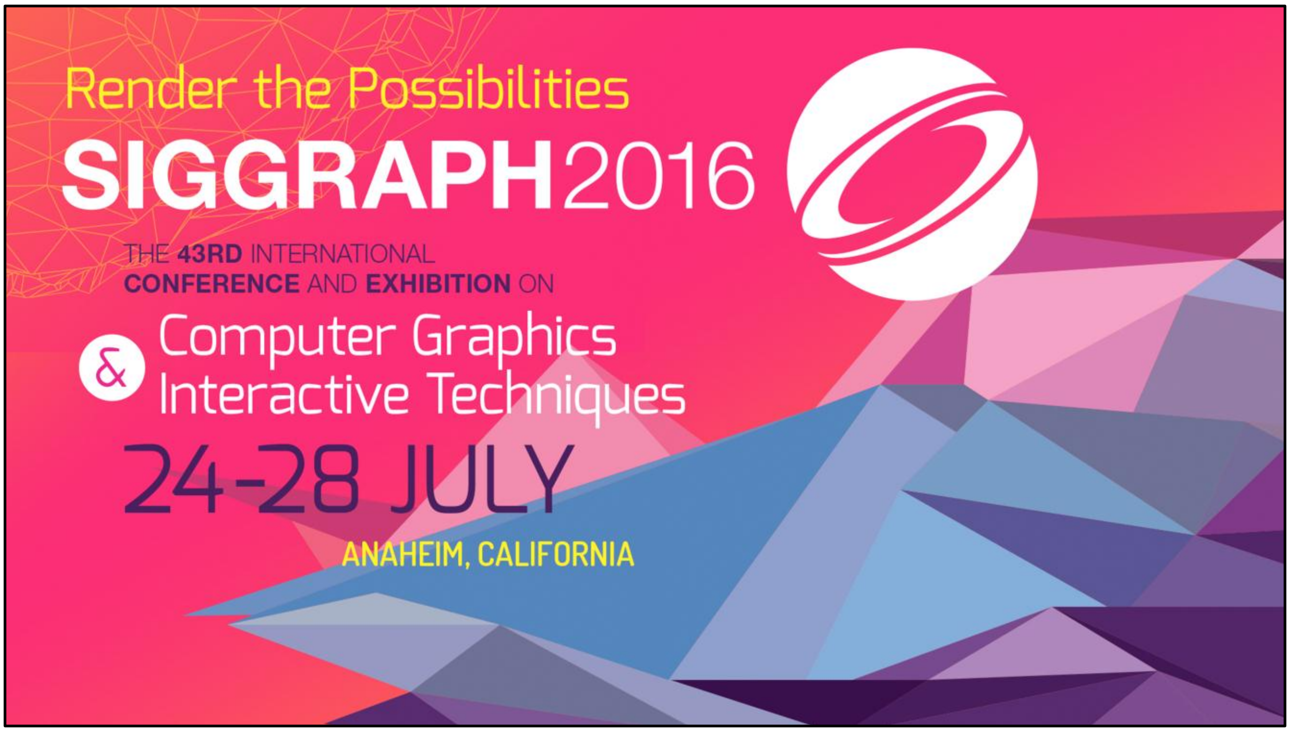


Render the Possibilities

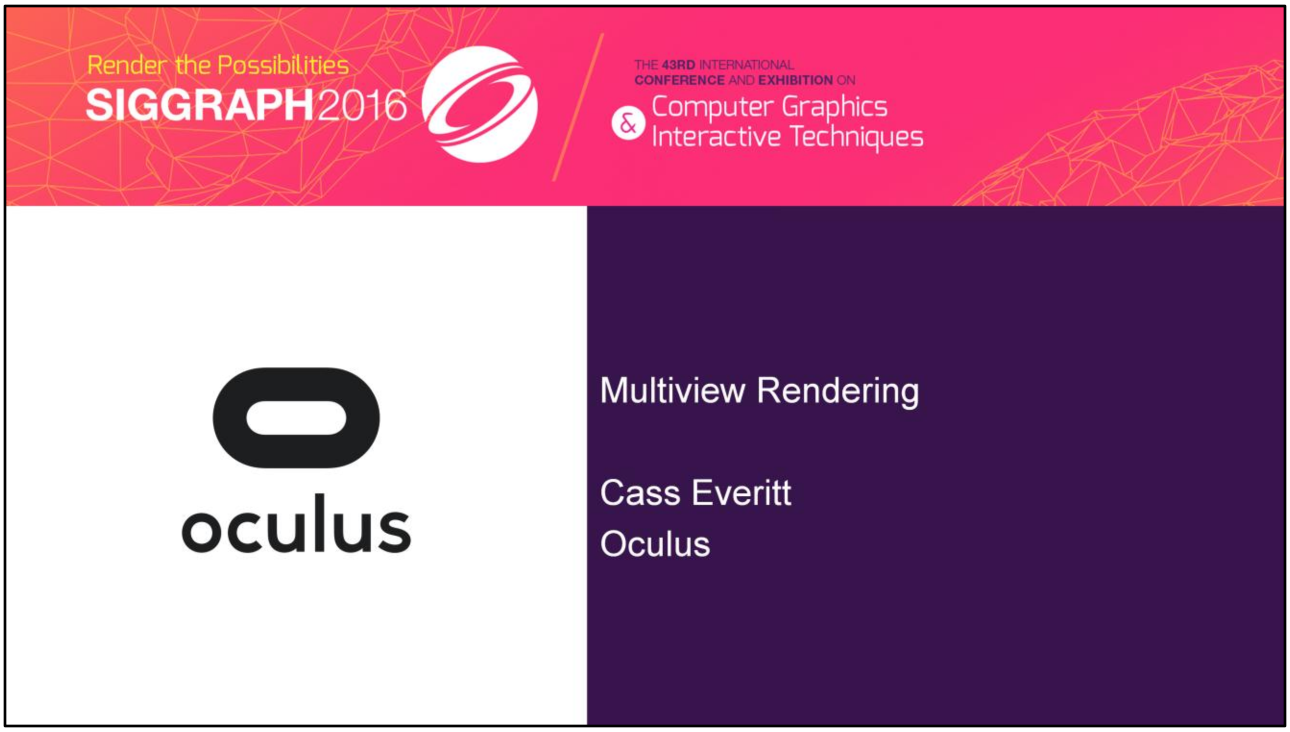
SIGGRAPH2016

THE **43RD** INTERNATIONAL
CONFERENCE AND **EXHIBITION** ON

& Computer Graphics
Interactive Techniques

24-28 JULY

ANAHEIM, CALIFORNIA



Render the Possibilities

SIGGRAPH2016

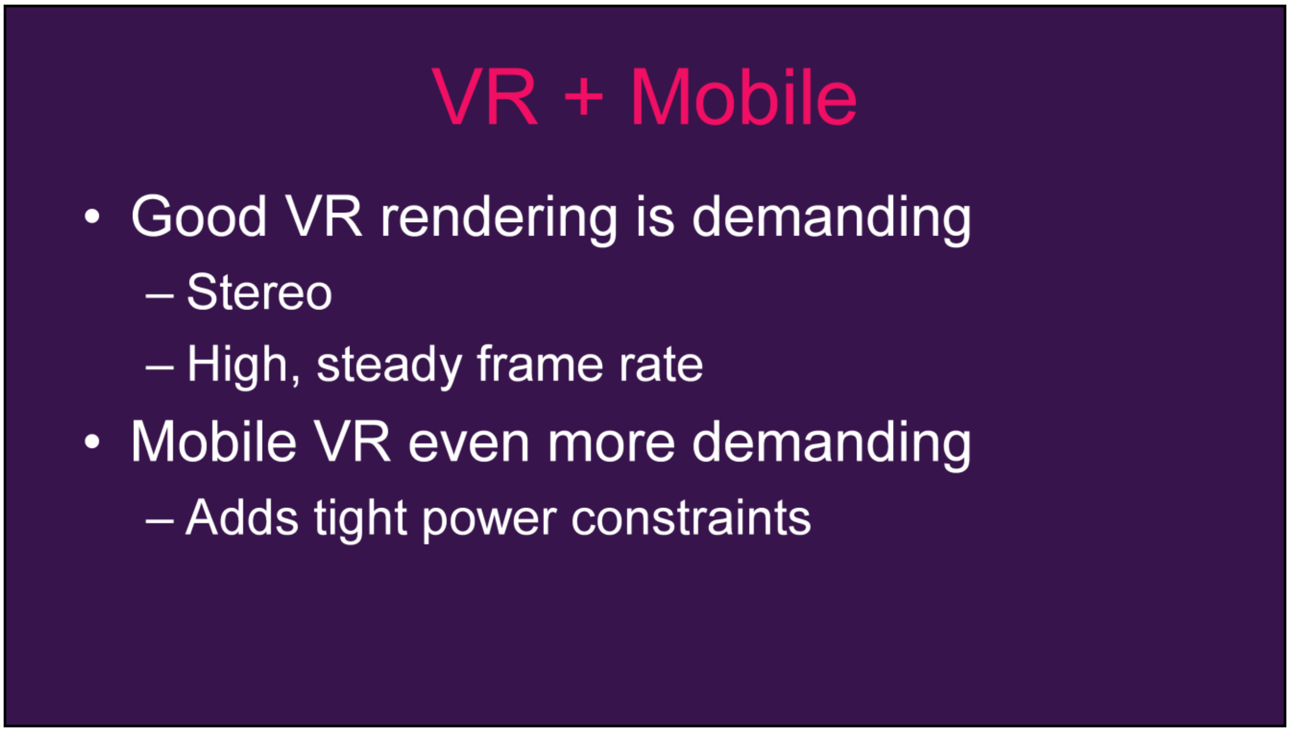
THE 43RD INTERNATIONAL CONFERENCE AND EXHIBITION ON

Computer Graphics & Interactive Techniques

oculus

Multiview Rendering

Cass Everitt
Oculus



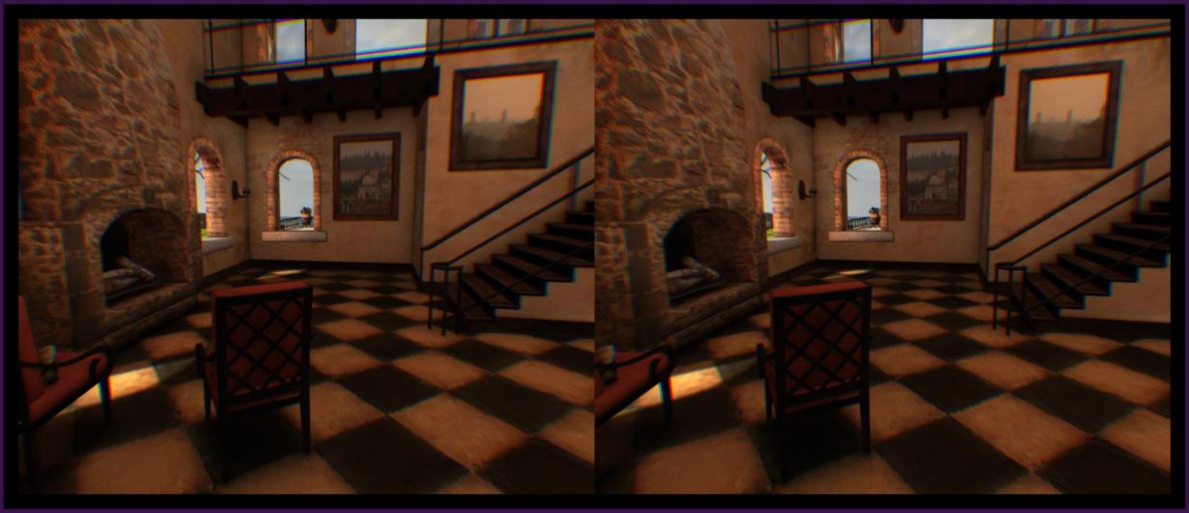
VR + Mobile

- Good VR rendering is demanding
  - Stereo
  - High, steady frame rate
- Mobile VR even more demanding
  - Adds tight power constraints

VR is challenging in the best of circumstances.
It implicitly requires stereo.
Presence requires a steady frame rate and very low latency.
Mobile VR adds system-level constraints.
- There’s the obvious challenge of battery powered devices with limited heat dissipation.
- But we also must live with constraints of devices whose primary function is not VR.
- And thus where increased bill-of-materials for VR-only hardware features may be a difficult sell.

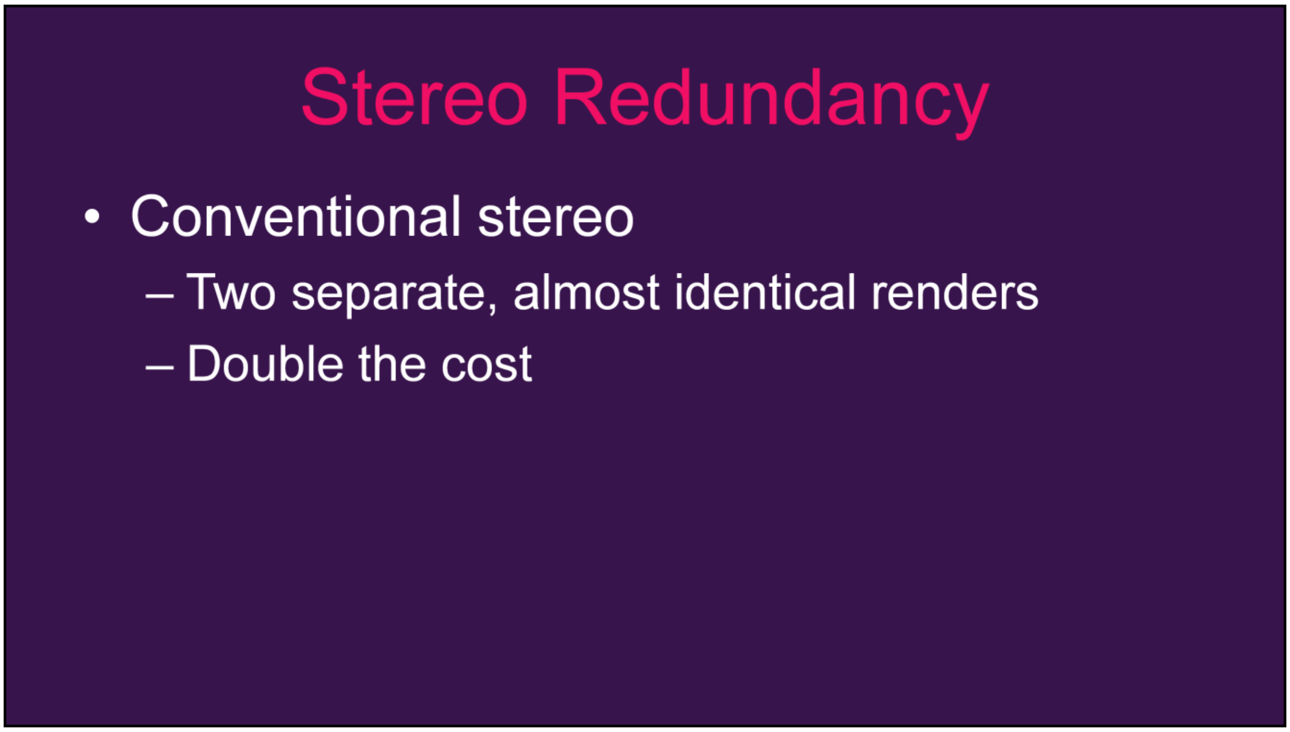
So… software-only features that can significantly improve efficiency are appealing.

Stereo Redundancy



Take a second to look at what a stereo pair for a nominal 3D scene looks like.
Pretty similar, huh?
Not surprising. Same scene, same time, almost the same camera position and orientation.

Surely there's some efficiency to be extracted from eliminating redundancy here!



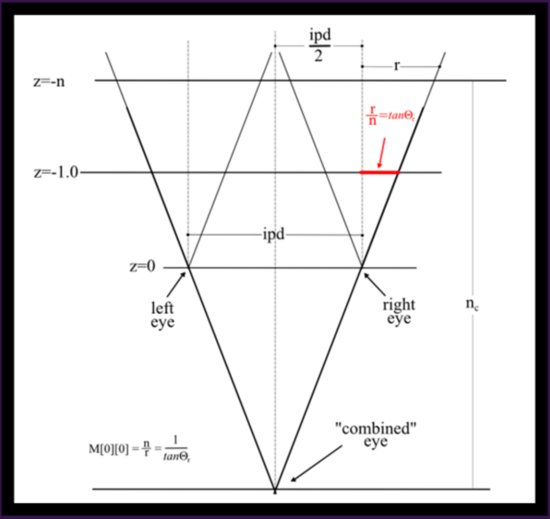
Stereo Redundancy

- Conventional stereo
  - Two separate, almost identical renders
  - Double the cost

Generating two almost identical images actually costs approximately 2x with conventional stereo.
The driver doesn’t know the frames are almost identical, so there’s not much to be done there.

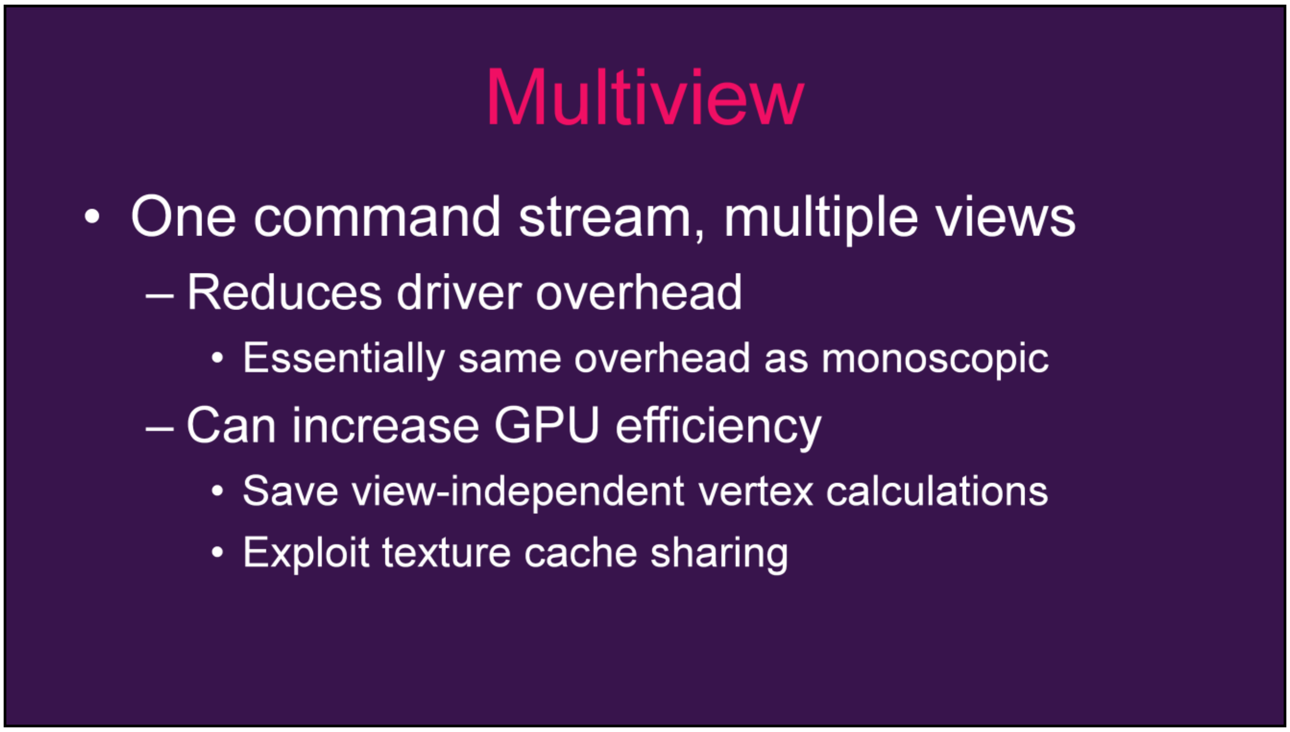
App Stereo Overhead

- Easy app optimization:
  - Combined frustum culling https://goo.gl/kdH2ur



$\frac{ipd}{2}$
r
z=-n
$\frac{r}{n}=\tan\Theta_r$
z=-1.0
ipd
z=0
left eye
right eye
n_c
"combined" eye
$M[0][0] = \frac{n}{r} = \frac{1}{\tan\Theta_r}$

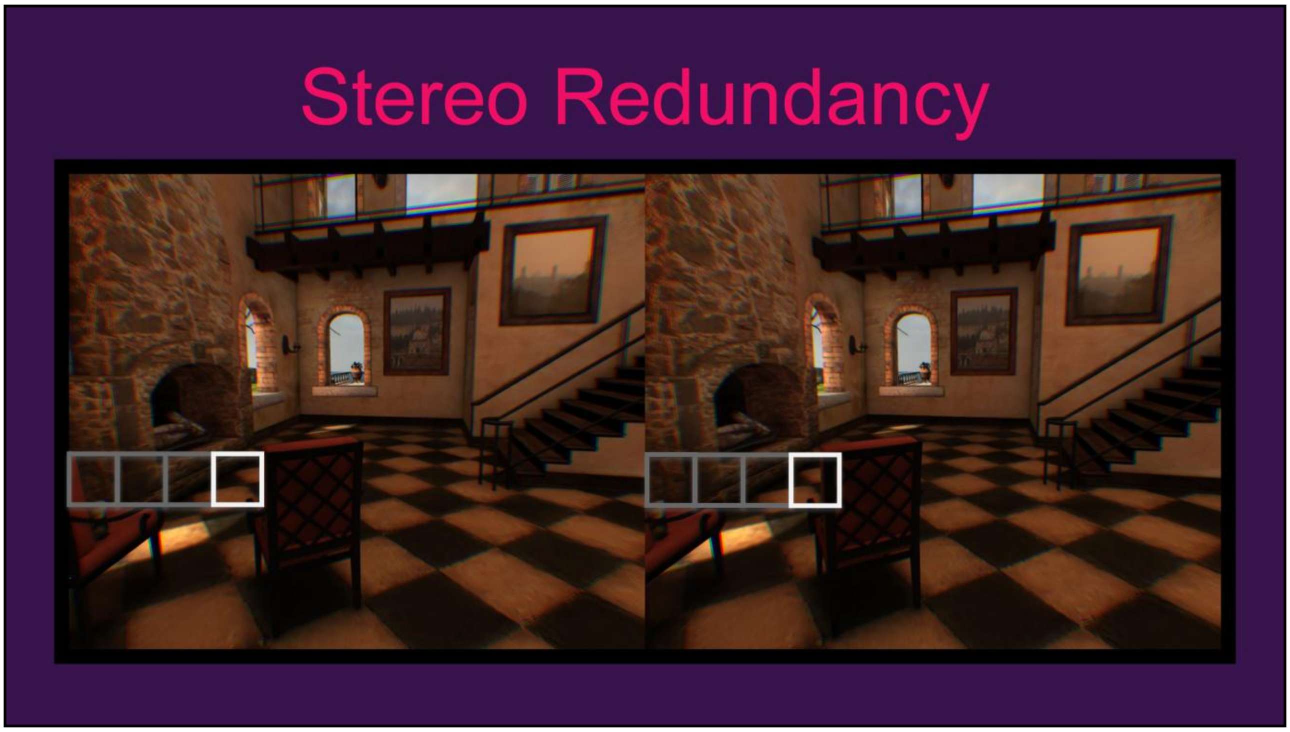
Even without API support though, if the app knows it's doing stereo rendering, it can eliminate a double-traversal of its internal scene representation.
Apps and engines usually build command stream data structures that can be executed by a render thread.
Instead of building a command stream for each eye, it's simple to build a single one that will work for both eyes. Each eye with it's own view and projection transform.
You can still do frustum culling during the traversal. You just need to use a "combined eye" frustum that contains both of the eye frusta.
The math for this is described in the link there, and it's also available in the Oculus SDK source.



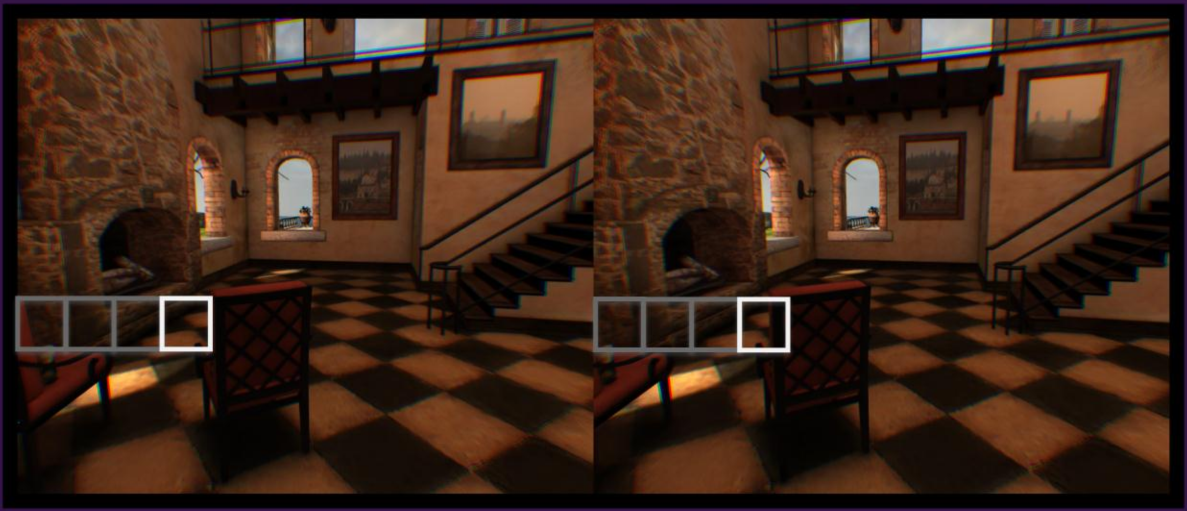
Multiview

- One command stream, multiple views
  - Reduces driver overhead
    - Essentially same overhead as monoscopic
  - Can increase GPU efficiency
    - Save view-independent vertex calculations
    - Exploit texture cache sharing

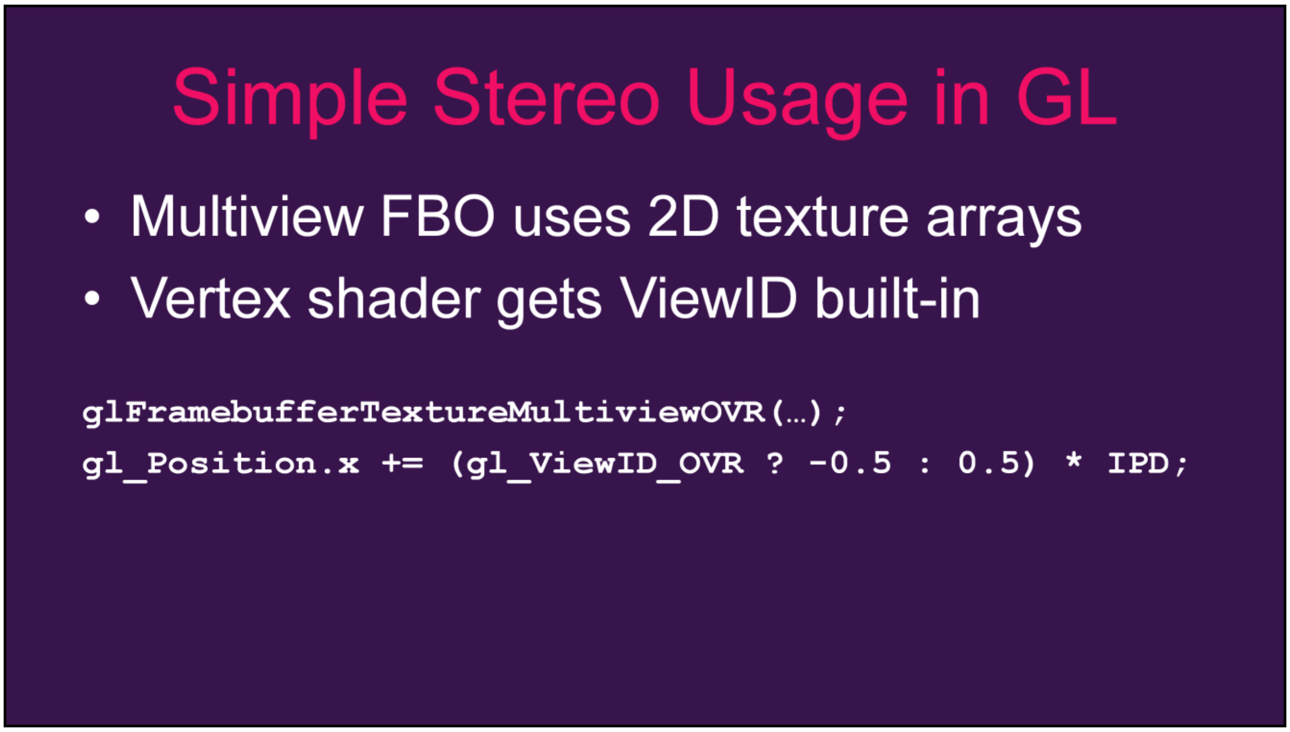
If we have made a command stream that can be used for both eyes, the next obvious thing would be to issue it once, and have the driver render to both eyes.
And that’s what the multiview extensions are about.
Draw once, and render the same command stream with N different position calculations to N different images.
Driver overhead and validation is approximately the same cost as monoscopic.



Stereo Redundancy



Worth looking at this image again and considering the situation for tiling GPUs.
If we divided these images into tiles, how close would each tile come to having the same geometry?
There's clearly room to simplify the tiling calculations – though I don't want to imply that this would be easy work for any particular tiling implementation.
And even with there being some extra clipped or culled geometry per bin, it'd likely be a win over binning twice.
If corresponding tiles were rendered at the same time, there'd also be a likely reduction in system memory bandwidth usage, because of texture cache sharing.

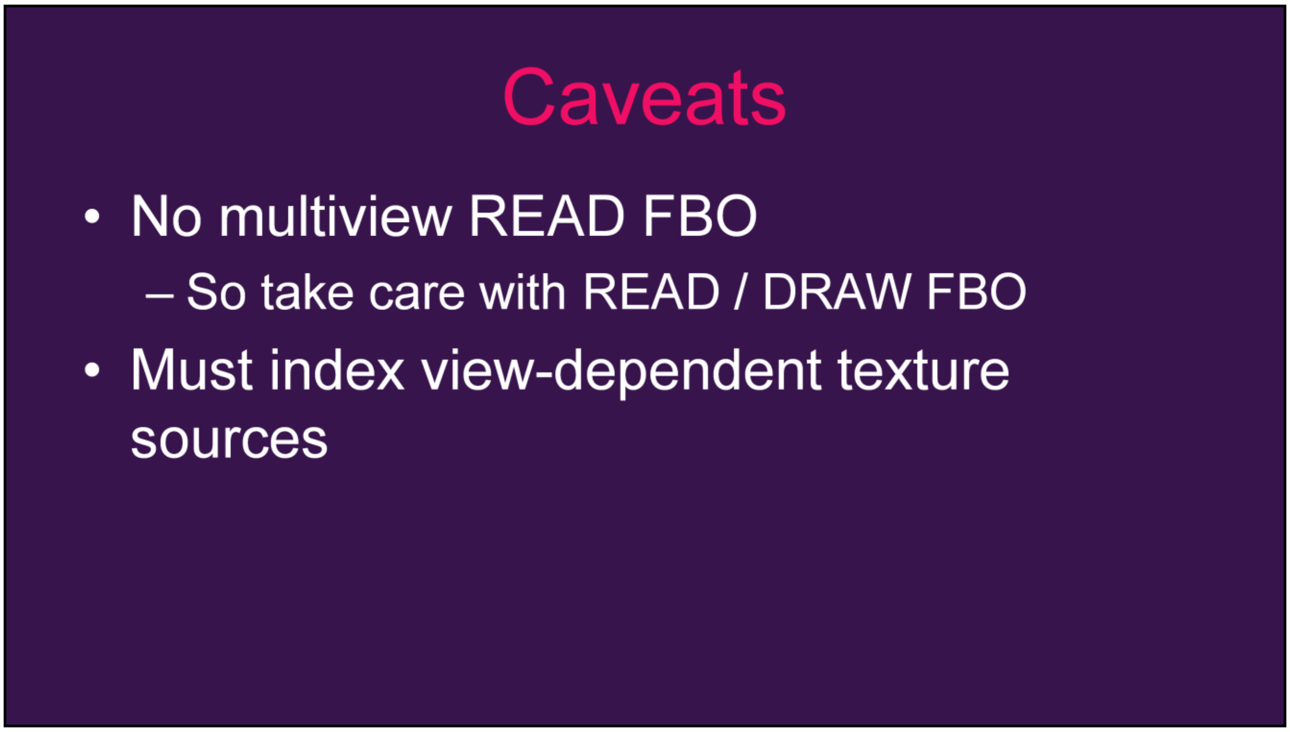


Simple Stereo Usage in GL

- Multiview FBO uses 2D texture arrays
- Vertex shader gets ViewID built-in

```
glFramebufferTextureMultiviewOVR(…);
gl_Position.x += (gl_ViewID_OVR ? -0.5 : 0.5) * IPD;
```

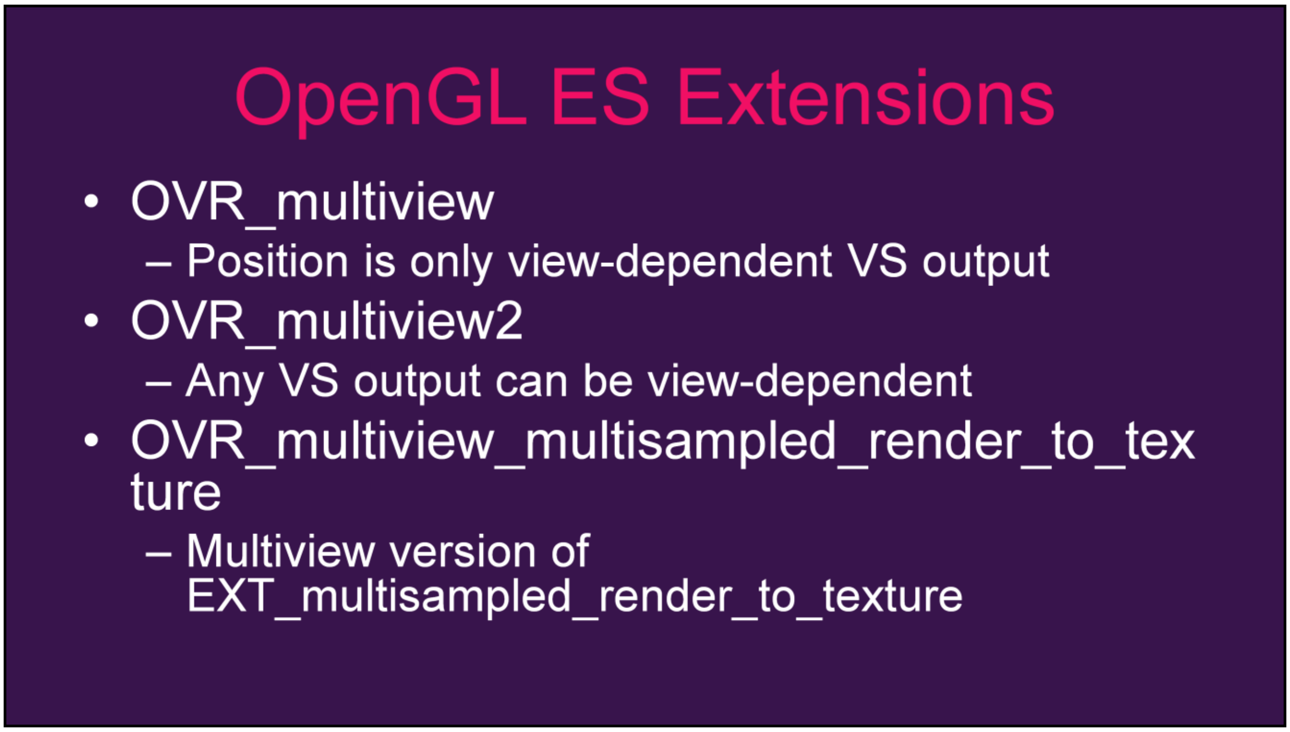
Structurally, the way this works is a bit like instancing. You create an FBO that uses a texture array for color and depth attachments.
Elements of the array correspond to separate views.
The vertex shader has a ViewID variable, which can be used to adjust the position of the output vertex.
In this very simple example, we're just applying a plus or minus bias to the output x coordinate of half the IPD.
More general and potentially more complex usage would be to index into different view and projection matrices based on ViewID.



Caveats

- No multiview READ FBO
  - So take care with READ / DRAW FBO
- Must index view-dependent texture sources

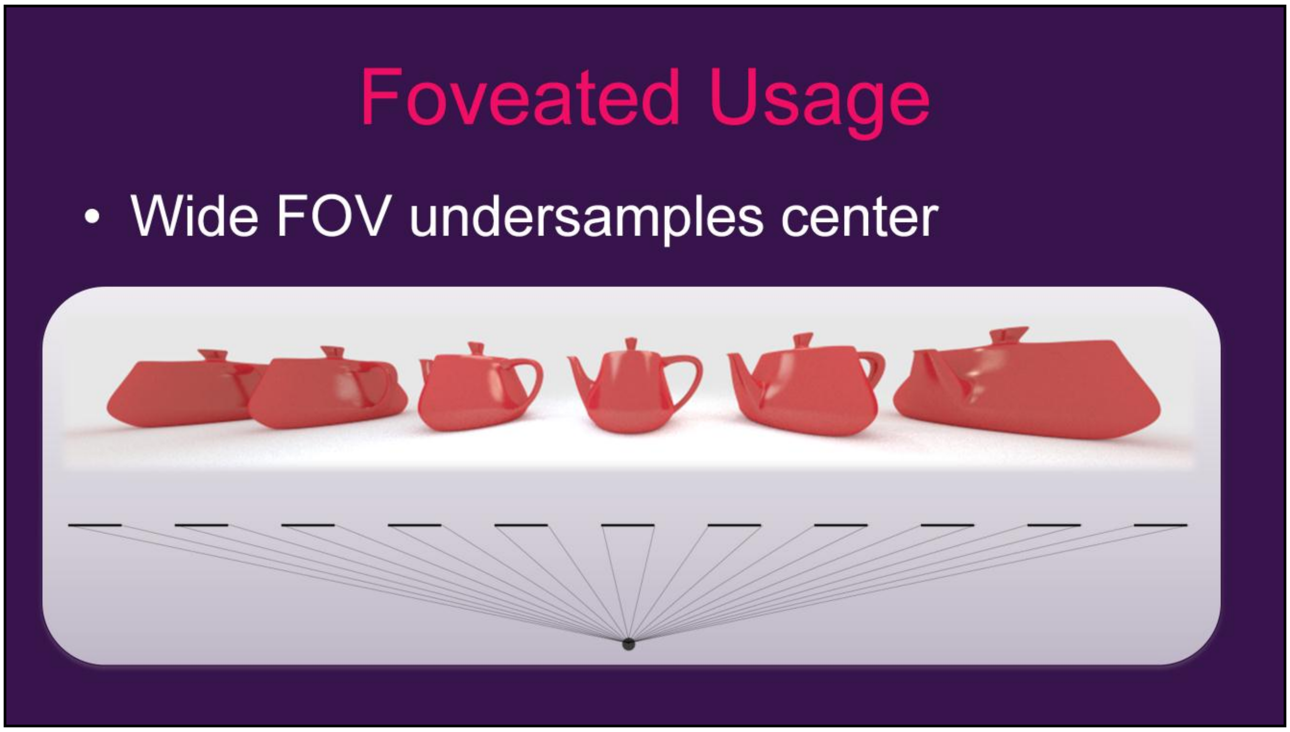
Maybe the least obvious thing about multiview in GL is that glReadPixels and friends are not defined for multiple simultaneous views, so to keep things simple, we just disallowed using multiview FBOs as a READ target.
If you have view-dependent textures, they need to be indexed as well. Mostly textures are view-independent, but it does come up in VR contexts like stereo movies being displayed in a 3D scene…



OpenGL ES Extensions

- OVR_multiview
  - Position is only view-dependent VS output
- OVR_multiview2
  - Any VS output can be view-dependent
- OVR_multiview_multisampled_render_to_texture
  - Multiview version of EXT_multisampled_render_to_texture

Hopefully the idea of multiview makes sense at a conceptual level now, and roughly how it integrates with the API.
But how does it look in terms of API extensions?
The first and simplest of the extensions is OVR_multiview. In it, you can have different vertex positions per view, but no other output from the vertex shader can depend on the view.
The early implementors of this extension also could support any vertex shader output being view dependent at no additional cost, so why not do that too?!
In fact, there was a nominal extra cost to just check that no other attributes varied with position.
One of those unfortunate cases where doing less actually costs more. It happens in APIs.
In mobile graphics, you can often get multisample nearly for free if your multisample buffer lives only in tile memory, and is resolved to an aliased buffer when it needs to be saved to system memory.
Doing things this way requires the use of an extension in OpenGL. Making it work with multiview rendering requires an additional extension.

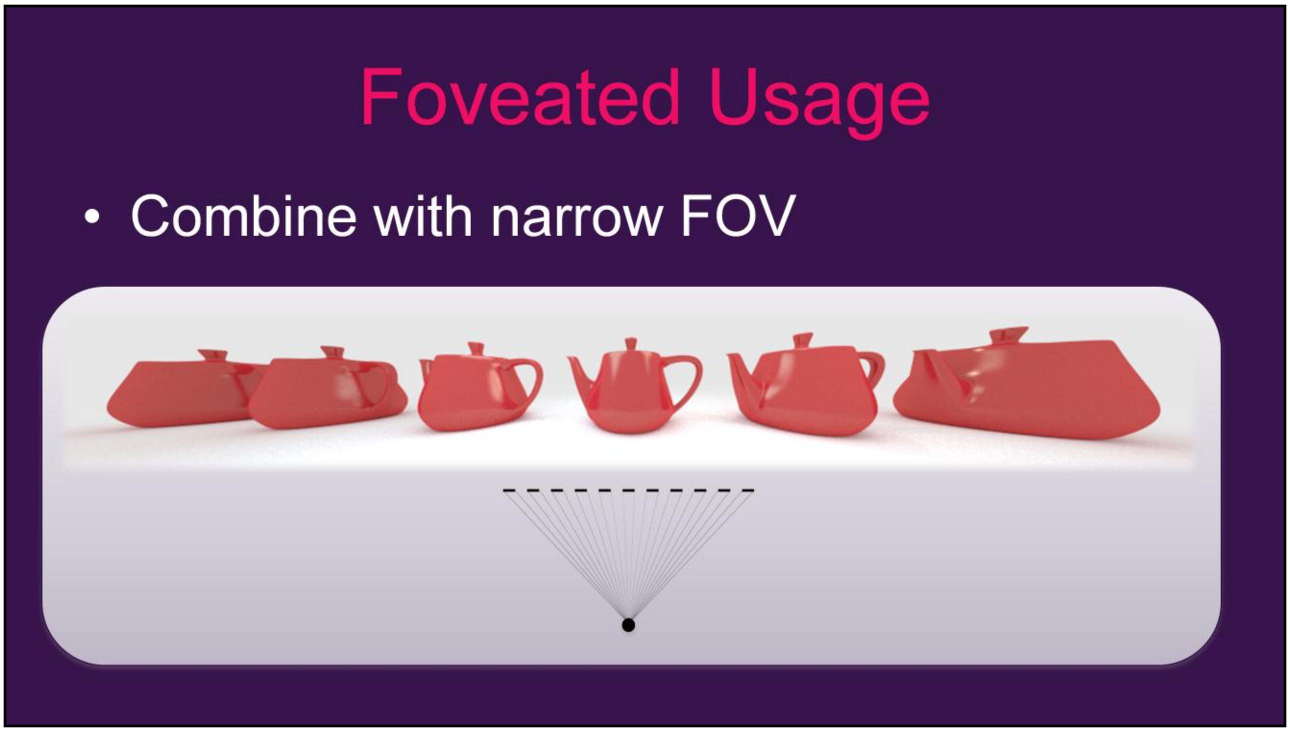


Foveated Usage

- Wide FOV undersamples center

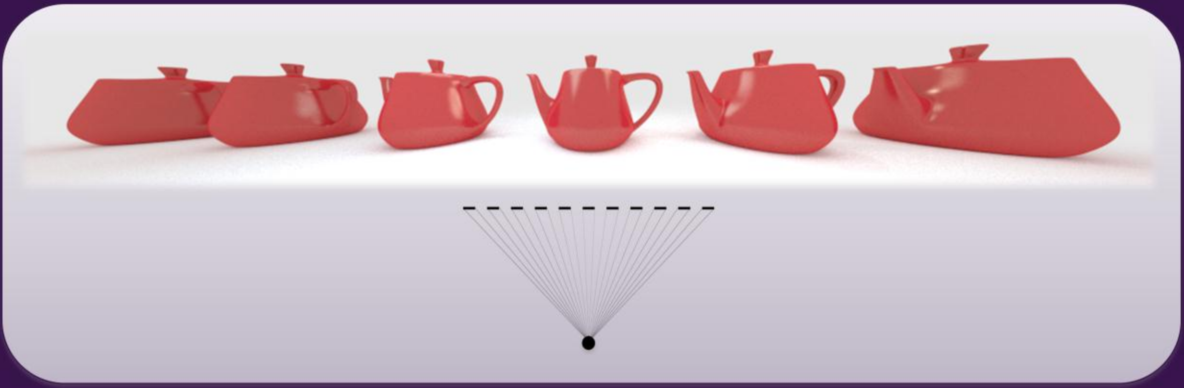
The way planar projection rendering works, you can get a 180 degree field of view with an infinite number of samples.
If you imagine standing in front of a picket fence, where each picket represents a pixel, this will give you an idea of how much angular distance each pixel covers.
The pickets directly in front of you cover the widest angles in your view. The ones off toward infinity to the left and right approach infinitely thin.
This is a terrible situation, because you spend tons of pixels in your periphery to represent a relatively small portion of the field of view.
And the region with the lowest – worst – sample density is directly in front of you.

Naïve planar projection is a terrible way to get really wide fields of view. And it breaks down completely if you need to go wider than 180 degrees anyway.

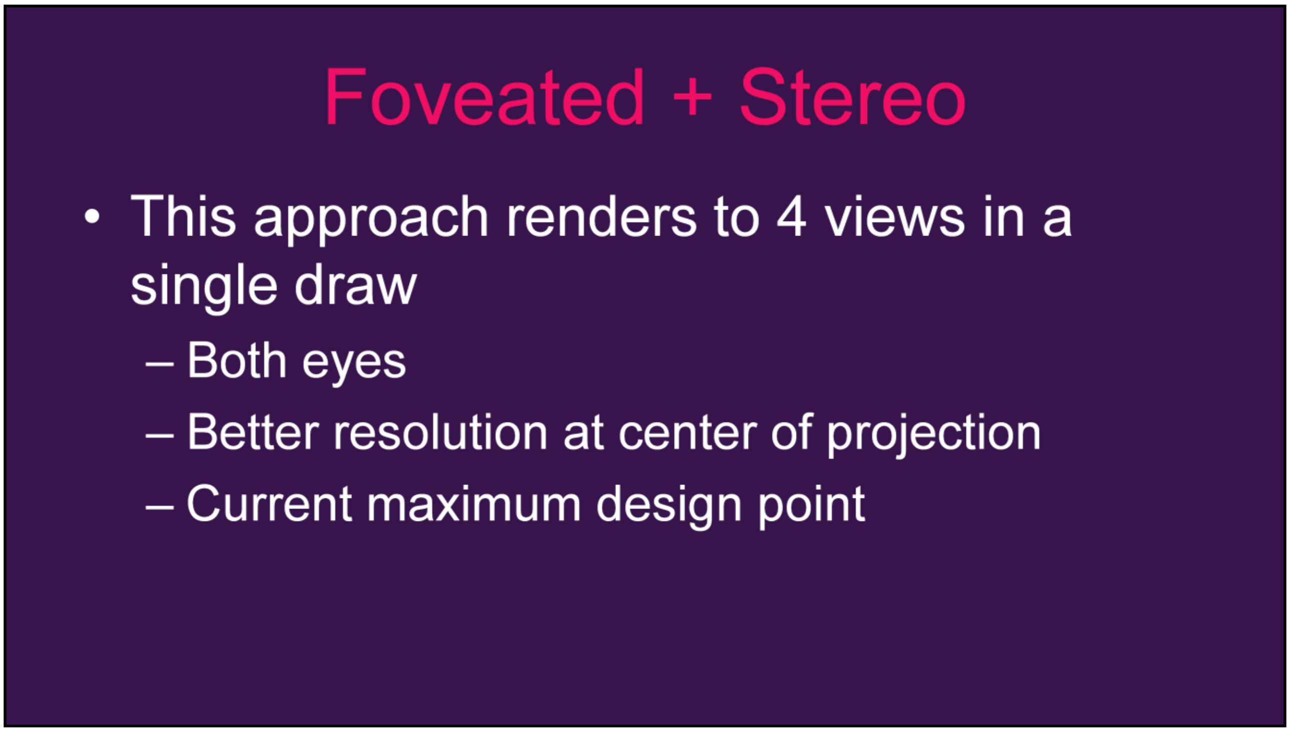


Foveated Usage

- Combine with narrow FOV



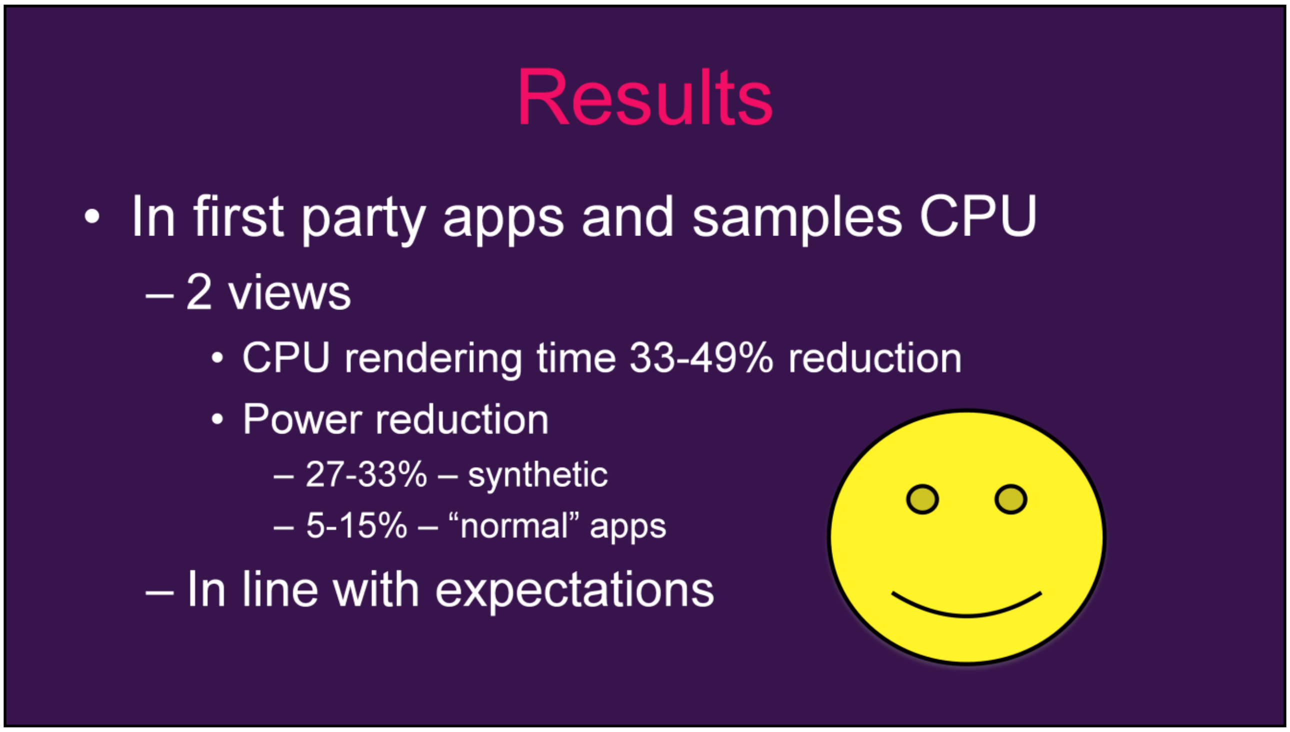
The human visual system has its highest sample density directly in front of you. In VR people can mean significantly more complicated and sophisticated things when they say “foveated rendering”.
In this context, we really just mean a simple multi-resolution rendering where one view is modest (say 60 degree) fov inset of a wider (say 120 degree) fov.
The way that multiview rendering works, they both have to be the same resolution images because they’re elements of a texture array.
But because the fov is different, you get reasonable sample distribution in the narrow fov view in the region where the sample distribution is worst in the wide fov view.
Yes, “fov view” is a bit like “atm machine”. Sorry.



Foveated + Stereo

- This approach renders to 4 views in a single draw
  - Both eyes
  - Better resolution at center of projection
  - Current maximum design point

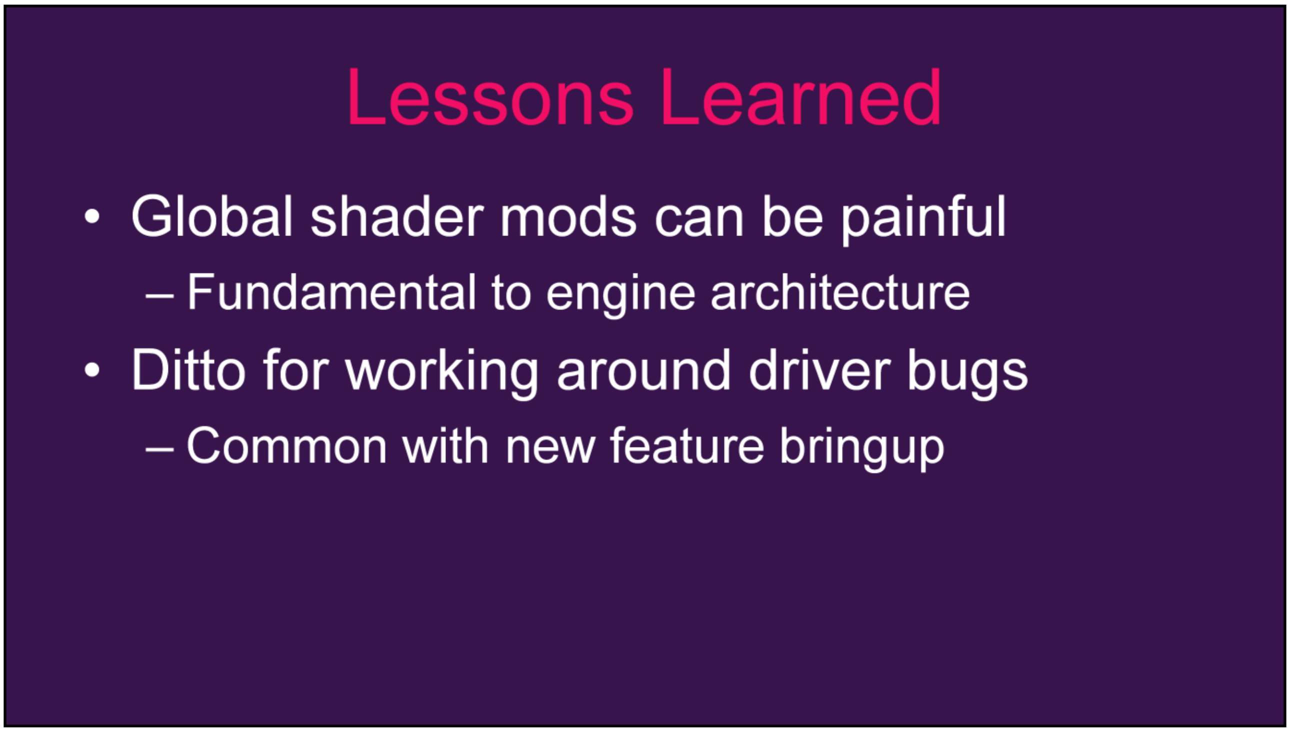
In practice we would expect to do stereo if we're using only two views. But if the implementation supports 4, we could do both foveated and stereo in a single pass.
It's worth noting that we don't have significant experience with foveated rendering, but we do expect it to be a natural follow-on to single-pass stereo, and particularly relevant when fov's begin to push past 100 degrees.
There may be other, better solutions to this problem by then, but this one naturally falls out of the need for efficient stereo, and may well be a natural evolutionary path.



Results

- In first party apps and samples CPU
  - 2 views
    - CPU rendering time 33-49% reduction
    - Power reduction
      - 27-33% – synthetic
      - 5-15% – “normal” apps
  - In line with expectations

When you embark on an extension like this, it can take a long time from idea to implementation.
This is especially true in mobile development where software release pipelines can be very deep.
It was gratifying when we did finally get devices with multiview support that real and synthetic apps showed cpu time reductions in the 33-49% range.
And because rendering and driver overhead isn’t the only thing that the cpu does, and because we still have the bulk of the GPU work to do, power numbers dropped 27-33%.
These are still very respectable numbers for doing the same thing though!
And I’m glad to say the numbers were in line with what we expected to get.
Architectures designed to increase GPU efficiency with multiview might get even better results, but multiple views is unlikely to ever be “free” (relative to the cost of a single view).
We’ll take these gains though.

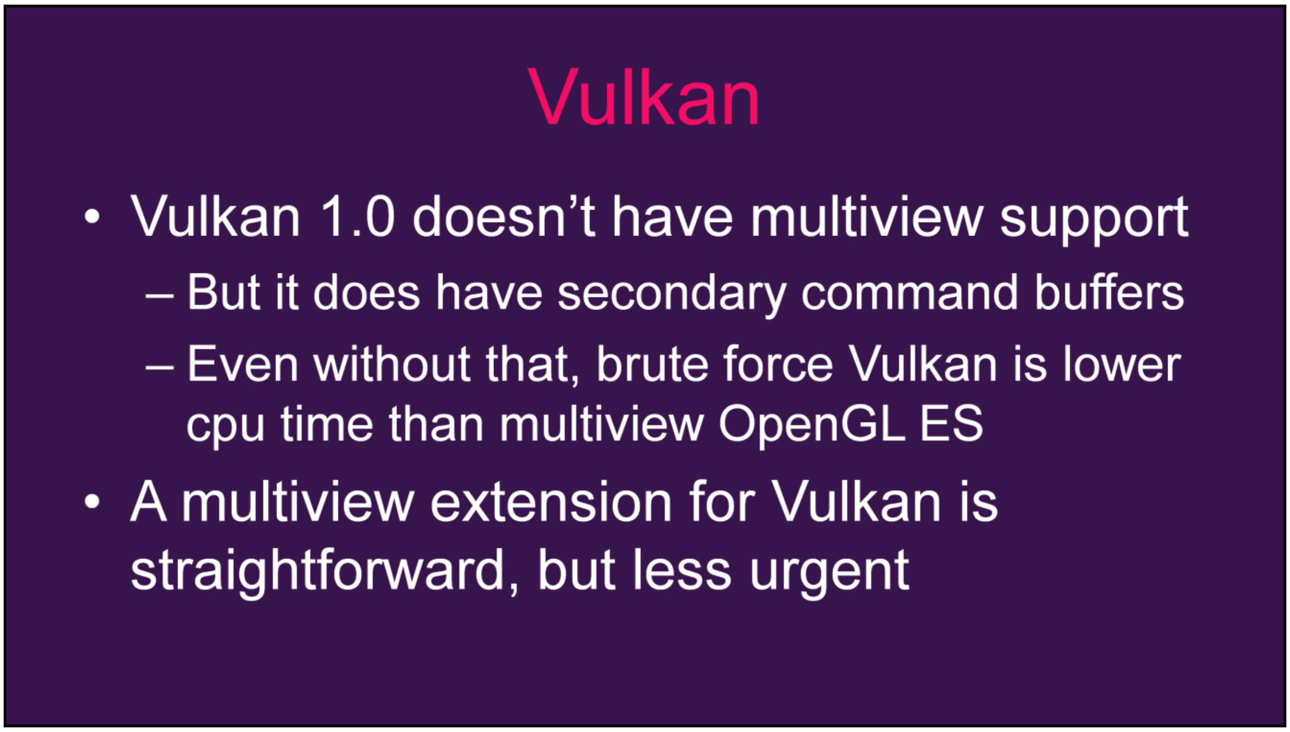


Lessons Learned

- Global shader mods can be painful
  - Fundamental to engine architecture
- Ditto for working around driver bugs
  - Common with new feature bringup

During the formulation of this extension, we considered a number of options. Have multiple bound FBOs instead of arrays, for example.
That wound up being complicated because different FBOs could have radically different configurations, and direct hardware support for multiple views was easier when using array targets.
As there was already a variation on that kind of support with geometry shader (and even vertex shader) support for specifying which layer of a 2D array a primitive should go to.
Choosing arrays had its complications too though, as so much of our software expected FBOs with simple non-array targets.
And things like multisampled render-to-texture needed an additional extension.

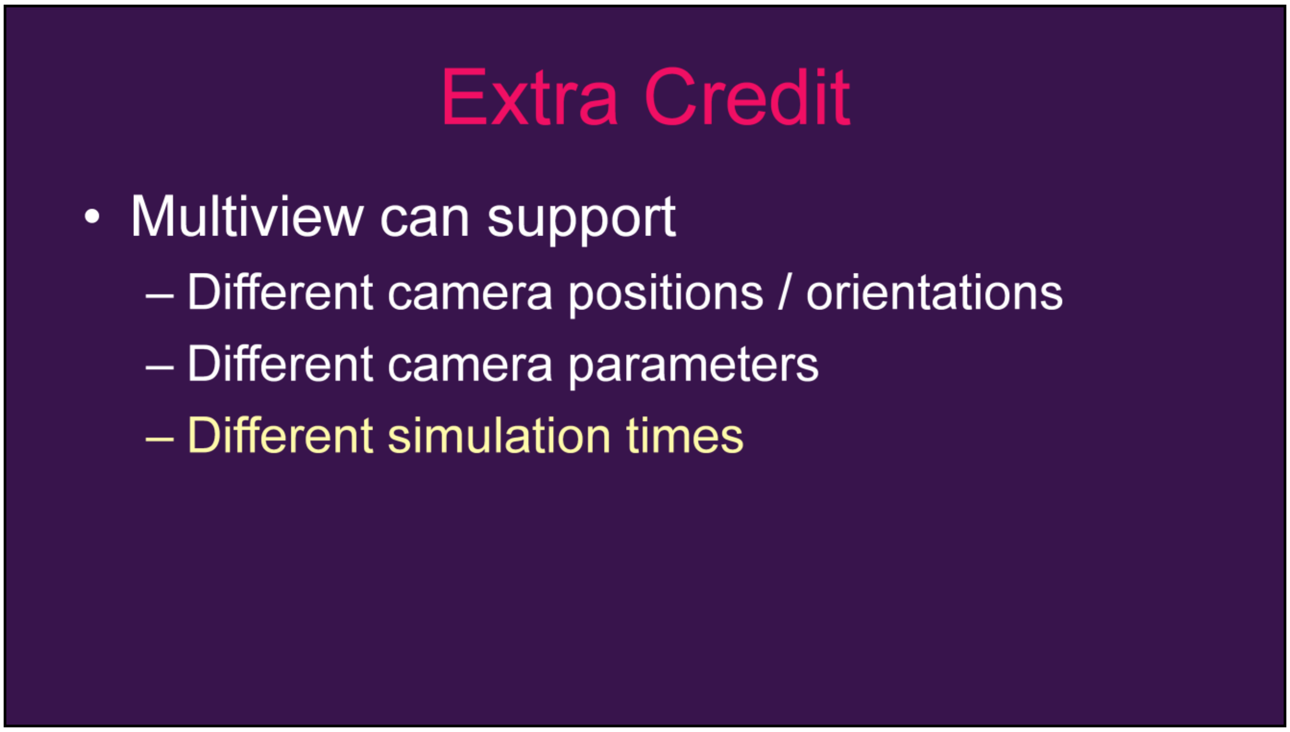
We also considered having the multiview support be fixed-function. This would have offered significant simplification for porting, as the model we chose requires every single vertex shader to be rewritten.
Even though those modifications were very well-defined, touching every shader is a bigger effort. Not least because we discovered some driver bugs in that process that stalled progress until we could find a workaround.
But these things happen with feature bringup, and we think the path is pretty stable now.



Vulkan

- Vulkan 1.0 doesn't have multiview support
  - But it does have secondary command buffers
  - Even without that, brute force Vulkan is lower cpu time than multiview OpenGL ES
- A multiview extension for Vulkan is straightforward, but less urgent

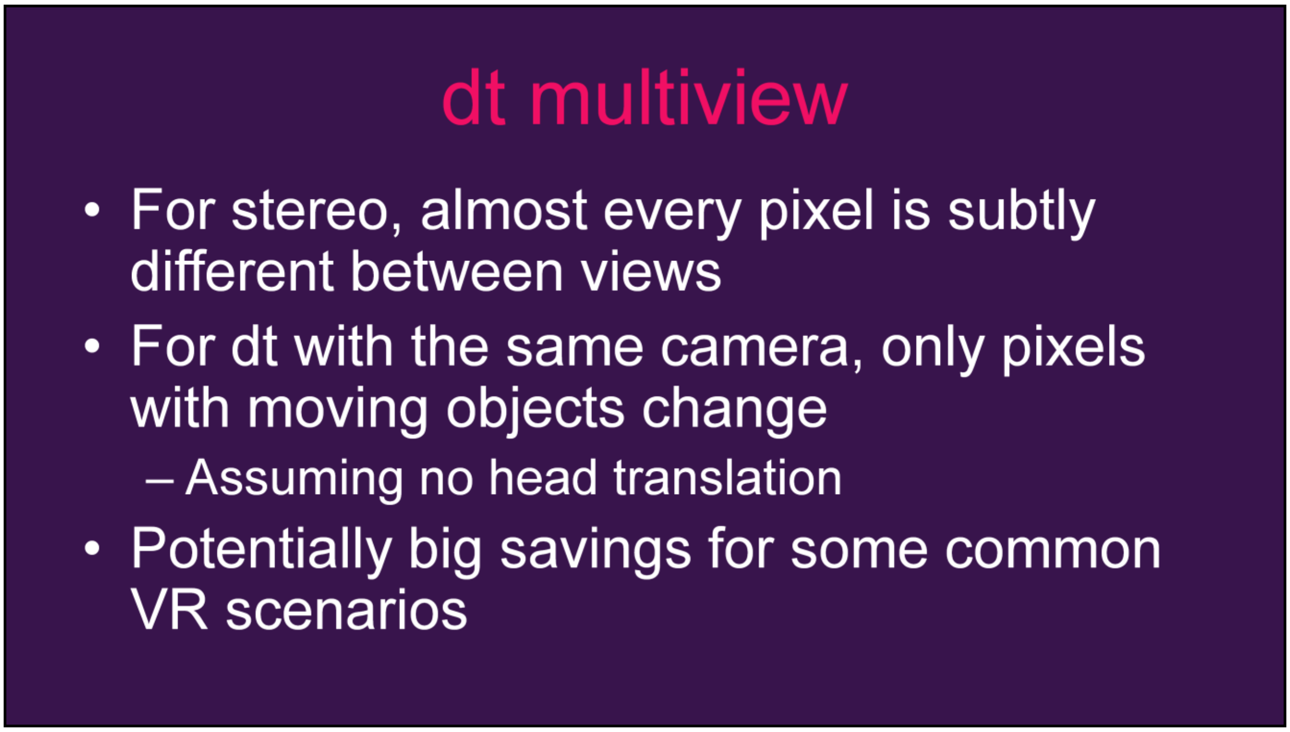
We went through the process of scoping out what multiview support in Vulkan would look like. It wouldn't be too hard to support.
But Vulkan is already so very low overhead.
Even brute force Vulkan beats the best case OpenGL ES multiview.
So while there's room for improvement – in particular being able to improve GPU efficiency – it's not the highest priority issue right now for Vulkan, and it's definitely not what's blocking VR on Vulkan.



Extra Credit

- Multiview can support
  - Different camera positions / orientations
  - Different camera parameters
  - Different simulation times

It’s interesting to consider that we first considered using multiview to make stereo rendering more efficient.
Then we realized, that with this generality, we could also do a form of foveated rendering to accommodate wider fields of view.
Because there’s enough generality in the design, it can also accommodate cameras that are at different points in time.
That may not sound incredibly exciting, but it may actually be easier to enable than foveated rendering.
And for nearly the same app and driver overhead as 30 Hz mono rendering, you might be able to achieve 60 Hz stereo.



dt multiview

- For stereo, almost every pixel is subtly different between views
- For dt with the same camera, only pixels with moving objects change
  - Assuming no head translation
- Potentially big savings for some common VR scenarios

Considering how we might optimize for the dt multiview is interesting.
For a lot of common apps, it's comfortable to have stationary head position.
And for successive frames, in that context, if you don't change the eye orientation, pixels in frame N and frame N+1 are identical if there are no moving objects in them.
For a tile-based renderer, this could be as simple as detecting in the tiling phase that no time-dependent geometry is in the tile.
And when that happens, you can skip rendering the N+1 version. There are common scenarios in VR where this kind of optimization might be a significant savings.
For example, it might make 120 Hz rendering plausible, even on low-end devices.

So there's still some interesting headroom to explore significant efficiency optimizations in the traditional rendering framework for VR. So go do that! ;-)

The End

Thanks!

Feel free to follow up at

cass.everitt@oculus.com