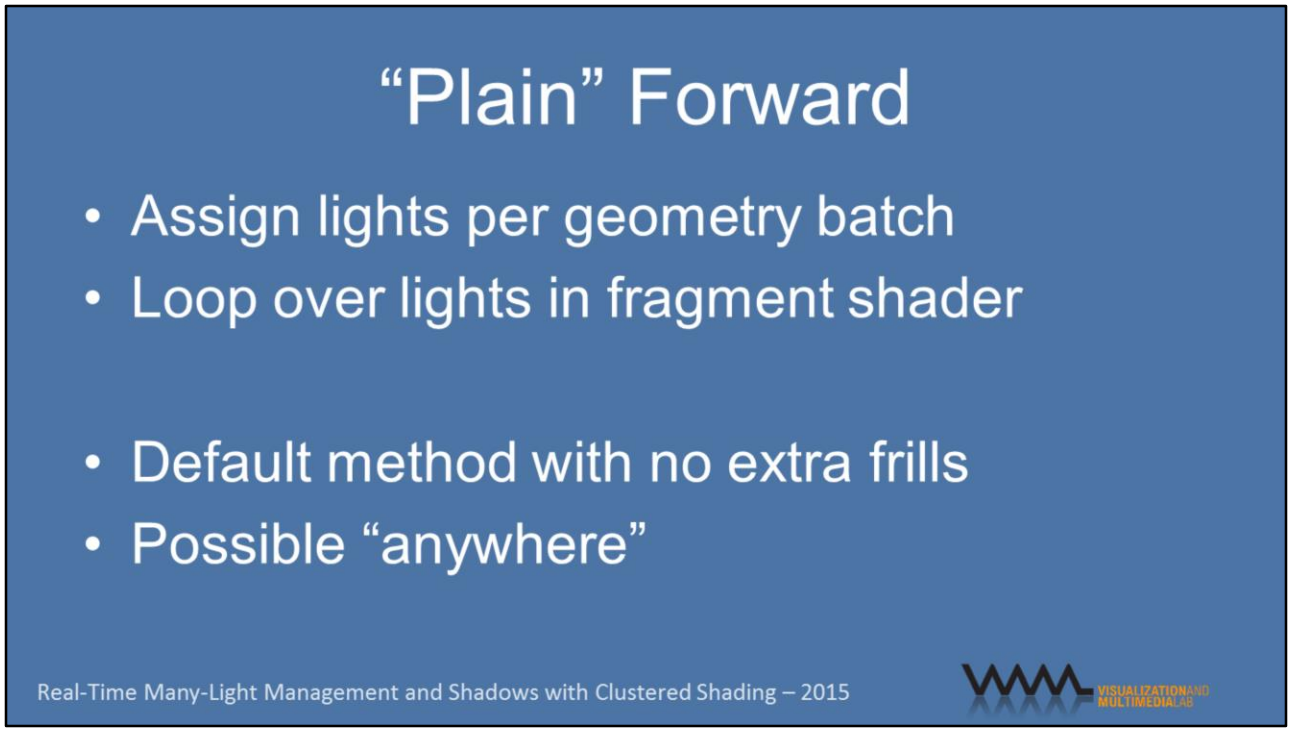
- **Plain Forward**
- **Traditional Deferred**
- **Clustered Deferred**
- **Clustered Forward**
- **Practical Forward**
- **Deferred Tile Store**
- **Forward Light Stack**

<contd.>

The first of which is the plain forward rendering method.



Here, lights are assigned to each geometry batch that's drawn. During shading you then simply loop over all the lights in your fragment shader and accumulate the results. ▶▶▶

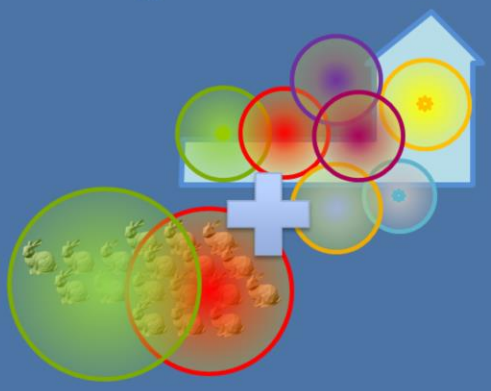


<contd.>

This is pretty much the text-book way of doing rendering in OpenGL (and elsewhere), so it should be possible pretty much anywhere, regardless of e.g. OpenGL ES version.

“Plain Forward” (II)

- Scales badly with number of lights
  - See Ola’s introduction



As explained by Ola in the introduction, it scales badly with large numbers of lights; or at least it’s difficult to robustly support scenes with many lights in the general case.

The Methods

- **Plain Forward**
- **Traditional Deferred**
- **Clustered Deferred**
- **Clustered Forward**
- **Practical Forward**
- **Deferred Tile Store**
- **Forward Light Stack**

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Traditional Deferred							
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

So, for each method I'm going to add an entry to this table that summarizes some of the properties. I'll start with the plain forward. As mentioned, I'd not really consider it a many light method. But it can get its work done with a single geometry pass, and only needs to store the colors off-chip – the depth buffer can be discarded, unless some later pass or screen-space techniques requires that data.

Plain forward natively supports HW-MSAA and blending, something that we will see that most forward methods have in common.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Traditional Deferred							
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

Let's move on to the next method, Traditional Deferred rendering.

Traditional Deferred

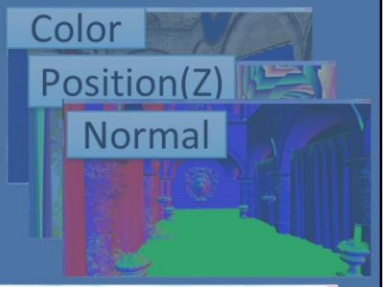
- Render scene to G-Buffers

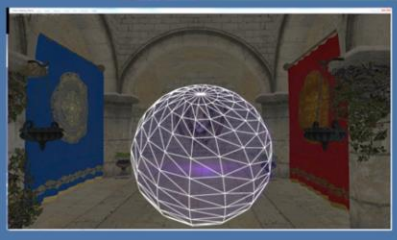
Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

Here, we start off by rendering the scene to generate the G-Buffers that store the information we later need to compute the shading. ▶▶▶

Traditional Deferred

- Render scene to G-Buffers
- Render lights using proxy geometry





Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

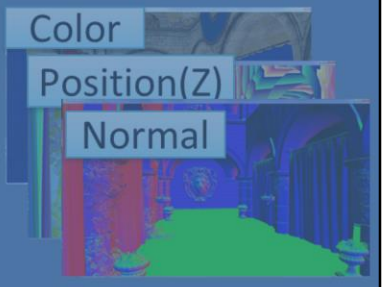
VISUALIZATION AND MULTIMEDIA LAB

<contd.>

After rendering the G-Buffers, we render lights using proxy geometry. ▶▶▶

Traditional Deferred

- Render scene to G-Buffers
- Render lights using proxy geometry
- In shader:
  Read G-Buffers, compute lighting, blend results.



Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

VISUALIZATION AND MULTIMEDIA LAB

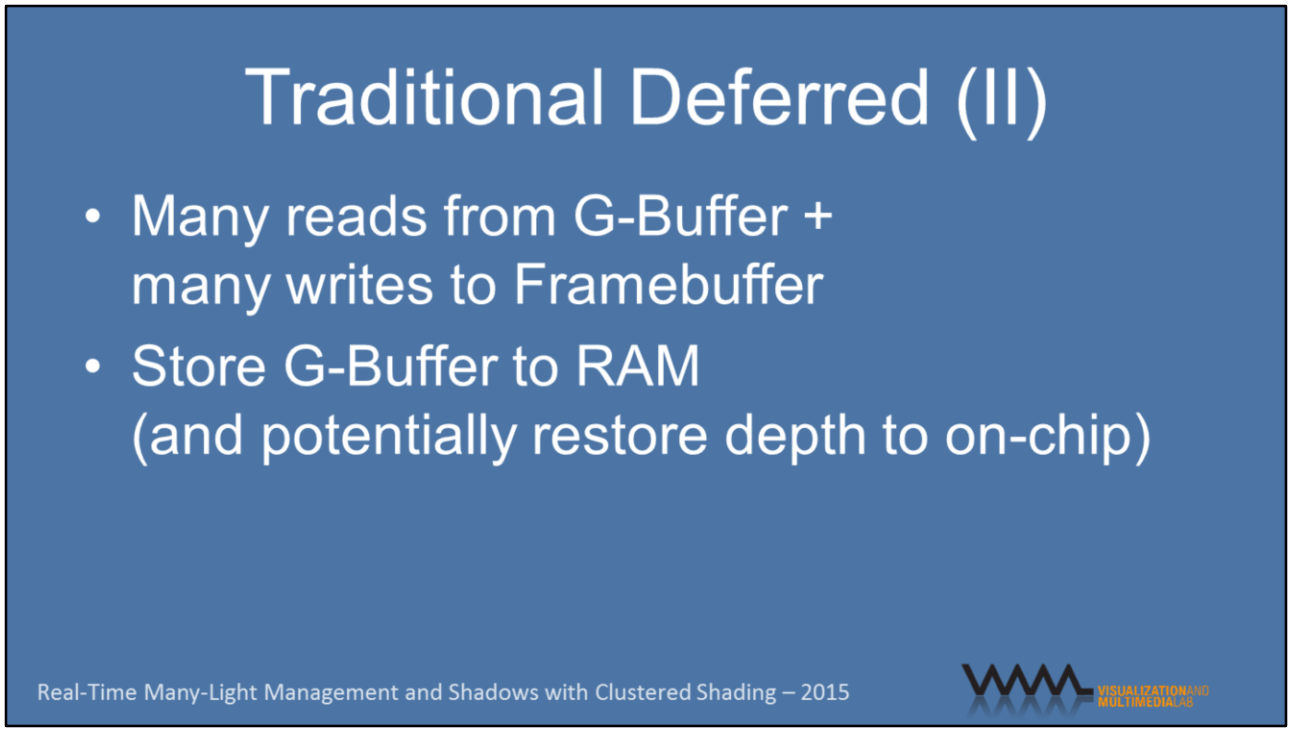
<contd.>

For each fragment generated from the proxy geometry, we sample the G-Buffers and then compute the contribution from the current light source. That contribution is accumulated into the resulting framebuffer via blending.

Traditional Deferred (II)

- Many reads from G-Buffer + many writes to Framebuffer

So, for each light that ends up affecting a certain sample, we need to read from the G-Buffer once and write to the framebuffer once. ▶▶▶



<contd.>

In addition to that, after the G-Buffers have been rendered, we need to transfer that data off-chip, to RAM, so that it can be sampled using textures.

If we want to use the depth buffer during the second pass, to better cull the proxy geometry, it has to be additionally restored from RAM to the on-chip storage before the lighting pass shading can run.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Traditional Deferred							
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

So, for the entry for traditional deferred looks as follows:

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

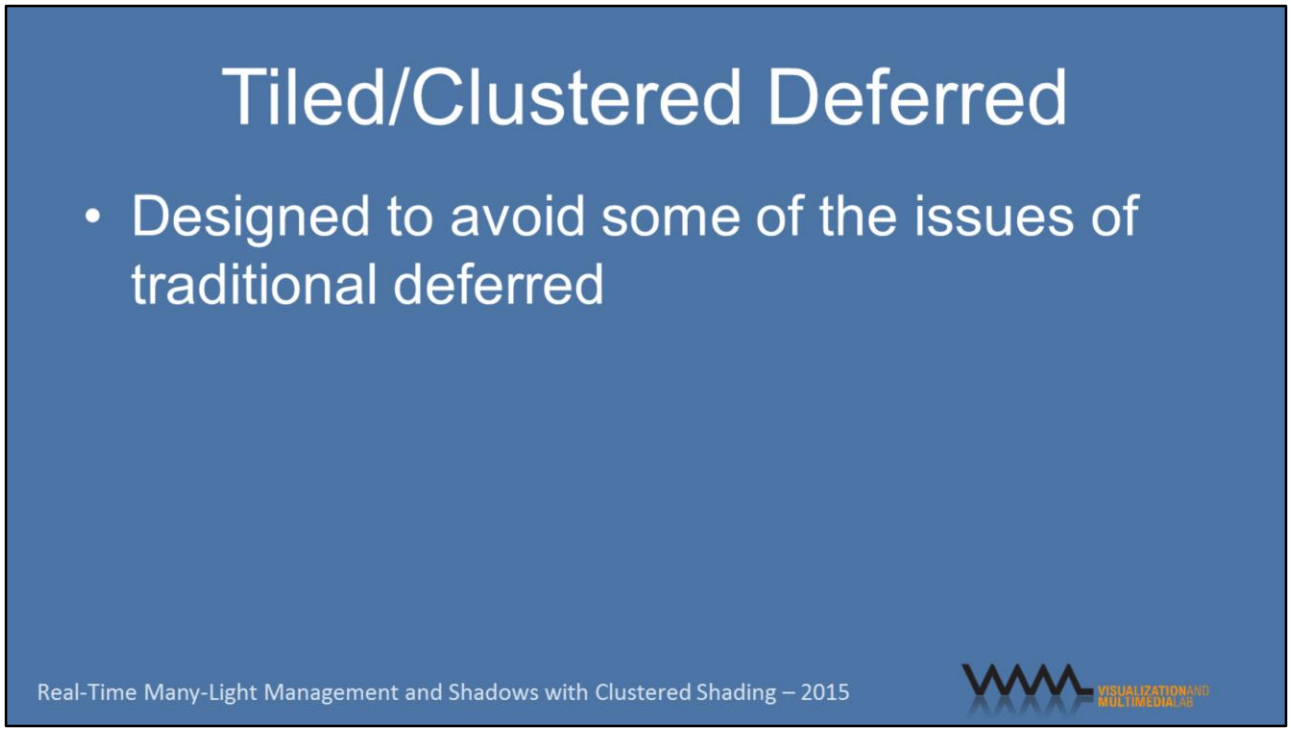
<contd.>

We need the G-Buffers; there's still only one geometry pass, and we avoid the overshading issues. MSAA and blending on the other hand become more tricky, as is usual with deferred techniques. Generating G-Buffers in a single geometry pass requires support for multiple render targets, which puts this technique into OpenGL|ES 3.0 territory.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

Next up is the clustered and tiled deferred technique.



Basically, the tiled deferred method was developed to avoid some of the issues of the traditional deferred method, namely the repeated reads from the G-Buffers and the repeated writes to the resulting framebuffer.

Tiled/Clustered Deferred (II)

- Render scene to G-Buffers
- Compute light assignment
  - Use e.g. depth from G-Buffers

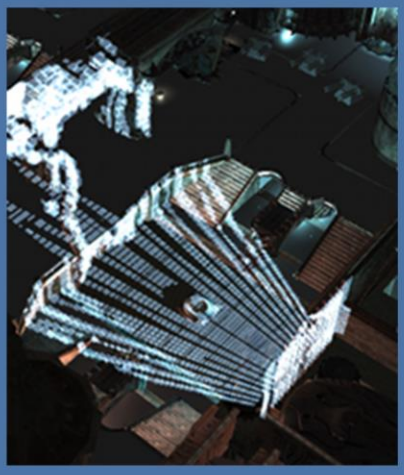
VISUALIZATION AND MULTIMEDIA LAB

It works roughly as follows. The scene has to be again rendered to the G-Buffers.

Further, we need to perform a light assignment. There's a couple of different options here, so for a basic tiled deferred variant the light assignment can be done independently of the rendered geometry, by just projecting lights to the screen and assigning them to the 2D tiles they overlap. ▶▶▶

Tiled/Clustered Deferred (II)

- Render scene to G-Buffers
- Compute light assignment
  - Use e.g. depth from G-Buffers



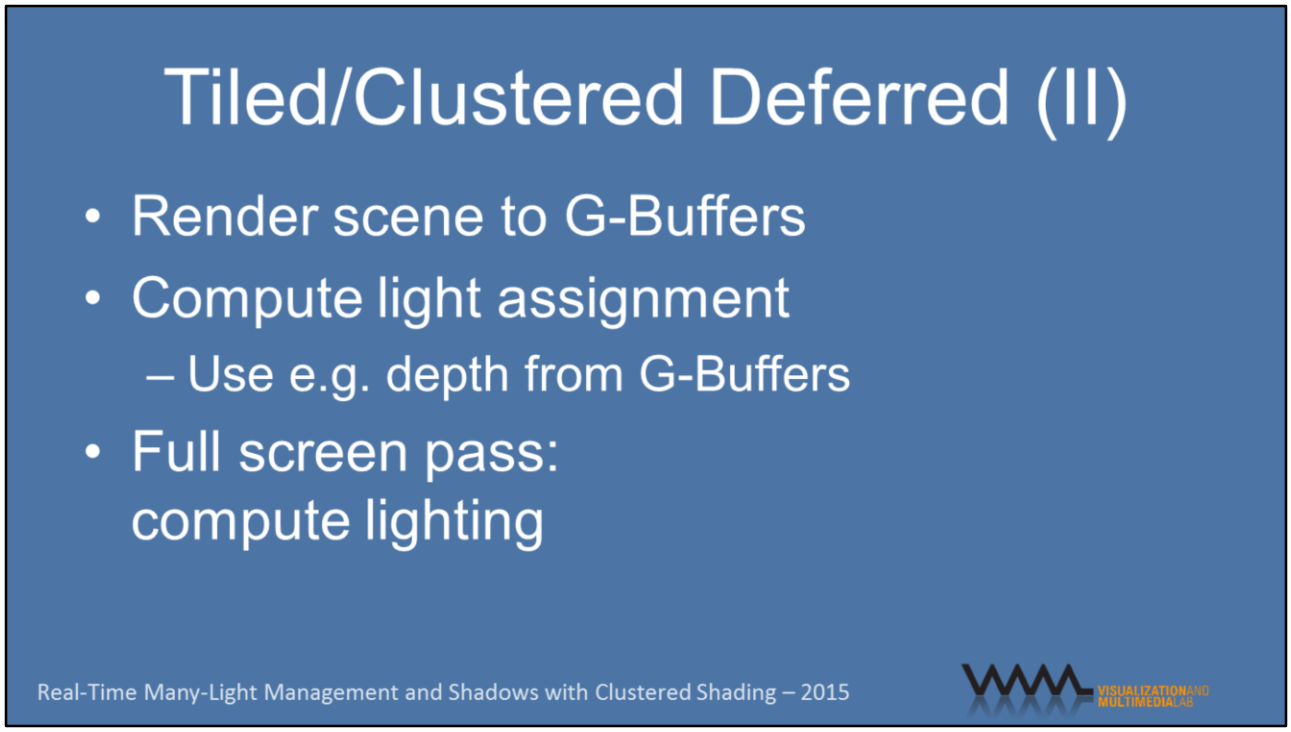
Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

VISUALIZATION AND MULTIMEDIA LAB

<contd.>

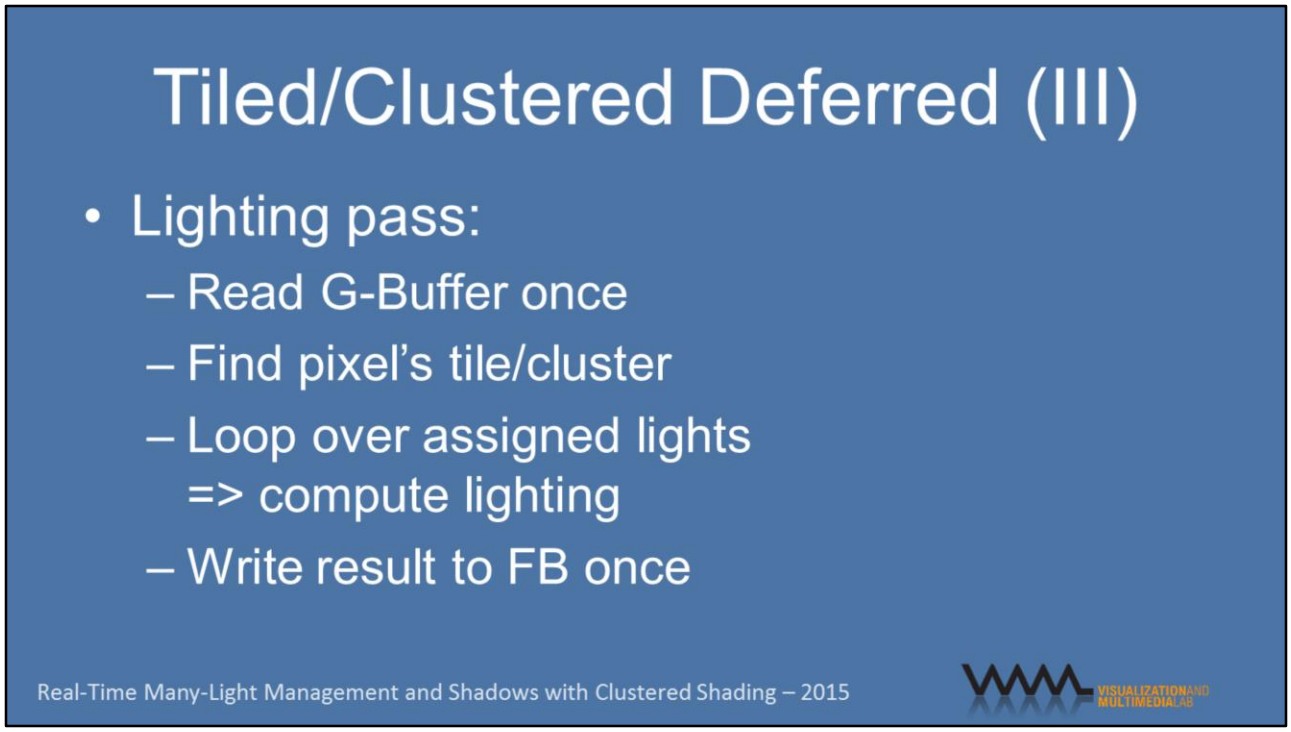
More complex methods use information from the G-Buffers to compute a better light assignment, so for instance, our original sparse clustering extracted clusters from the depth-buffer and assigned lights only to these active clusters.

Either way, the result of the light assignment are per-tile or per-cluster lists of lights that potentially affect the corresponding tile or cluster. ▶▶▶

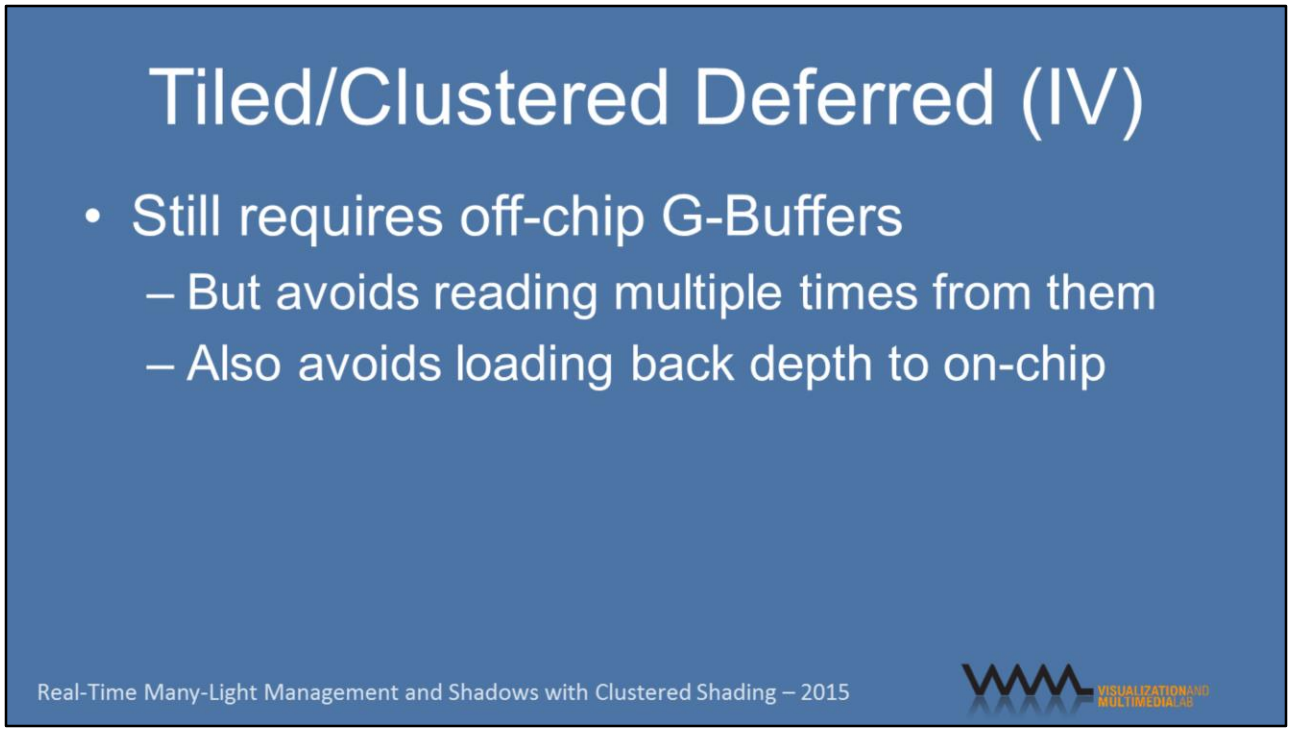


<contd.>

Finally, there's a single full screen pass, where the lighting is computed.



This full-screen pass looks as follows. For each sample, the sample's data is read from the G-Buffer once. We then find out which tile or cluster that sample belongs to, and from this, which lights potentially affect the sample. At that point we can simply loop over the lights in the shader, compute the contribution of each light source and accumulate the results locally in the shader. Finally, we store the shaded results once to the framebuffer.



Now, this technique still requires the G-Buffers, but instead of having to read from the G-Buffers once for each light, they are sampled only once in total. Similarly, we only need to write the final result once to the resulting framebuffer.

We also avoid having to restore the depth-buffer from RAM to the on-chip store, since our full-screen pass doesn't benefit from depth testing in any way. ▶▶▶

Tiled/Clustered Deferred (V)

- Original clustered method relies on compute shaders to
  - Identify active clusters
  - Perform light assignment

The original clustered method relies heavily on compute shaders to first extract the list of active clusters and secondly to compute the light assignment.

Tiled/Clustered Deferred (V)

- Original clustered method relies on compute shaders to
  - Identify active clusters
  - Perform light assignment
- Avoid with Emil's Practical Clustered

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

<contd.>

The compute shaders can be avoided by using Emil's Practical Clustered variation, though. I'll return to the Practical Clustered in a few slides, but then with focus on the forward variant.

Tiled/Clustered Deferred (VI)

- For tiled shading
  - Perform light assignment on CPU
  - Compute per-tile bounds in fragment shader
  - No compute shaders, but potential read-back

For tiled shading, the situation is a bit better. In the simplest form, the light assignment can be done independently from the rendered geometry. Alternatively, the light assignment can be improved by finding the min-max depth-bounds of each tile and using that information to cull the light sources more aggressively. The min-max depth-bounds can be reduced using e.g., a fragment shader and then read back to system memory.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Deferred							
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

The tiled/clustered deferred method looks relatively similar to the traditional deferred; this table doesn't show the reduction in G-Buffer reads and framebuffer writes, though.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

Next, tiled and clustered forward.

Tiled/Clustered Forward

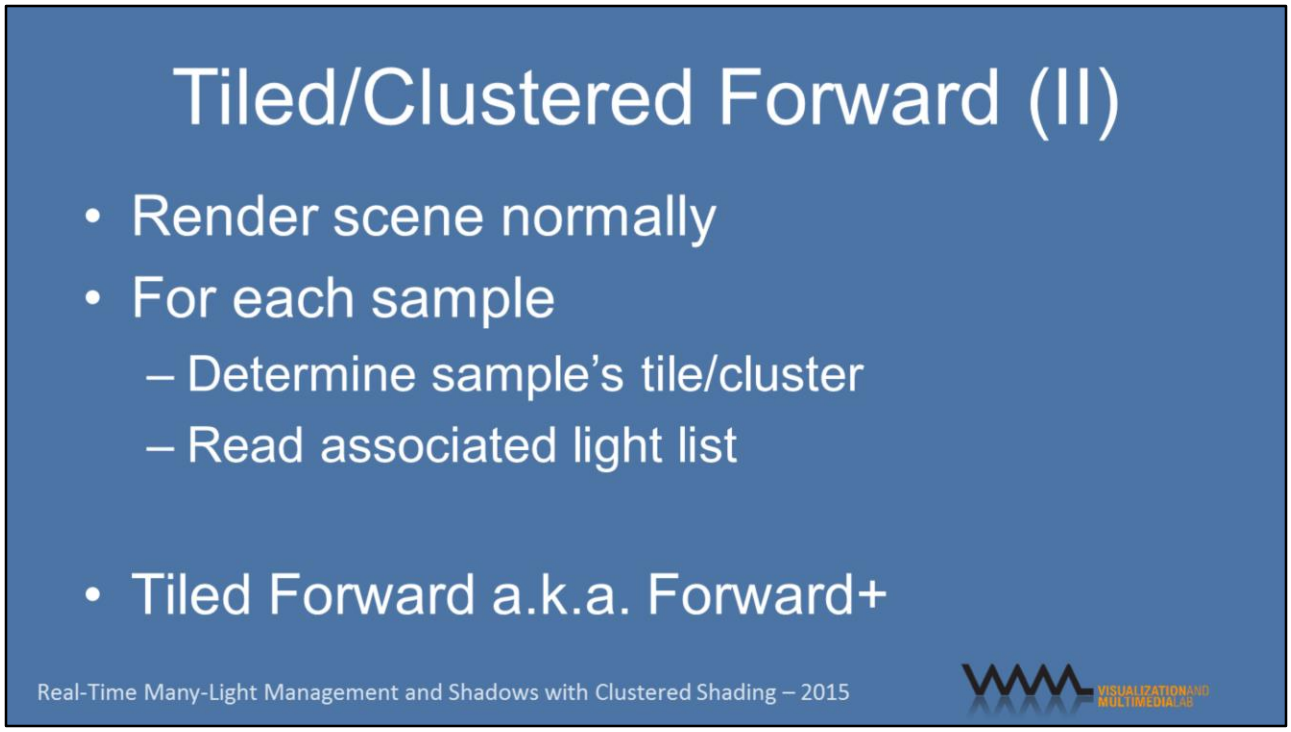
- Use depth-pass to determine per-tile Z-bounds or clusters
  - For light assignment

Instead of rendering the full G-Buffers, the tiled/clustered forward method performs a depth-only pre-pass. This again allows us to identify active clusters in the case of clustering, or per-tile depth bounds in the case of tiling. Then, similarly to the deferred methods, we compute per-cluster or per-tile light lists.

Tiled/Clustered Forward (II)

- Render scene normally
- For each sample
  - Determine sample's tile/cluster
  - Read associated light list

In a second pass, we render the scene "normally" in a forward fashion. For each generated fragment, we find what cluster or tile it belongs to so that we can access the list of lights that potentially affect the fragment. We loop over those lights, and accumulate the contributions in the shader. ▶▶▶



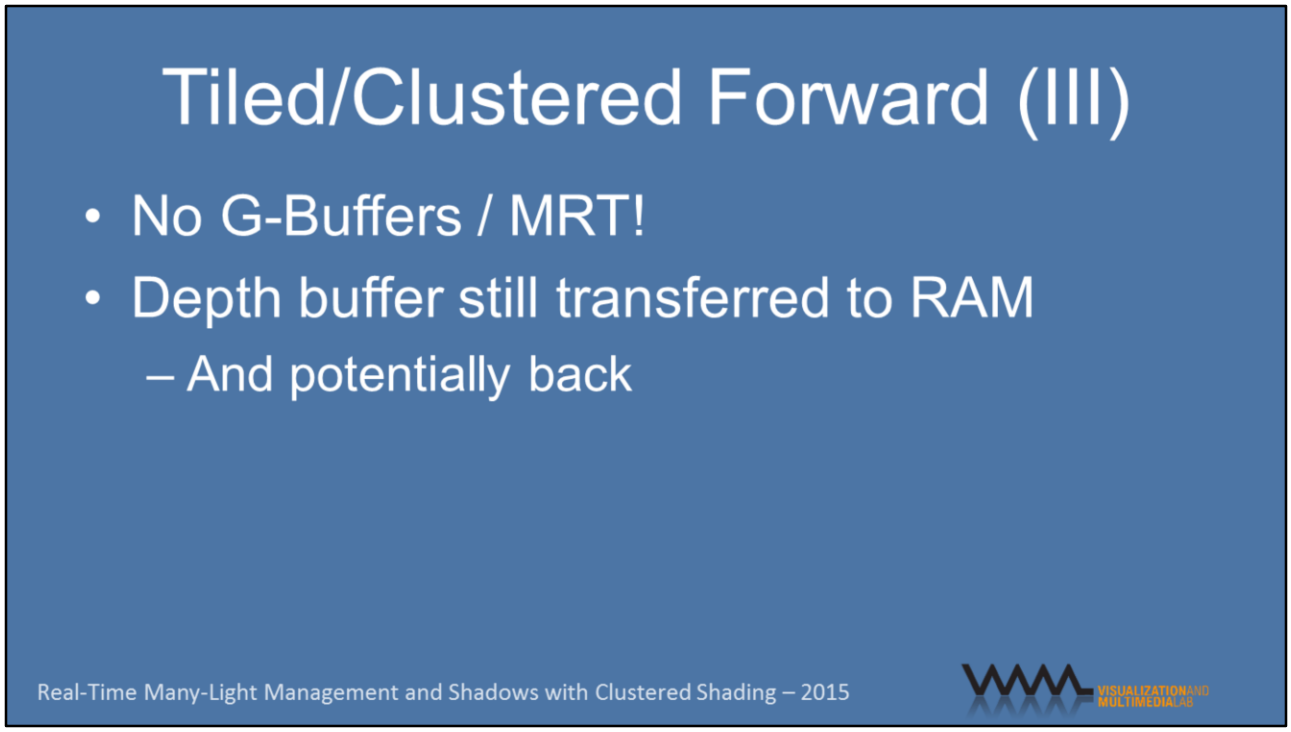
<contd.>

For completeness, it's also worth noting that Tiled Forward is also known as Forward+ in some publications.

Tiled/Clustered Forward (III)

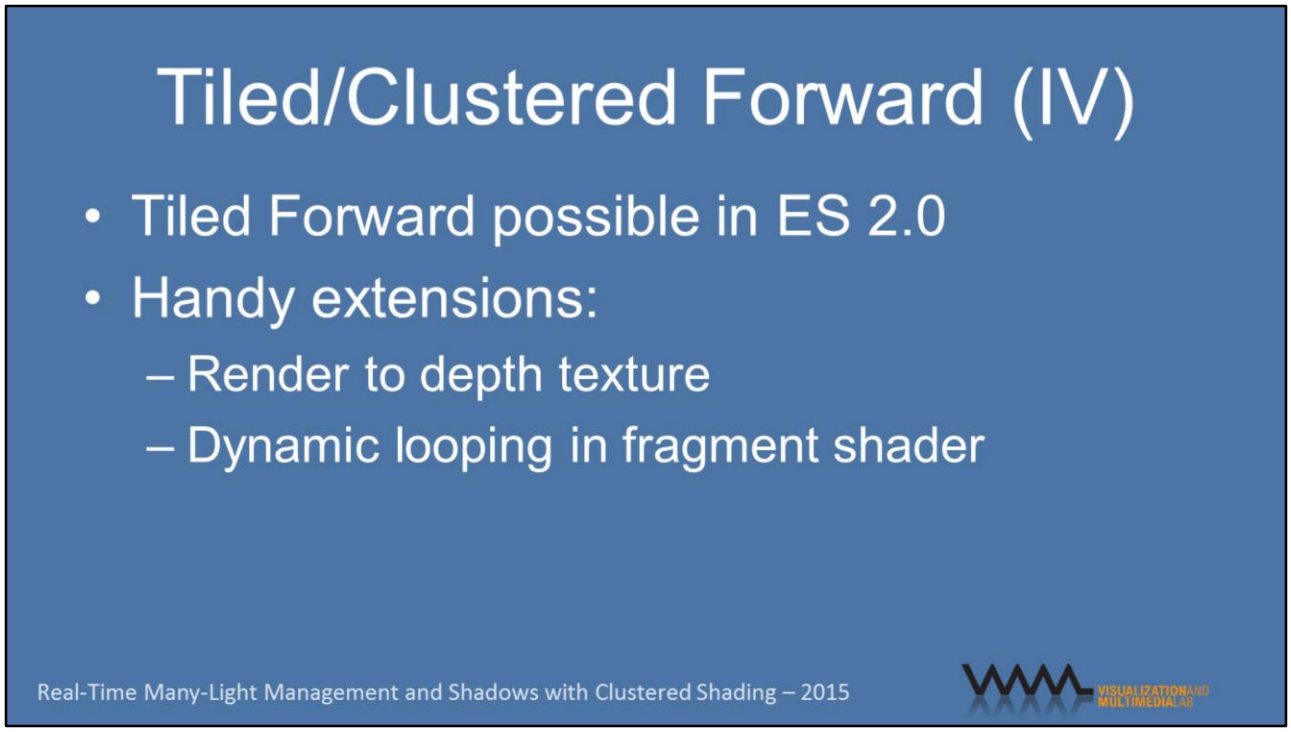
- No G-Buffers / MRT!

One of the key properties of the forward variants is that they do not involve the heavy G-Buffers. As such we also avoid requiring support for MRT. ▶▶▶



<contd.>

A small note: here we're still transferring the depth buffer from the on-chip storage to RAM, as we need access to the depth data during the light assignment. Further, if we want to use the depths during the forward shading pass to avoid overshading, the data needs to be transferred back as well.



Tiled Forward is, by the way, possible to implement with only OpenGL ES 2.0. There are some handy extensions that make this easier, though, mainly the ability to render to a depth texture, and being able to loop dynamically in the fragment shader.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clustered Forward							
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

As mentioned, the forward variation doesn't require G-Buffers or multiple render targets. On the other hand, with the preZ pass, at least two geometry passes are performed. The information from the PreZ pass can be reused to avoid overshading, albeit this comes at the cost of having to copy the depth buffer back to the on-chip storage on additional time. Also, when accessing the depth buffer from a texture the MSAA would have resolved, so one needs to be a bit careful with there.

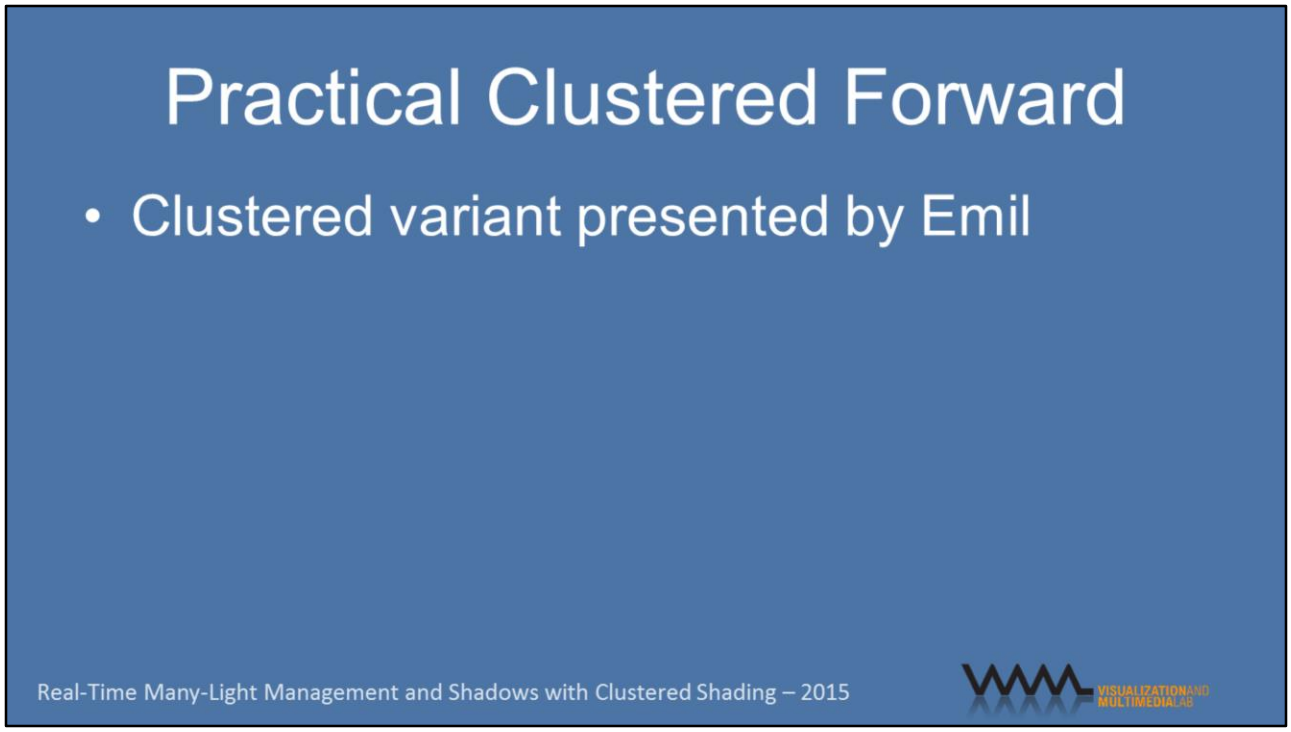
Being a forward method, blending is in principle supported. Some extra work might be required during light assignment as to ensure that the transparent surfaces also get correct light lists, since these surfaces are not present in the depth buffer from the preZ pass.

It's possible to implement the tiled forward method using OpenGL ES 2.0. Finding clusters is a bit trickier to do, and would require some more advanced features like compute shaders or the ability to write to arbitrary memory locations from a shader.

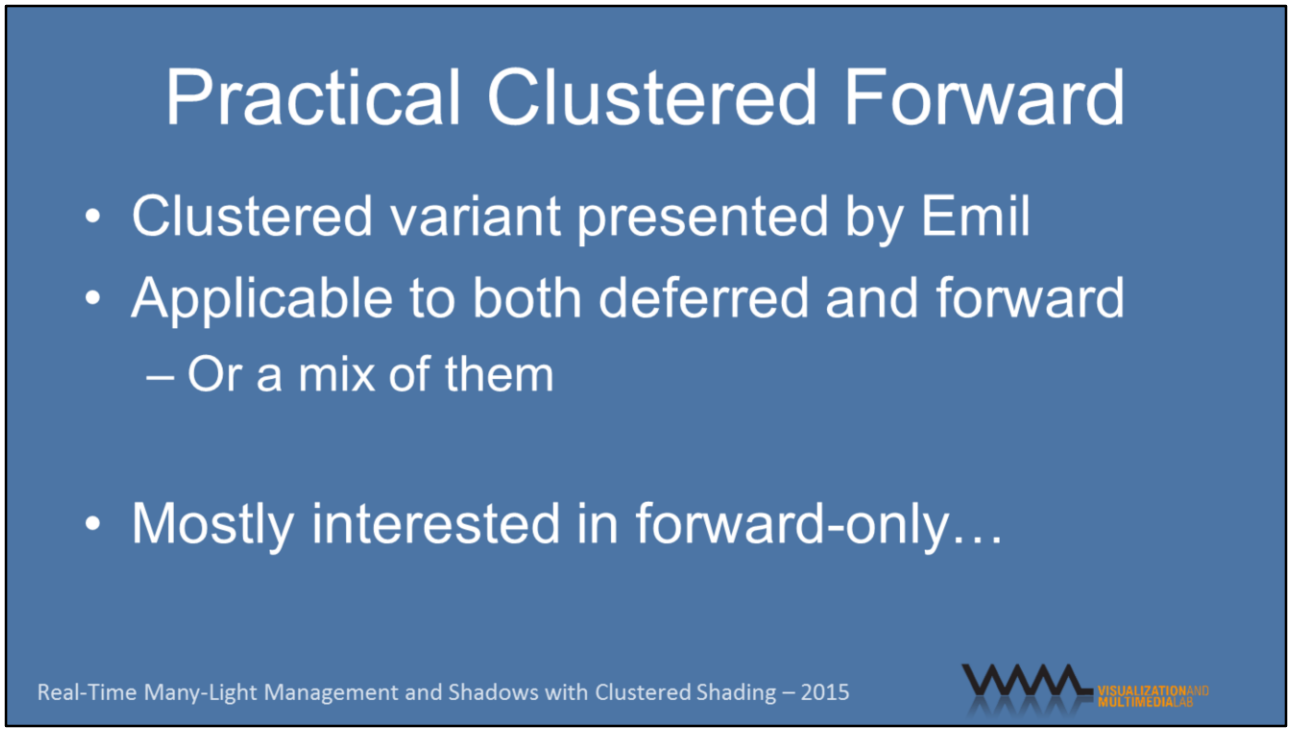
The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

This is something that the Practical Clustered-variation avoids.

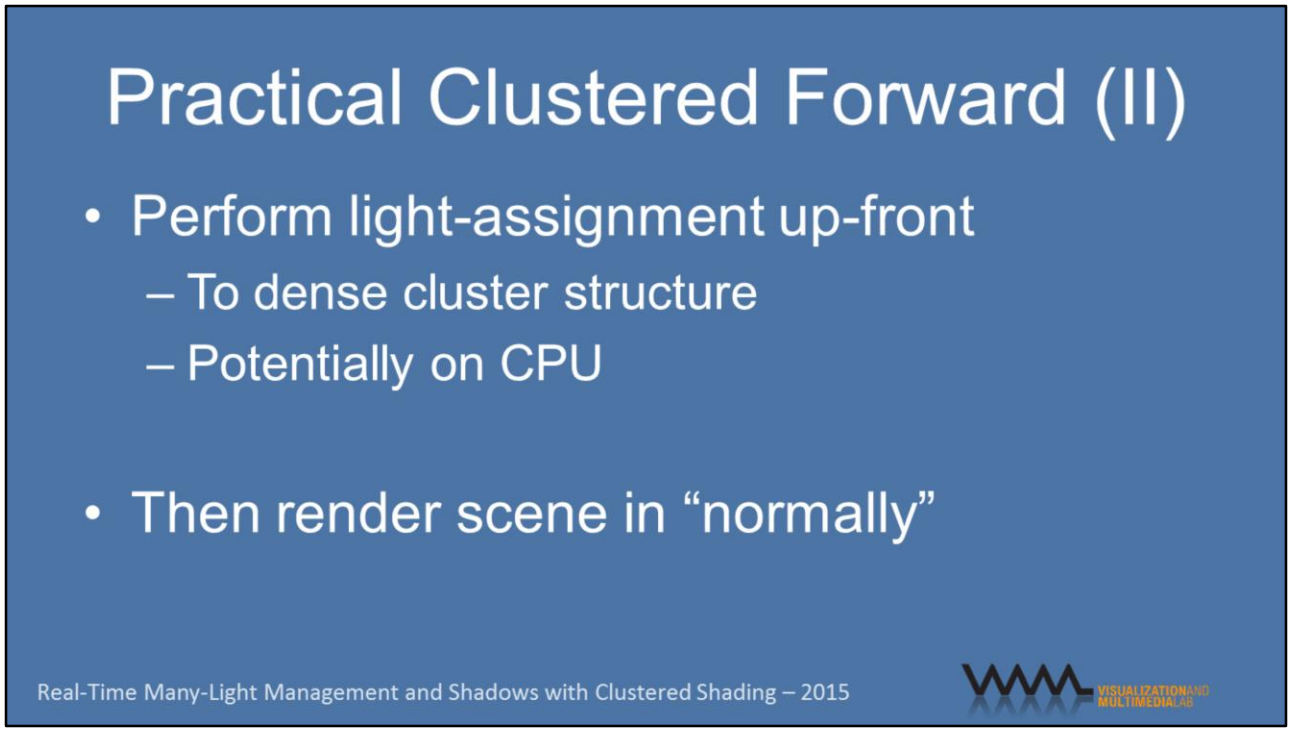


The Practical Clustered method Emil presented previously. ▶▶▶



<contd.>

As he mentioned, it's applicable to both deferred and forward shading, or even a mix of them. For this talk, I'm mostly interested in the forward-only variant, though – and based on the discussion of the previous technique, you can perhaps already guess why.



The key idea behind the method is to perform the light assignment up-front, into a dense cluster structure, potentially on the CPU. After this, we render the scene normally again.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Forward							
Deferred Tile Store							
Forward Light Stack							

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Deferred Tile Store							
Forward Light Stack							

I'll quickly summarize the properties of the practical forward method. We don't have any heavy G-Buffers, and compared to previous methods, we don't even have to transfer the depth buffer off-chip since there's a single geometry pass, but we pay for this by potentially getting overshading.

Blending is trivially supported, since the dense cluster structure is expected to cover the whole view frustum, in contrast to the sparse cluster structure of the previous method.

The overshading issues can be mitigated using standard tricks, such as front-to-back drawing and perhaps occlusion culling. Further, it can be avoided via an extra PreZ pass, rising the number of geometry passes to two. However, compared to previous method, the PreZ depth buffer can stay on-chip in TBR as we're never trying to access it via a texture or similar.

The Methods

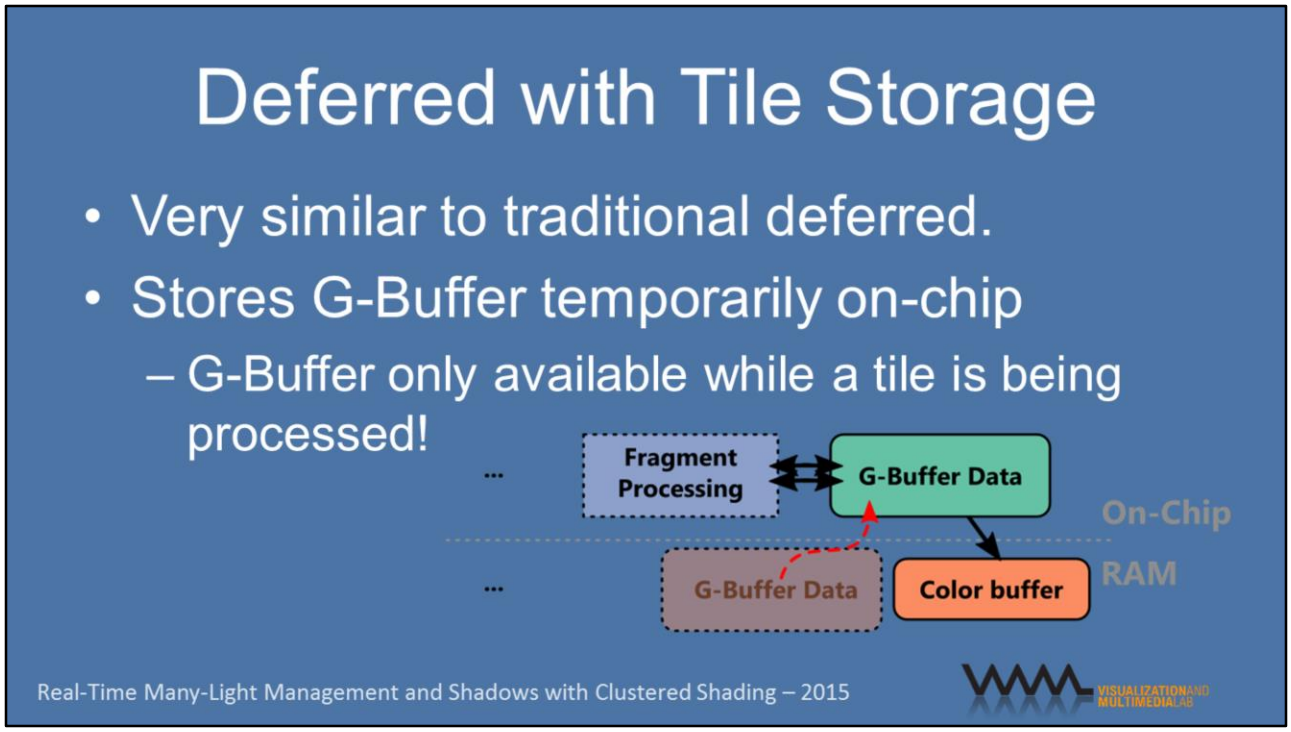
	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Deferred Tile Store							
Forward Light Stack							

The next method that I'll briefly present is called "Deferred with Tile Storage" and was presented by Sam Martin at SIGGRAPH 2013. It's an interesting method because it explicitly exploits the on-chip storage of the TBR architecture.

Deferred with Tile Storage

- Very similar to traditional deferred.

As indicated by its name, this method is very similar to the traditional deferred method presented earlier. ▶▶▶



<contd.>

The main difference is that the on-chip storage of the TBR architectures is used to temporarily hold the G-Buffer data.

Both the scene geometry and the proxy geometry of the light sources is submitted by the application in that order. Later, when a tile is being processed, the scene geometry is first rasterized to generate the G-Buffer data and the store it in the on-chip buffers. Immediately following this, the light proxy geometry will be rasterized and shaded. At this point the G-Buffer data for current tile is available in the on-chip buffer.

Deferred with Tile Storage (II)

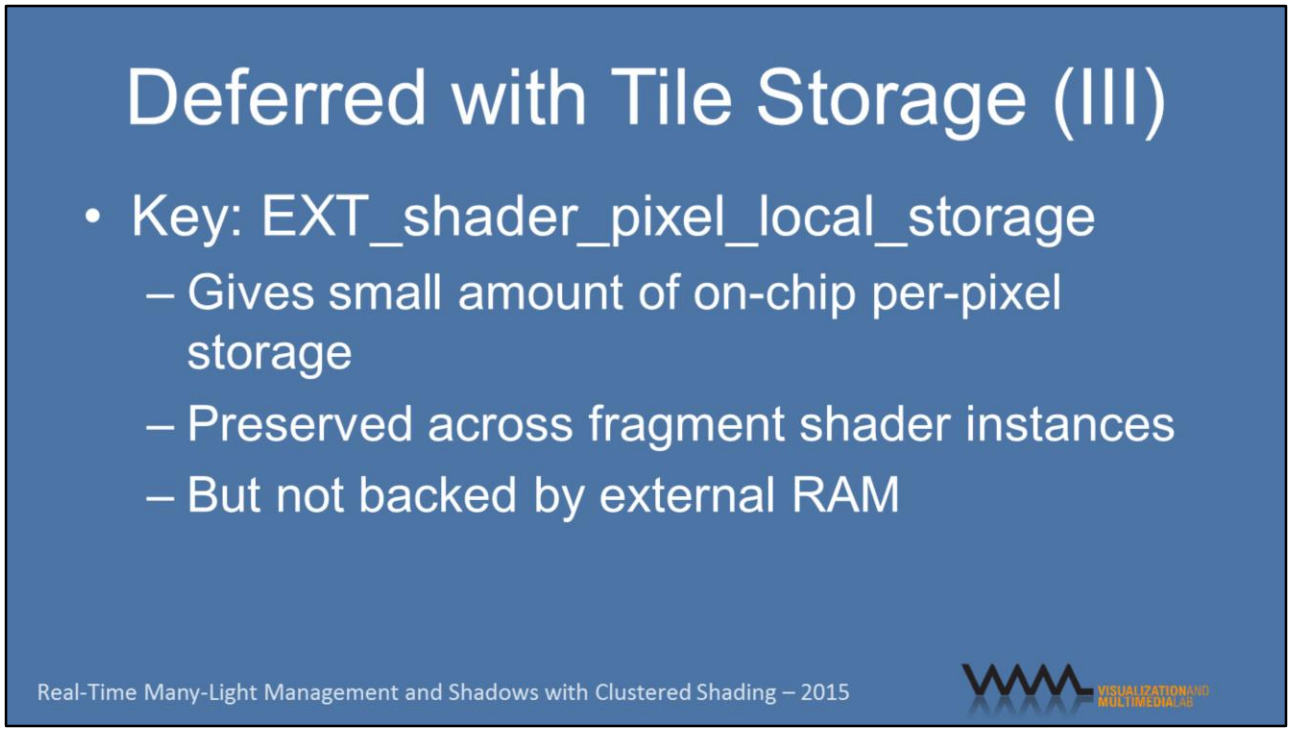
- Relies on OpenGL Extensions
  - EXT_shader_pixel_local_storage
  - ARM_shader_framebuffer_fetch
  - ARM_shader_framebuffer_fetch_depth_stencil

This technique relies on some OpenGL extensions, that enable manual access to the on-chip storage of the TBR architecture. ▶▶▶

Deferred with Tile Storage (III)

- Key: EXT_shader_pixel_local_storage

They key extension is this EXT_shader_pixel_storage one.



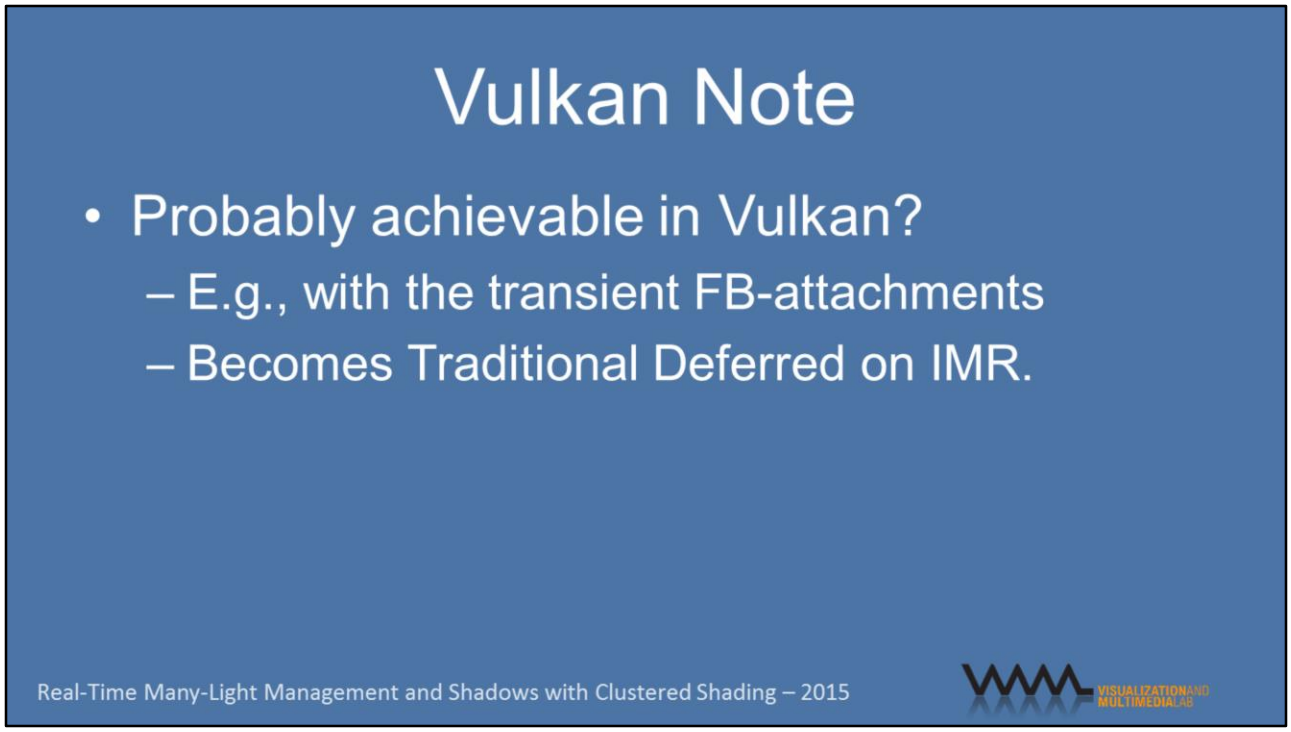
<contd.>

It enables the fragment shader to store a small amount of data per-pixel. This data is preserved across fragment shader instances, but not backed by external RAM.

So in our example, the G-Buffer is stored to this per-pixel storage, where it remains until we're done with the shading. But it's never transferred off-chip to some external buffer.

The on-chip storage is a bit finicky. So certain operations, such as writing to the shaders normal color output(s) will destroy the per-pixel storage contents. It's also incompatible with MSAA.

For some additional details, see Sam Martin's original presentation from SIGGRAPH 2013, and read the extension spec.



Time for a small Vulkan note. If you attended the Next-Gen course on Tuesday, you might have heard of the transient FB-attachments, that is FB-attachments not backed by an off-chip memory store. I'm guessing that you can implement this method using those. As a bonus, the method will then transparently work on IMR systems as well, where it essentially becomes the traditional deferred method.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Deferred Tile Store							
Forward Light Stack							

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Forward Light Stack							

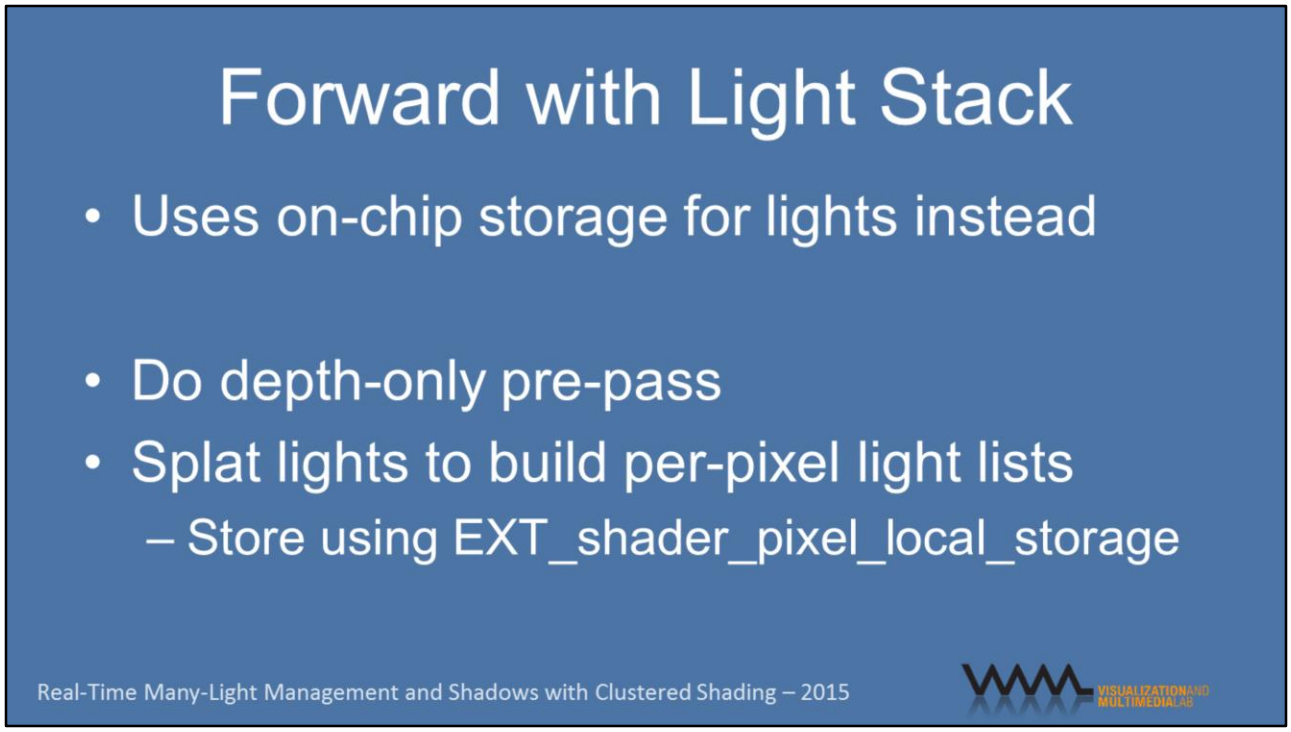
Despite being a deferred method, this method doesn't need any off-chip G-Buffers, as the G-Buffer data stays on-chip. Like other deferred methods, it needs only a single geometry pass and doesn't suffer from overshading issues. It also shares the draw backs of other deferred methods, namely that MSAA is tricky and that transparent surfaces can't be represented in a G-Buffer.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Forward Light Stack							

With this, it's time for the final method in this list.

This method was presented at the same time as the previous one, Deferred with Tile Storage. It also uses the on-chip storage via the same extension, but instead of storing the G-Buffers there, it uses the storage for the light lists. ▶▶▶



First, a depth-only pre-pass is performed to build up the per-tile depth buffers. Next, lights are rendered using proxy-geometry on top of that depth-buffer. During this pass, the fragment shader is used to build per-pixel light lists into the local on-chip storage provided by the EXT_shader_pixel_local_storage extension.

Forward with Light Stack (I)

- Forward render geometry

Finally, a forward pass is performed. Since the light lists are now available in the local storage for each pixel, the fragment shader can access these quite efficiently.

Forward with Light Stack (II)

- Limited light list size

There's a few gotchas with this method. First, the size of the per-pixel light lists is limited by the amount of storage that the EXT_shader_pixel_local_storage extension provides to each pixel. On the ARM Mali T6xx GPUs that is only about 16 bytes. ▶▶▶

Forward with Light Stack (II)

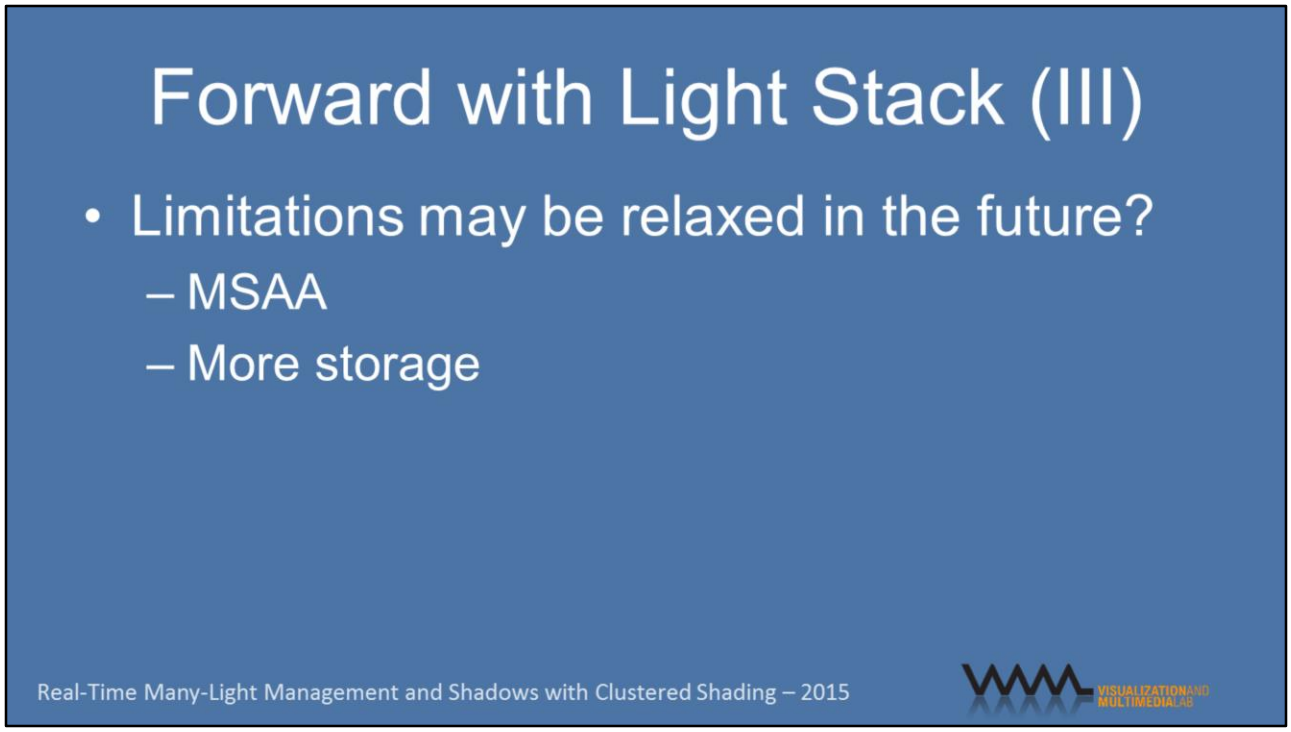
- Limited light list size
- Incompatible with MSAA

Second, the extension is (for now) incompatible with MSAA, meaning that the per-pixel storage cannot be used on a render target that has HW-MSAA enabled. ▶▶▶

Forward with Light Stack (II)

- Limited light list size
- Incompatible with MSAA
- Blending possible
  - "By hand"
  - Requires some bytes per per-pixel storage

On the other hand, Martin et al. demonstrate that blending is possible with this method, although it has to be done "by hand", since writing to shader's normal color outputs would destroy the contents of the per-pixel storage and thereby invalidate the per-pixel light lists. So, we need to allocate a few bytes of space in the per-pixel storage to hold the results of the blending temporarily. Additionally, a final pass is needed to "flush" this result from the per-pixel storage to the actual framebuffer – this is done by copying the value from the per-pixel storage to the shader's color output.



In their talk, Sam Martin mentions that some of the limitations may go away in the future. So, for instance a new improved extension may allow the per-pixel storage to be combined with MSAA, and future devices may provide for a larger per-pixel storage.

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Forward Light Stack							

The Methods

	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd.	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd.	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Fwd. Light Stack	✓	Color	2	PreZ	×	✓	ext

Much like the previous method(s), the Forward Light Stack keeps most of the framebuffer data in the on-chip tile storage, only eventually transferring the resulting colors to RAM. The Forward Light Stack requires two geometry passes, once to prime the depth buffer so that the per-pixel light lists can be generated efficiently. Unlike the deferred method also using the per-tile storage, it does support blending, but the blending has to be done manually and requires some additional space from the per-pixel storage, which further limits the maximum length of the light lists.

The Methods

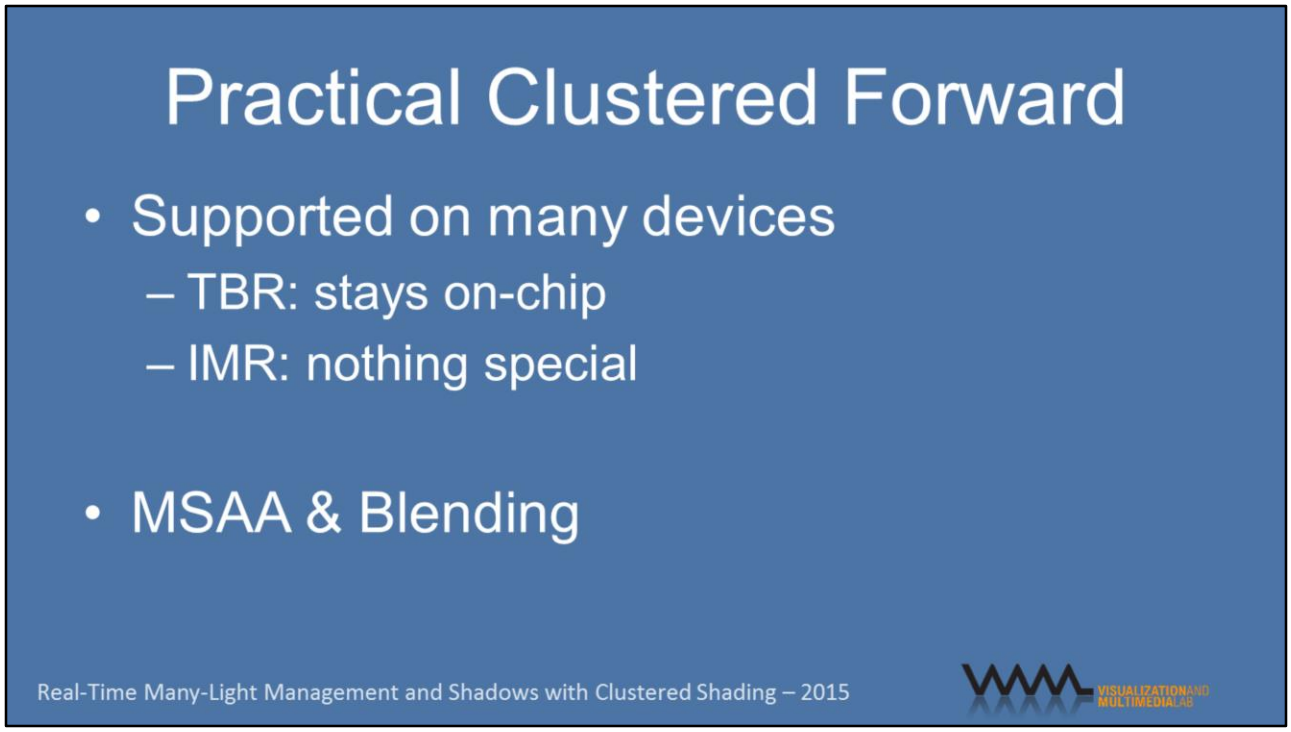
	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd.	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Fwd. Light Stack	✓	Color	2	PreZ	×	✓	ext

With this, we've covered the listed methods briefly, and are almost ready to move on to the last part of the presentation.

The Methods

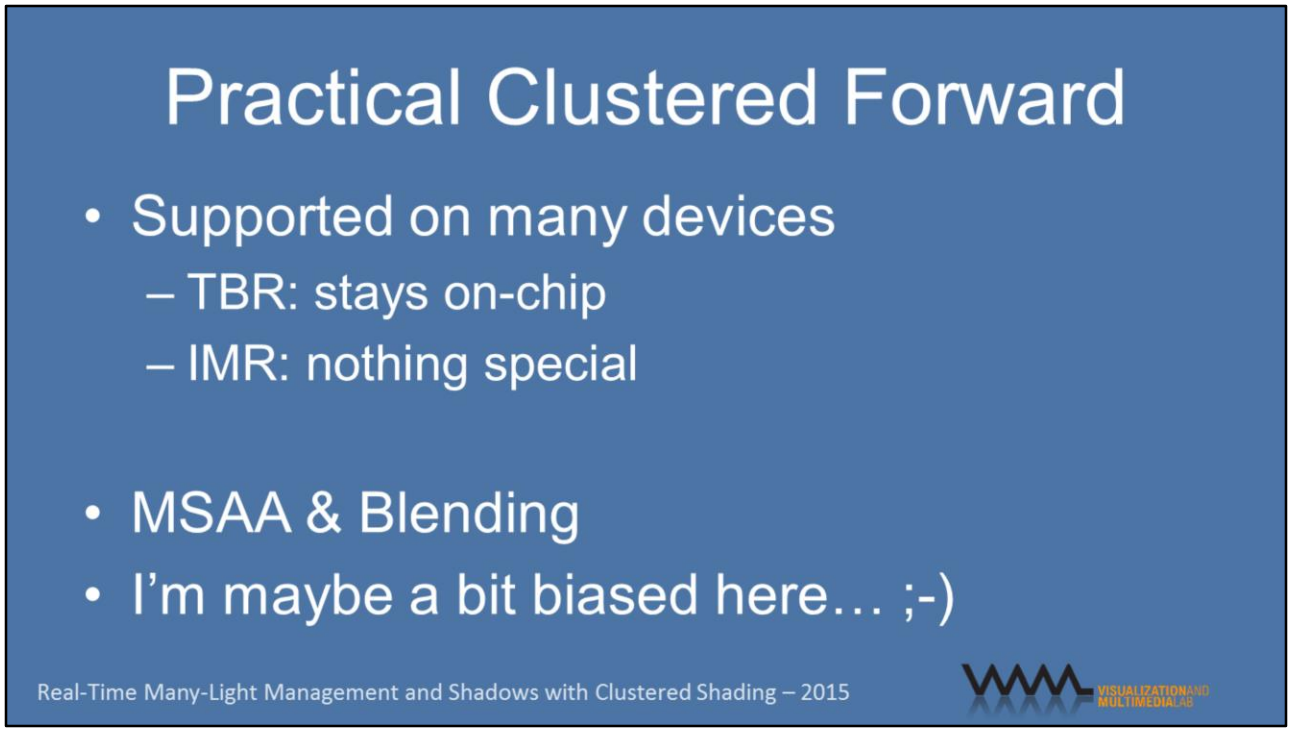
	Many Light	Off-Chip Buffers?	geometry passes	over-shading	MSAA	Blend.	GL|ES
Plain Fwd.	×	Color	1	Yes	✓	✓	Any
Trad. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Def.	✓	G-Buffers	1	No	×	×	3.0
Clust. Fwd	✓	Depth	2	PreZ	✓*	✓*	2.0
Practical Fwd	✓	Color	1	Yes	✓	✓	2.0
Def. Tile Store	✓	Color	1	No	×	×	ext
Fwd. Light Stack	✓	Color	2	PreZ	×	✓	ext

The method I will be talking about in the last part is a variation of the Practical Clustered Forward method, where I do the clustering slightly differently. But I'll quickly summarize some of the properties that make this method a good choice for mobile devices in my opinion.



So, on the TBR platforms, we don't need to have any off-chip buffers other than the final color buffer. The method works transparently on IMR too, we're not doing any fancy-pants architecture-dependent things. This is great for a number of reasons, at the very least because it's possible to run and debug your renderer on a desktop GPU with all the tools available there.

Finally, we get MSAA and blending. The former is very helpful when rendering at lower-than-native resolution. ▶▶▶



<contd.>

The reviewers made us mention this – so: small caveat. I might be a bit biased here. ;-)

Part 3

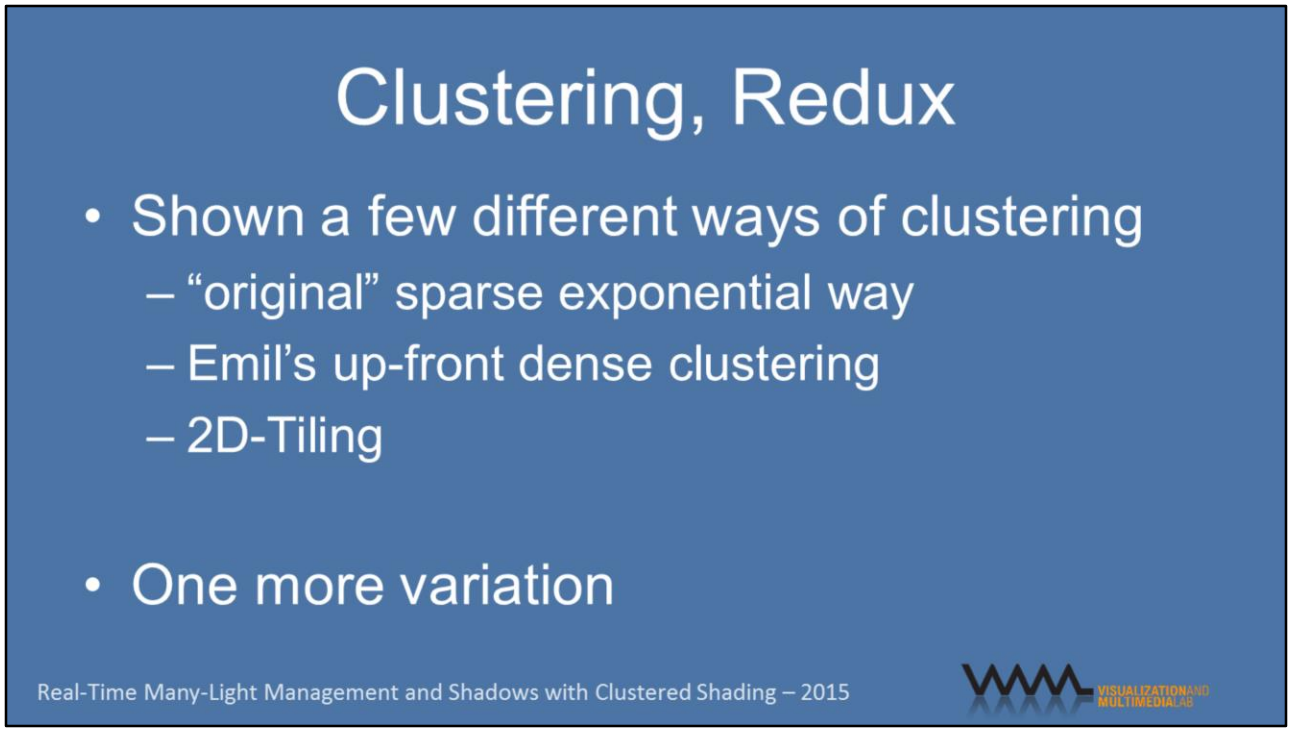
CUSTOM PRACTICAL CLUSTERED FORWARD.

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

Clustering, Redux

- Shown a few different ways of clustering
  - "original" sparse exponential way
  - Emil's up-front dense clustering
  - 2D-Tiling

We've shown you a few different ways of clustering view-samples for efficient light assignment. ▶▶▶



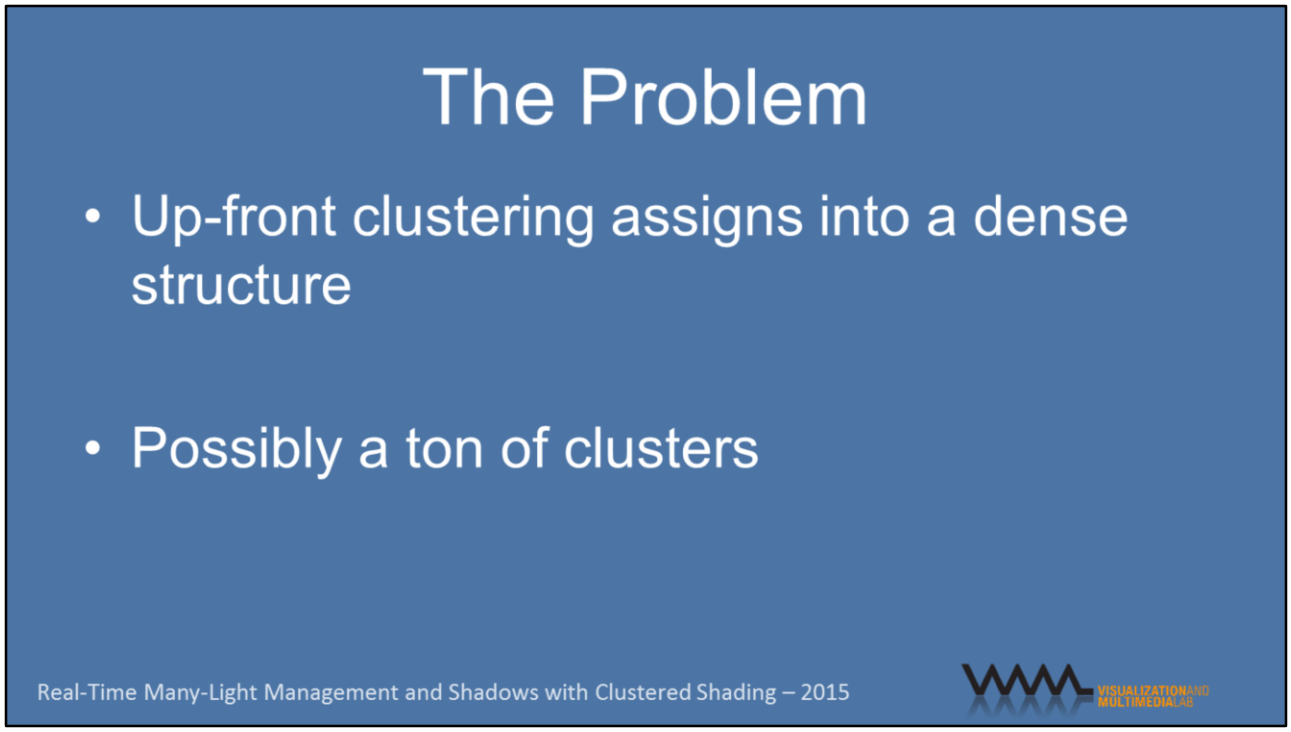
<contd.>

I’ll show you one more way.

The Problem

- Up-front clustering assigns into a dense structure

So, this builds on Emil's practical clustering, with the dense data structure. ▶▶▶



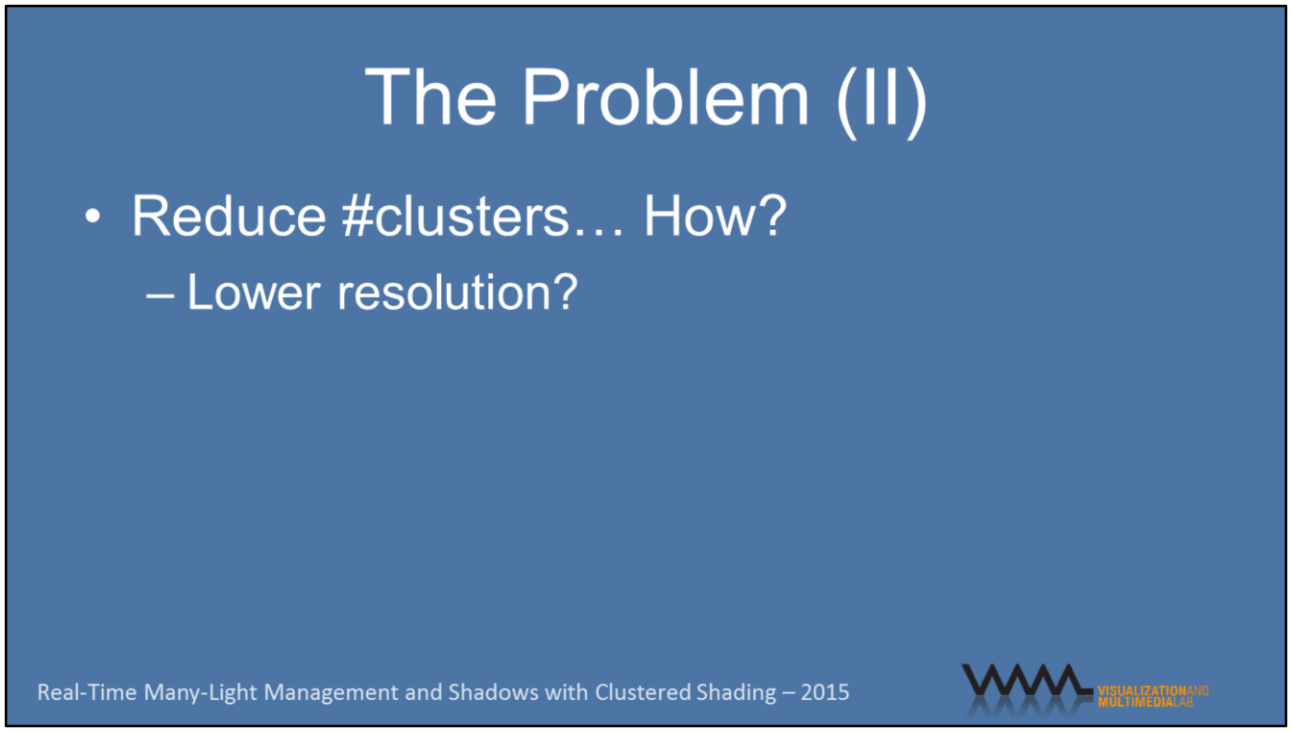
<contd.>

A dense 3D structure potentially results in a ton of clusters, which is a bit problematic, since that increases the cost of the light assignment.

The Problem (II)

- Reduce #clusters… How?

So, we'd like to reduce the number of clusters somehow.

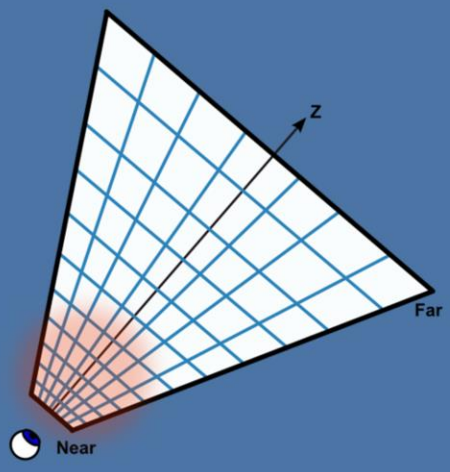


<contd.>

One way is to lower the resolution, but that gives us worse light assignment and more shading work. So, there's a trade-off here. ▶▶▶

The Problem (II)

- Reduce #clusters… How?
  - Lower resolution?
- Observation:
  Especially problematic close to the camera

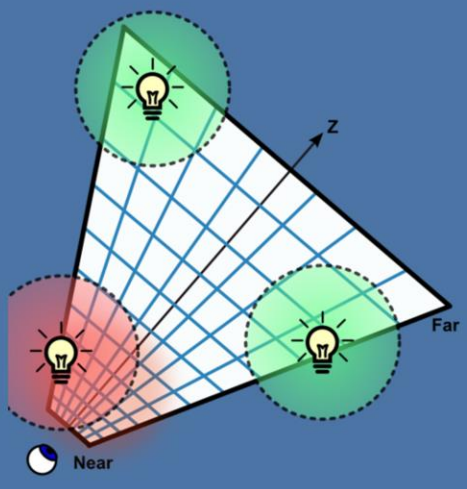


<contd.>

An observation at this point is that the problem is especially … well … problematic close to the camera. ▶▶▶

The Problem (II)

- Reduce #clusters… How?
  - Lower resolution?
- Observation:
  Especially problematic close to the camera

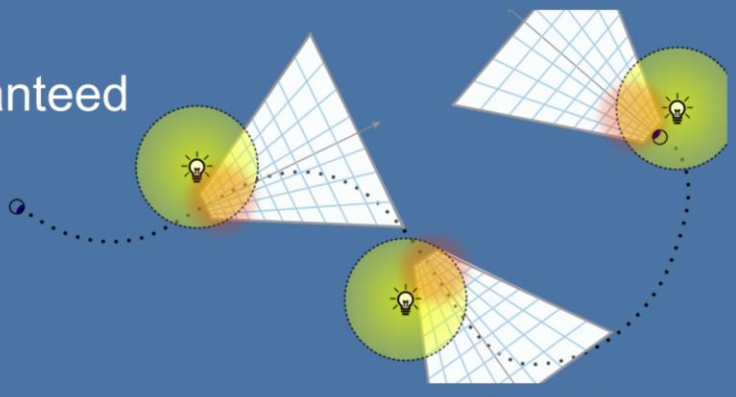


<contd.>

Here we get a lot of tiny clusters, and a single light source can overlap quite a lot of them.

The Problem (III)

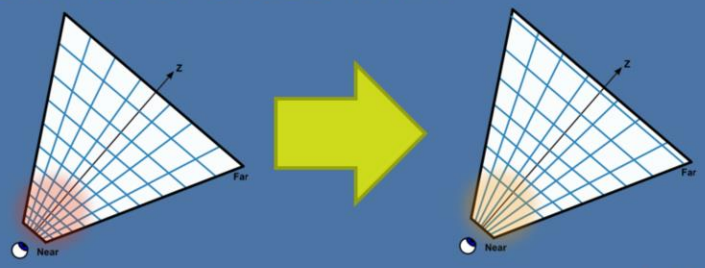
- Lot's of unnecessary work during light assignment.
  - Almost guaranteed to occur…



That results in a lot of unnecessary work during light assignment, since all those tiny clusters contain more or less the same information. And, if you let your camera move around freely, this is more or less guaranteed to occur, as the camera can move into a light's volume. Besides, I'd like to aim for a somewhat robust method without this kind of gotchas.

The Problem (IV)

- Discussed previously
  - E.g., move back first Z-subdivision
  - Still get a lot of slices in XY



The problem has been mentioned earlier, and one solution you've seen is to move back the first subdivision in the depth direction. This certainly helps, but still leaves a lot of slices in the XY direction.

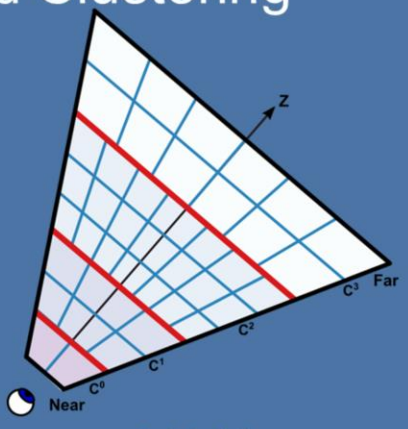
Cascaded Clustering

- Different solution: Cascaded Clustering

So, I took a different approach: cascaded clustering. ▶▶▶

Cascaded Clustering

- Different solution: Cascaded Clustering
- Subdivide frustum into cascades and select individual resolution for each



Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

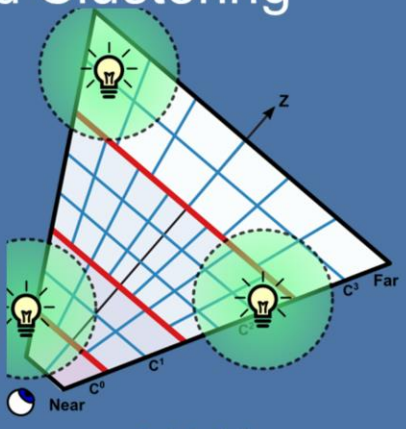
VISUALIZATION AND MULTIMEDIA LAB

<contd.>

Instead of using a single clustering across the whole view frustum, I subdivide it into a few “cascades”, and select the resolution for each of them individually. ▶▶▶

Cascaded Clustering

- Different solution: Cascaded Clustering
- Subdivide frustum into cascades and select individual resolution for each



Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

VISUALIZATION AND MULTIMEDIA LAB

<contd.>

And, yeah, now the density of clusters is much more even. ▶▶▶

Cascaded Clustering (II)

- Select resolutions to give
  - Approx. cubical clusters with NxN footprint
  - But enforcing a minimum size of each cluster

I select the resolution of each cascade so that I get approximately cubical clusters with a NxN pixel footprint in the frame buffer. This is very much the same as we've done earlier. But additionally, I clamp the cluster's size to a specific minimum. ▶▶▶

Cascaded Clustering (II)

- Select resolutions to give
  - Approx. cubical clusters with NxN footprint
  - But enforcing a minimum size of each cluster
- Currently use 12 cascades
  - Tweakable…

<contd.>

So, in my tests, I ended up using 12 cascades. Of course, this depends on the ratio between your near and far planes, so it's something you want to be able to tweak a bit.

Quick Results

Quick Results

- 192 lights
- Clustering: 0.35ms
  - Galaxy Alpha
  - 3500 non-empty clusters
  - with 3.5 lights avg. (worst = 20 lights)

With 192 lights, the clustering takes around 0.35 ms on my Galaxy Alpha device. Non-empty clusters contain on average 3.5 lights, with the worst case in some views going up to 20. That's for very few pixels on the other hand, so performance isn't too bad even then.

Quick Results

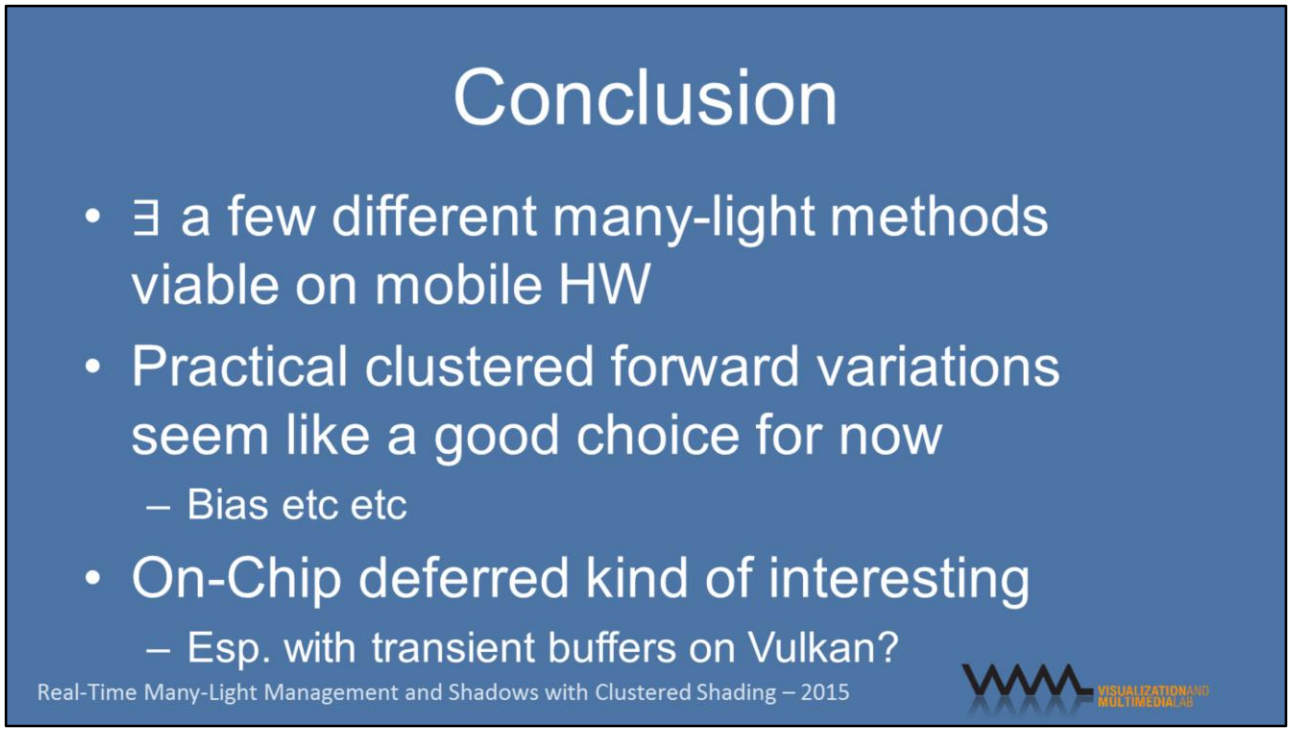
- 30ms per frame average
  - Worst: ~35ms
  - Small lights + limited overlap
- With PreZ

Real-Time Many-Light Management and Shadows with Clustered Shading – 2015

VISUALIZATION AND MULTIMEDIA LAB

Rendering performance is around 30ms per frame on average, with a slightly worse worst case.

I use a PreZ pass – this improves the overall performance a bit in my case. That's probably related to the fact that I don't really do any fancy culling or even front-to-back rendering, but just throw the whole scene at the GPU.



So, with this I'm pretty much at the end of my talk.

I hope to I've shown that there are a few different many-light methods that are viable on mobile HW. To me, the practical clustered forward variation seem like an overall decent choice, for reasons I listed earlier.

There are a few things in the future that seem quite interesting too, so, for instance, I'm looking forward to be able to experiment with the transient buffers in Vulkan, assuming this stuff eventually becomes available for normal mortals.

References

- Tiled Deferred Blending, Ladlani, 2014
- The Revolution in Mobile Game Graphics, Martin and Bjørge 2014
- Challenges with High Quality Mobile Graphics, Martin et al. 2013
- An Evolution of Mobile Graphics, Shebanow 2013
- Performance Tuning for Tile-Based Architectures, Merry 2012
- Courses from earlier this week:
  - Moving Mobile Graphics
  - An Overview of Next-Generation Graphics APIs
- Forward+: Bringing deferred lighting to the next level, Harada et al. 2012
- Light Indexed Deferred Lighting, Treblico 2007
- Our work on Tiled / Clustered Shading:
  - Clustered Deferred and Forward Shading, Olsson et al. 2012
  - Tiled and Clustered Forward Shading, Olsson et al. 2012
  - Tiled Forward Shading, Billeter et al. 2013

(This slide show brought to you at approx. 0.057 SPS [slides per second])

We'll get the slides online, and you can find the references there