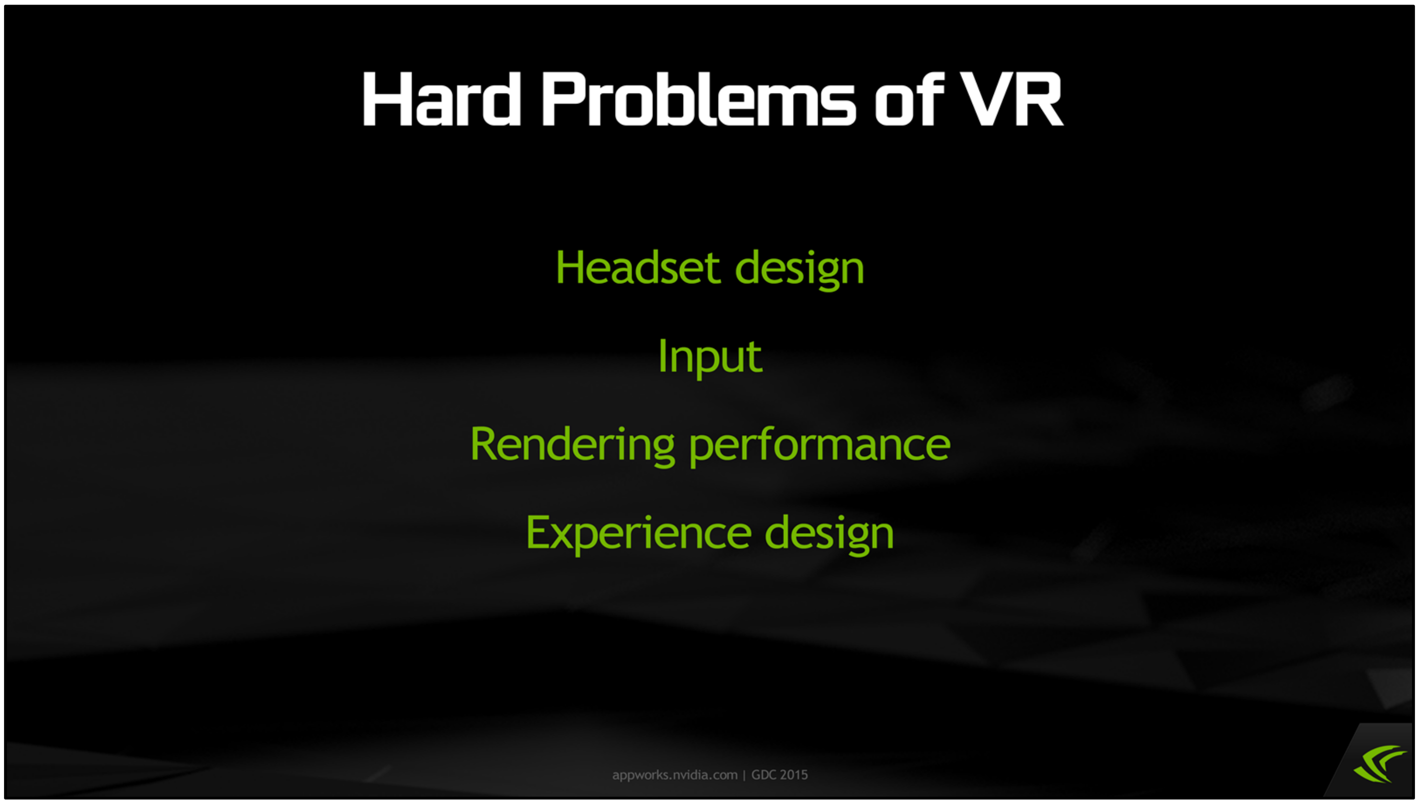
NVIDIA GEFORCE GTX

VR Direct: How NVIDIA Technology Is Improving the VR Experience

Nathan Reed — Developer Technology Engineer, NVIDIA
Dean Beeler — Software Engineer, Oculus

Who We Are

- Nathan Reed
  - NVIDIA DevTech — 2 yrs
  - Previously: game graphics programmer at Sucker Punch
- Dean Beeler
  - Oculus — 2 yrs
  - Previously: emulation, drivers, mobile dev, kernel



Hard Problems of VR

Headset design

Input

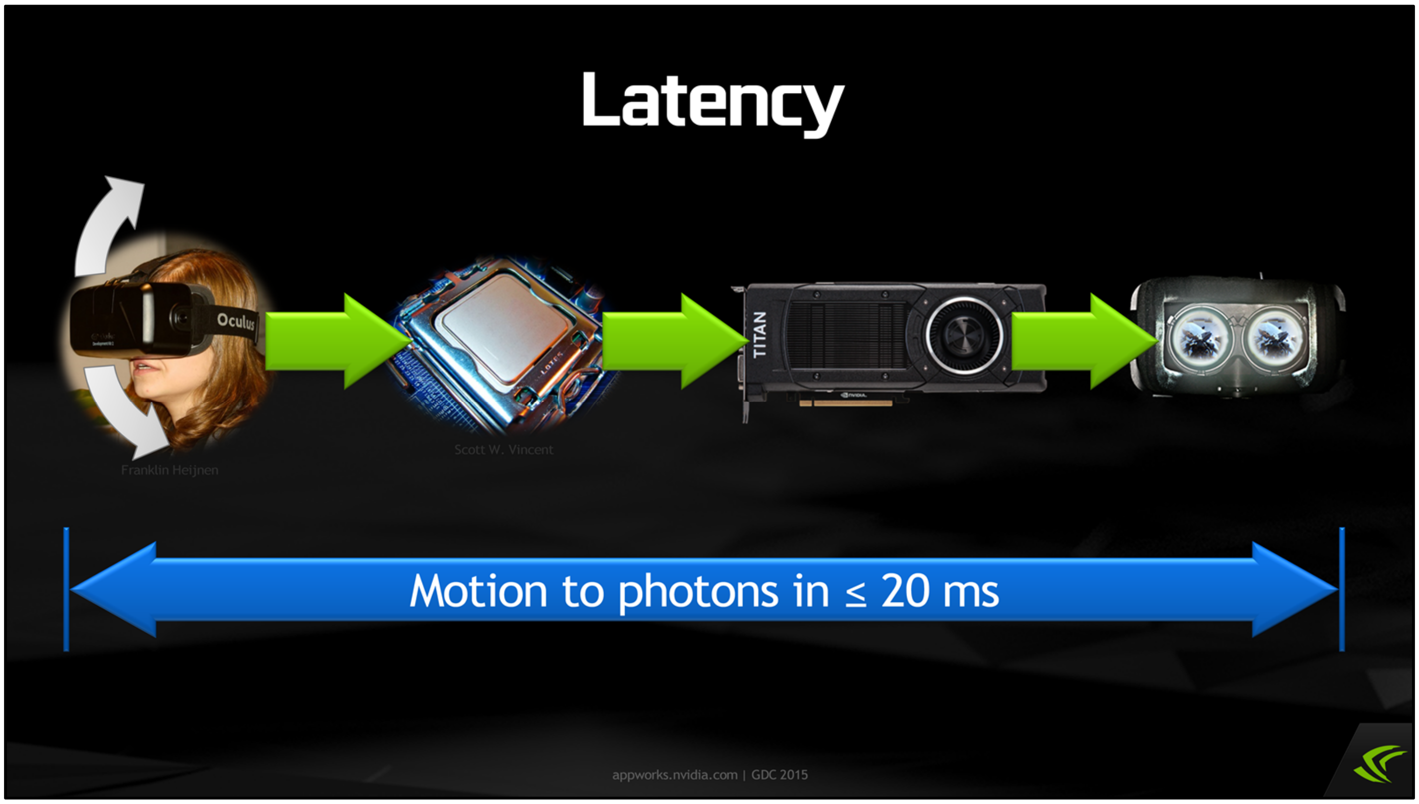
Rendering performance

Experience design

appworks.nvidia.com | GDC 2015

There are lots of hard problems to solve in VR — in headset design, in input devices, in rendering performance, and in designing VR-friendly user experiences, just to name a few areas.

At NVIDIA, being a GPU company, clearly rendering performance is the area we're going to concentrate on, as that's where we can help the most.



Latency

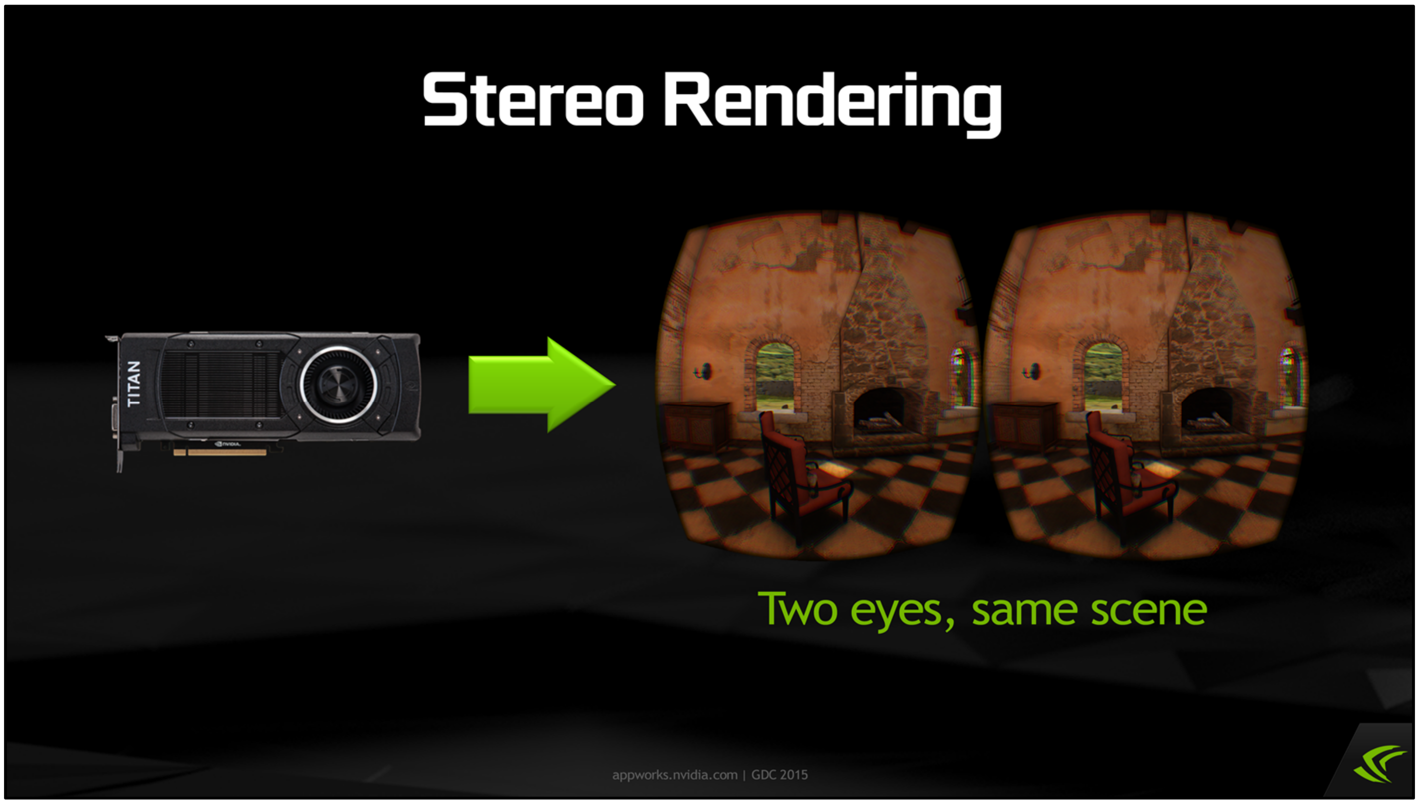
Motion to photons in ≤ 20 ms

Virtual reality is extremely demanding with respect to rendering performance. Not only do we need to hit a very high framerate — 90 Hz for Oculus' Crescent Bay — but we need to do it while maintaining low latency.

Latency is the amount of time between an input event — in this case, when you move your head — and when the result of that input shows up on the display. Mike Abrash at Oculus has done some research on this, and his results show that the motion-to-photons latency should be at most 20 milliseconds to ensure that the experience is comfortable for players.

The problem is that we have a long pipeline, where input has to first be processed and a new frame submitted from the CPU, then the image has to be rendered by the GPU, and finally scanned out to the display. Each of these steps adds latency.

Traditional real-time rendering pipelines have not been optimized to minimize latency, so this goal requires us to change how we think about rendering to some extent.

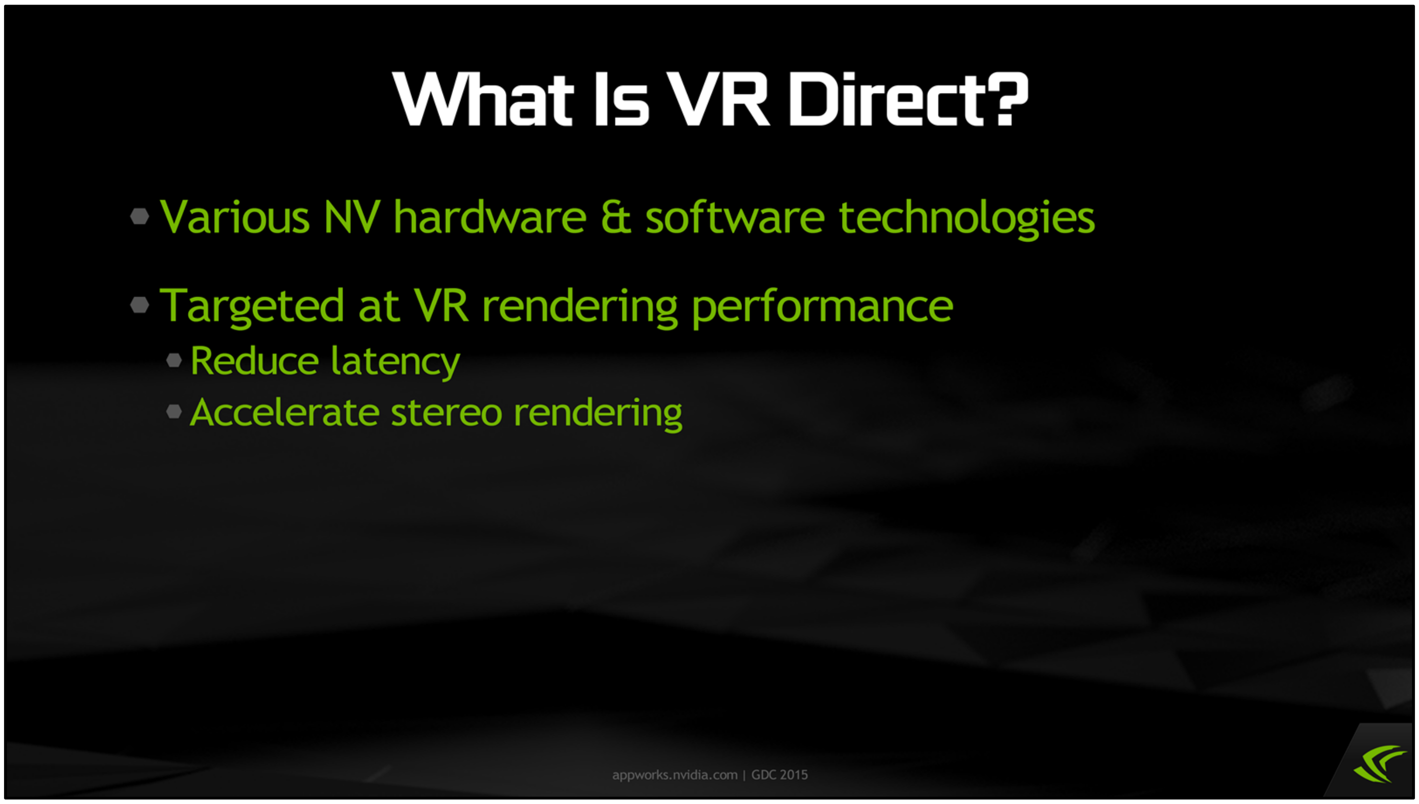


Stereo Rendering

TITAN
Two eyes, same scene
appworks.nvidia.com | GDC 2015

The other thing that makes VR rendering performance a challenge is that we have to render the scene twice, now, to achieve the stereo eye views that give VR worlds a sense of depth. Today, this tends to approximately double the amount of work that has to be done to render each frame, both on the CPU and the GPU, and that clearly comes with a steep performance cost.

However, the key fact about stereo rendering is that both eyes are looking at the same scene. They see essentially the same objects, from almost the same viewpoint. And we ought to be able to find ways to exploit that commonality to reduce the rendering cost of generating these stereo views.

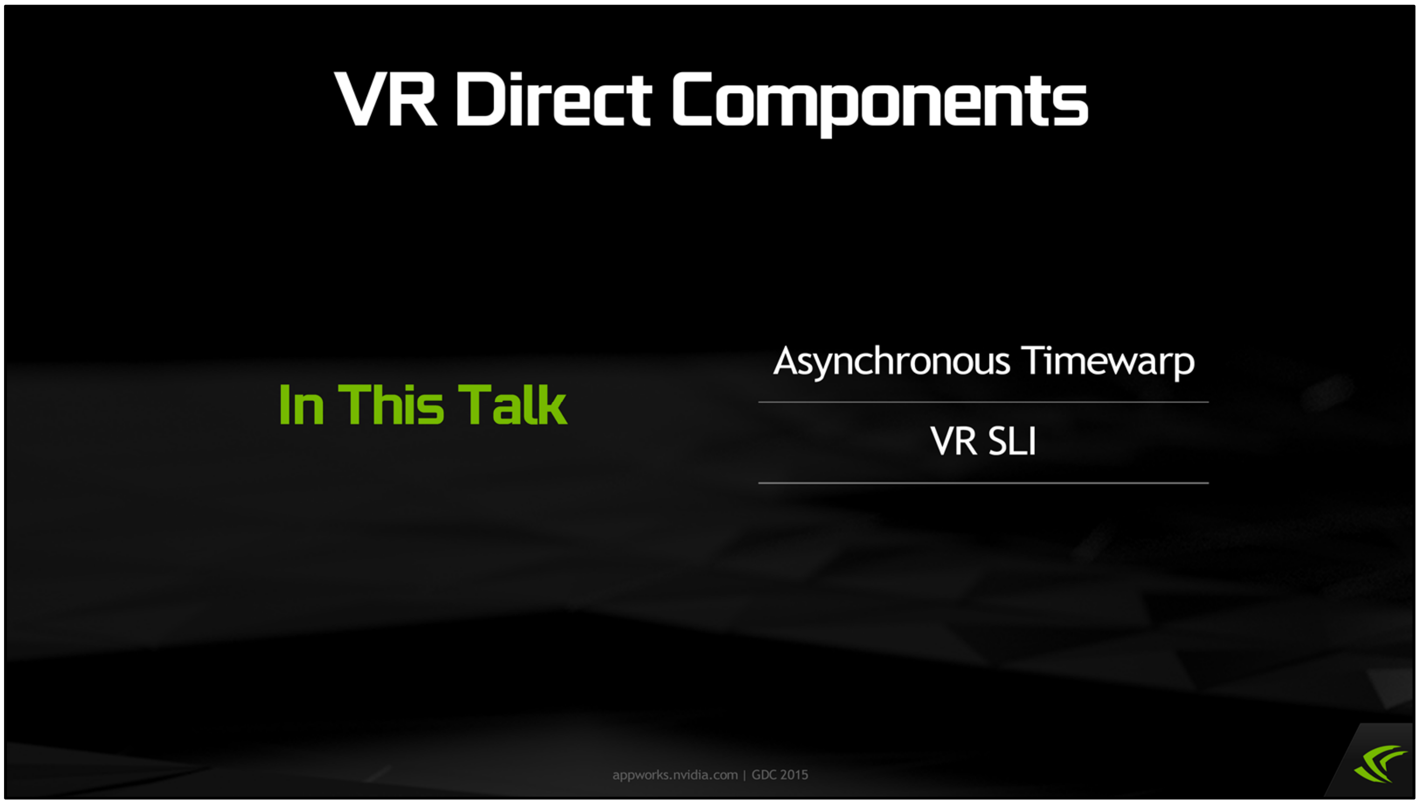


What Is VR Direct?

- Various NV hardware & software technologies
- Targeted at VR rendering performance
  - Reduce latency
  - Accelerate stereo rendering

And that's where VR Direct comes in. VR Direct refers to a variety of hardware and software technologies NVIDIA is working on to attack those two problems I just mentioned — to reduce latency, and accelerate stereo rendering performance, ultimately to improve the VR experience.

We have a LOT of people at NVIDIA interested in VR, and a lot of different technologies being built. Some of these are closer to production-ready things that developers are already using, and others are a little more researchy and farther away.



VR Direct Components

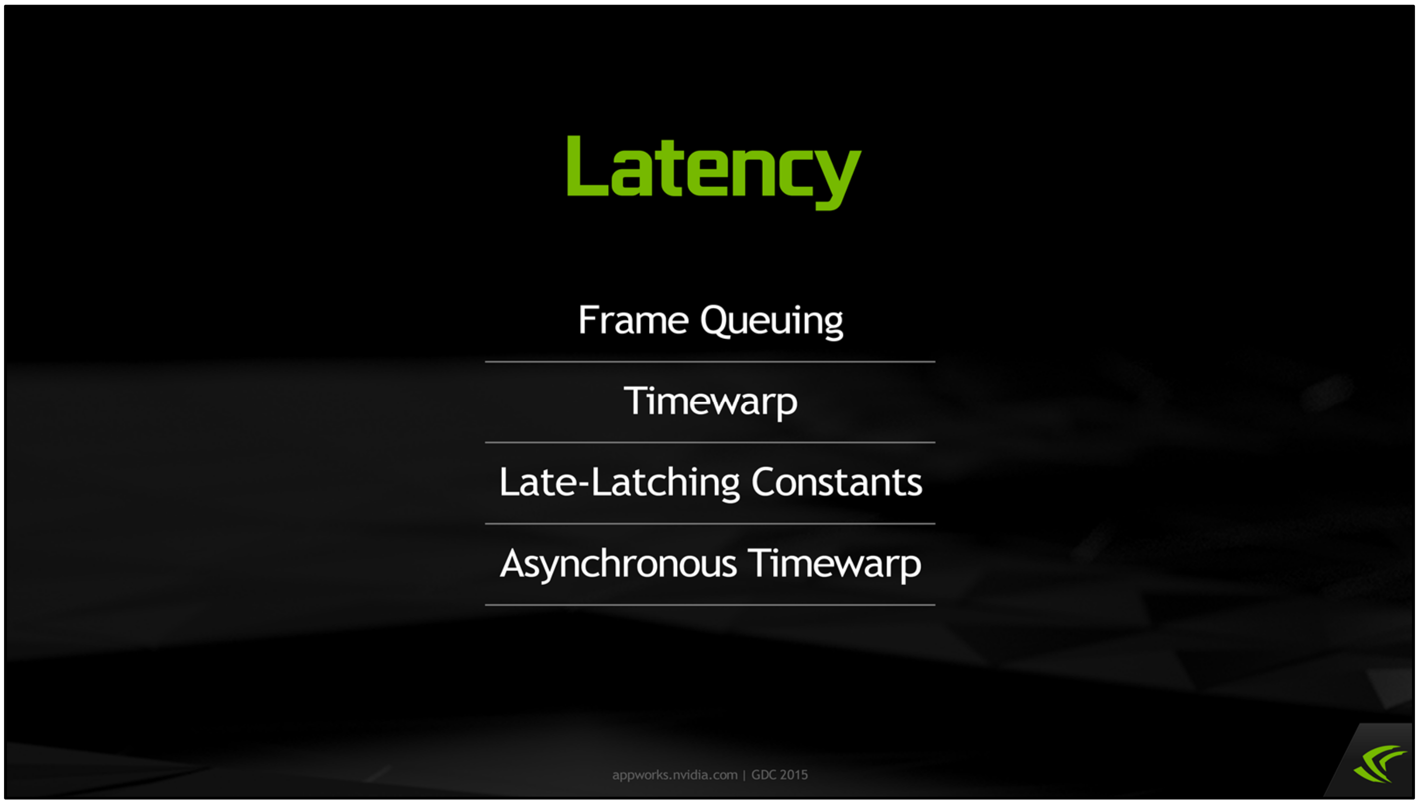
In This Talk

- Asynchronous Timewarp
- VR SLI

appworks.nvidia.com | GDC 2015

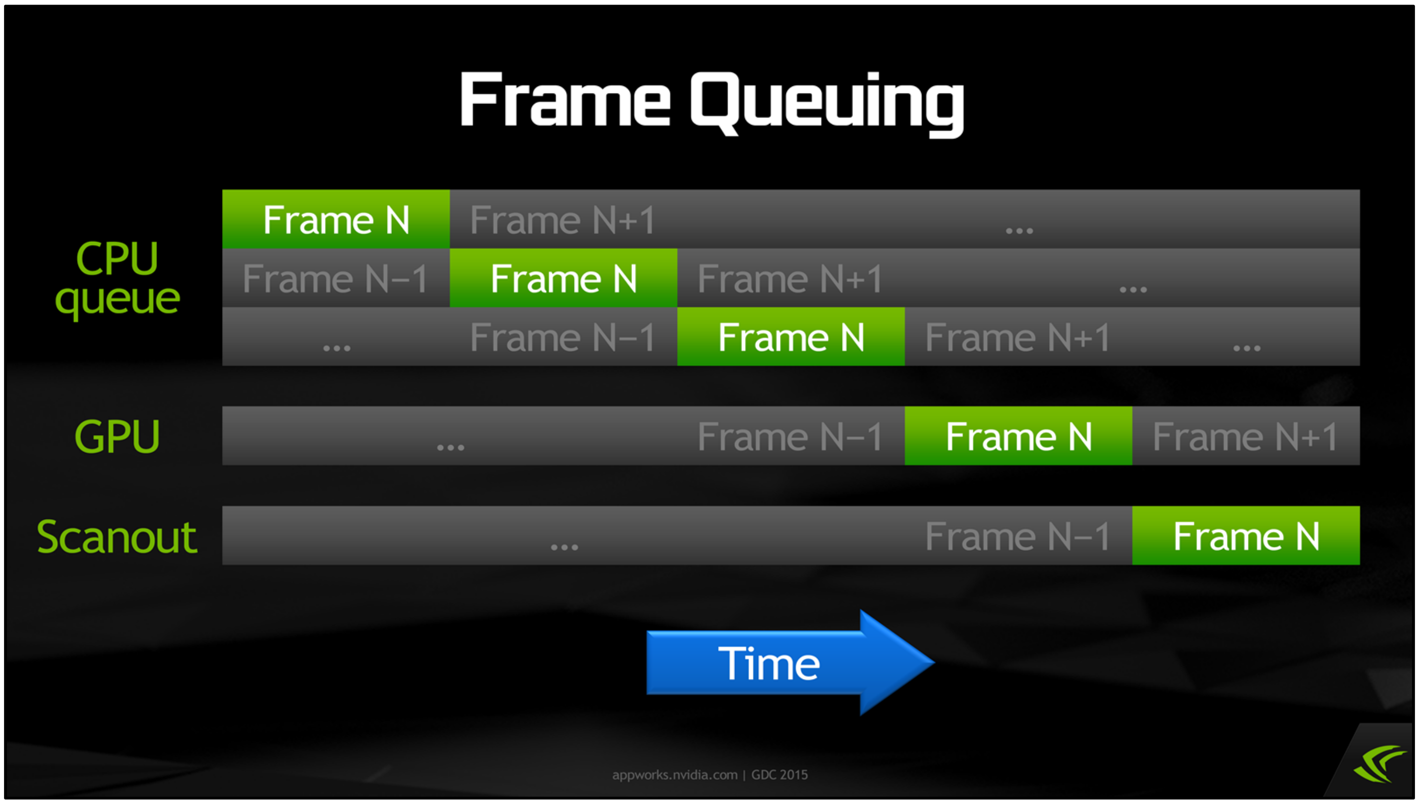
Specifically, today we're mainly going to be discussing two components of VR Direct, which are: asynchronous timewarp and VR SLI. These are technologies that are pretty close to production-ready, with early implementations already in the hands of a few developers. Each of these technologies is focusing on a different aspect of the VR rendering performance problem.

There are other components we've announced previously and more may be announced in the future, but we're not talking about those today.



Latency

- Frame Queuing
- Timewarp
- Late-Latching Constants
- Asynchronous Timewarp

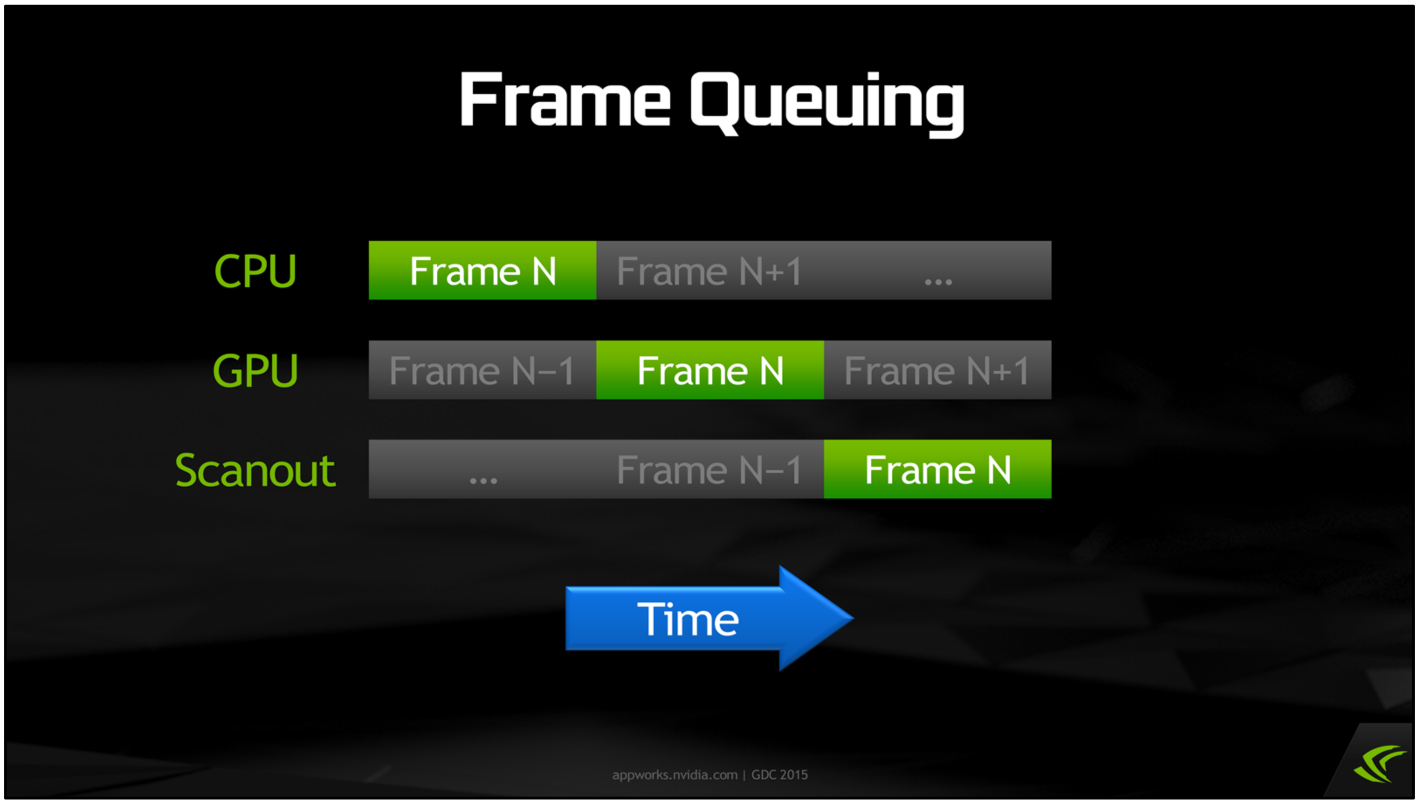


Frame Queuing

Traditionally, in real-time graphics we allow the CPU to run ahead of the GPU. The OS can queue up several frames' worth of rendering commands. This is done to obtain the maximum possible framerate: it ensures the CPU and GPU run in parallel and do not need to wait for each other unnecessarily. It also provides a cushion so that frames can keep being delivered at a steady rate even if there's a momentary hitch in CPU performance.

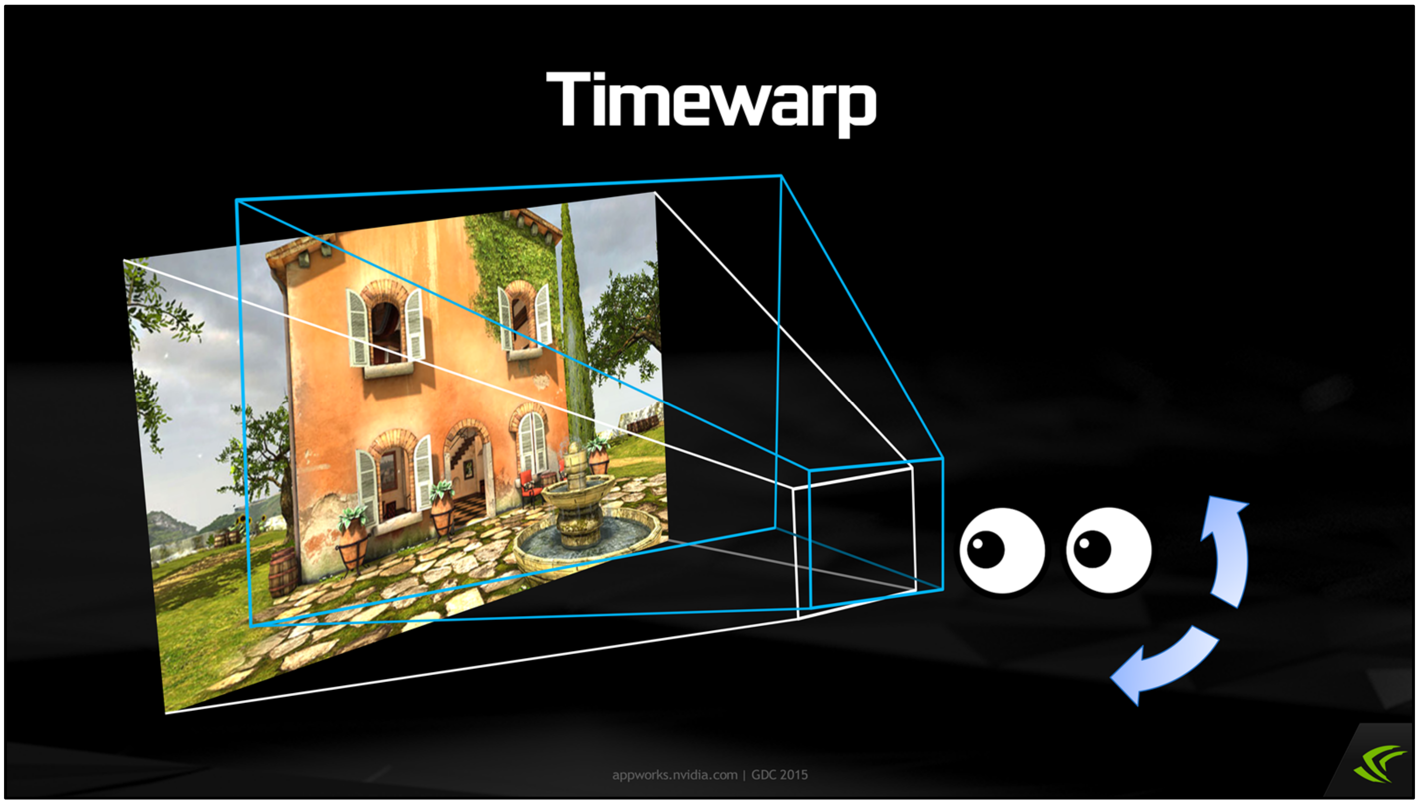
With 2D displays, the latency that accumulates in frame queueing is generally considered acceptable for many types of apps.

However, at 90 Hz, each additional queued frame adds 11.11 milliseconds of latency, and that is very perceptible by the viewer in VR. With several queued frames, we're blowing right past our 20 ms target.



Frame Queuing

There are number of ways to limit the size of the frame queue; for example, more recent versions of DXGI expose SetMaximumFrameLatency. However, even with just one queued frame we still have too much latency for a comfortable VR experience.



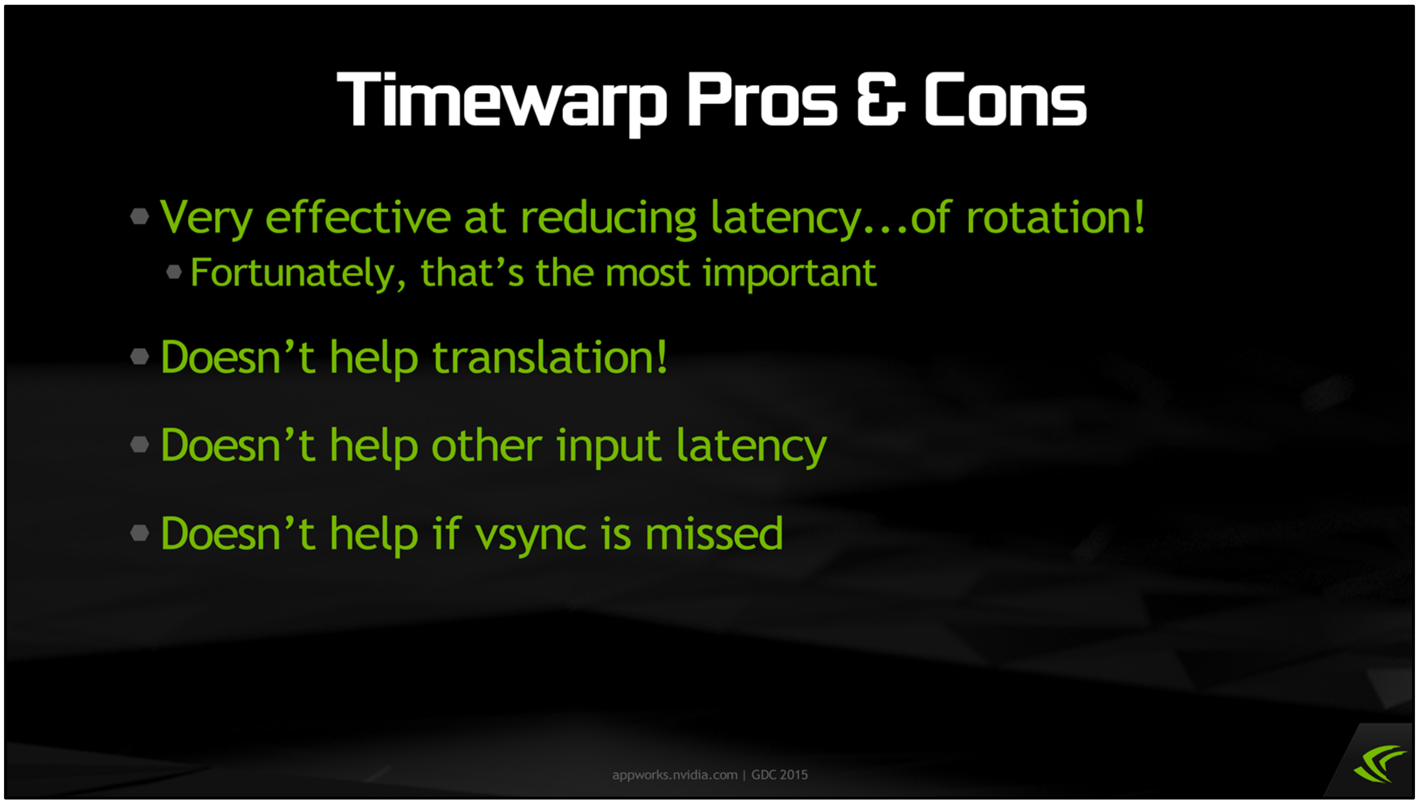
Timewarp

That's where timewarp comes in. Timewarp is a feature implemented in the Oculus SDK that allows us to render an image and then perform a postprocess on that rendered image to adjust it for changes in head motion during rendering. This reduces latency further and opens doors for modifications in the render pipeline from the app's perspective.

This is a purely image-space operation, and in the Oculus SDK it's incorporated into the same rendering pass that performs lens distortion. The pass is very quick — it takes less than half a millisecond on most GPUs. So we can put it right at the end of the frame, rendering it at the last possible moment before we start to scan out the frame to the display.

It's important to understand the distinction between distortion and timewarp. Distortion is necessary to take what the app has generated and warp it to compensate for the lenses. Timewarp takes this a step further, and shifts the image to compensate for changes in the orientation of the viewer's head since the app began rendering.

This gets the apparent latency of head-tracking down to only a few milliseconds longer than scanout. This is great — we've reached our 20 ms target and even exceeded it a bit!



Timewarp Pros & Cons

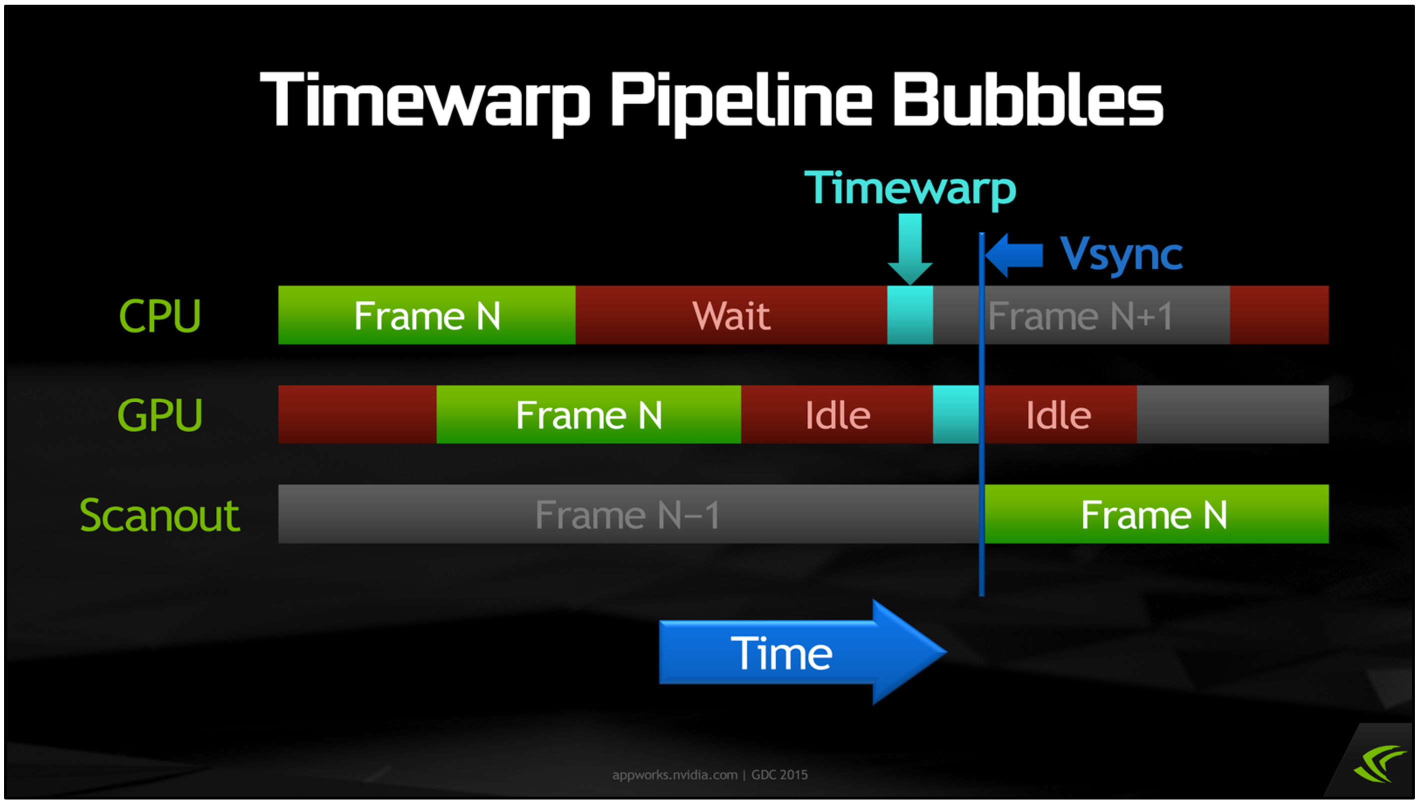
- Very effective at reducing latency…of rotation!
  - Fortunately, that's the most important
- Doesn't help translation!
- Doesn't help other input latency
- Doesn't help if vsync is missed

appworks.nvidia.com | GDC 2015

So we're done, right? Well, not so fast. Timewarp is very effective at reducing the apparent latency of head *rotation*. That's very important, as excessive latency in orientation is very uncomfortable. However, timewarp as it currently exists can't help you with head *translation*. It only has a 2D image to work with, so it can't move the viewpoint.

Timewarp also can't help you with other input latency. And it only works if your app is already rendering fast enough to hit vsync rate on the headset — if you miss vsync, you'll have a stuck frame on the display until your rendering catches up, which isn't great.

So while timewarp is a great tool, it isn't a magic wand for solving latency-related issues.



Timewarp Pipeline Bubbles

The major tradeoff in the fight to reduce latency is the creation of bubbles in the CPU/GPU pipeline. If you recall a few slides ago when we discussed frame queueing, the CPU and GPU were processing two different frames in parallel with each other, and that enabled both the CPU and the GPU to be used as efficiently as possible, without spending too much time waiting for each other.

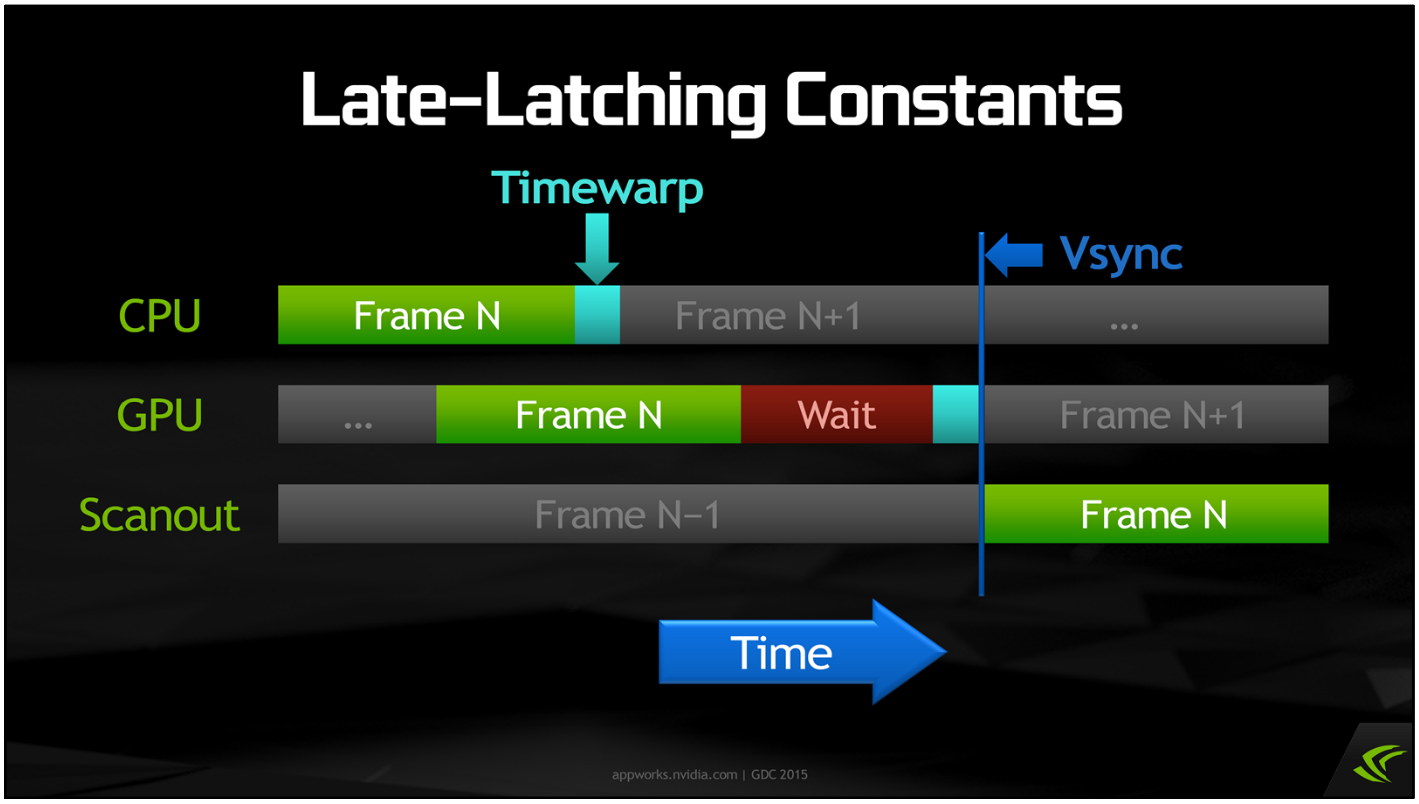
However, the Oculus SDK currently forces a completely synchronous model by flushing and waiting on the GPU via event queries. That is to say, in the Oculus SDK there is no notion of queuing frames ahead.

The CPU begins work at the start of a frame on the vertical blanking interval. The GPU then starts when either the CPU flushes the GPU, or command buffers fill up. At frame completion, the Oculus SDK waits to submit distortion just-in-time before the next vertical blank.

This achieves several latency advantages. One, the pose used by the CPU has a consistent delta from the previous frame. Additionally, the distortion/timewarp pass is run as close to the end of the frame as possible, as consistently as possible.

However, this means the CPU can't run ahead and start submitting the following frame — the app's render thread will be idle while we wait for the right time to kick off timewarp. And in turn, after the distortion/timewarp pass finishes, the GPU will go idle waiting for the CPU to submit the next frame.

So we have sacrificed CPU/GPU parallelism for latency. What can we do to get the parallelism back?

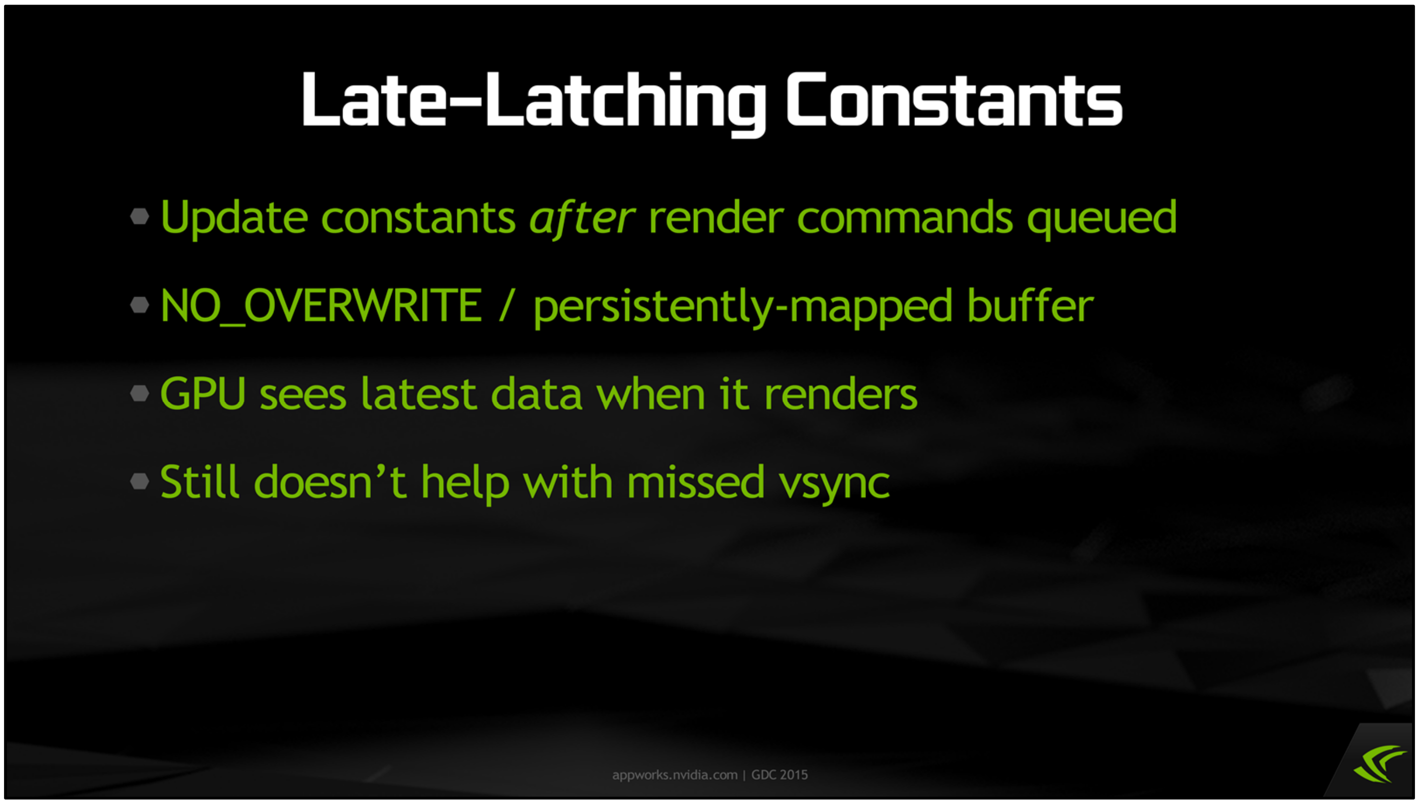


Late-Latching Constants

There are a couple different ways we can address this problem. The key point is that we had to wait to submit the timewarp rendering commands because the head pose, in the form of a set of constants in a constant buffer, gets latched in (so to speak) at the time the render commands are submitted.

However, what if we could continue to update the constants *after* the render commands are submitted? In that case, we don't need to wait to submit timewarp. We can submit it early and unblock the render thread, allowing it to proceed to the next frame.

We still do need to wait on the GPU, after the main rendering is finished, until just before vsync before we perform the timewarp pass. But this is no different from ordinary rendering with vsync enabled: there's little point in letting the GPU render faster than what the display can present.



Late-Latching Constants

- Update constants *after* render commands queued
- NO_OVERWRITE / persistently-mapped buffer
- GPU sees latest data when it renders
- Still doesn't help with missed vsync

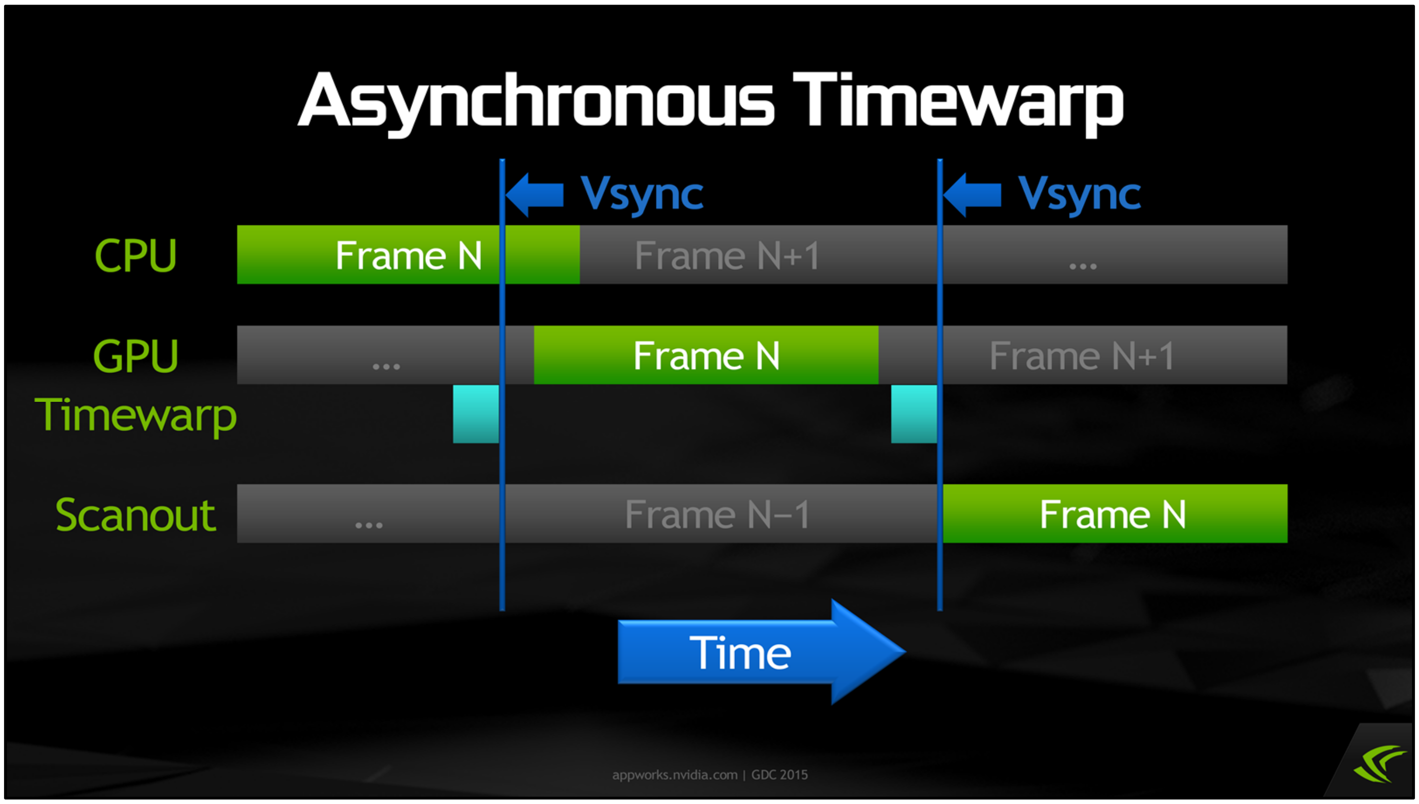
appworks.nvidia.com | GDC 2015

This notion of late-latching constants can be implemented in OpenGL 4.4 using its concept of a persistently-mapped buffer, which enables the CPU to update constants directly in video memory without tying them to the submission of rendering commands. All we have to do is set up a thread that updates the constants with new head poses, say every 1 ms, continuously. Then when the GPU executes the timewarp pass, it will see the latest head pose.

Direct3D 12 also supports persistently-mapped buffers. Direct3D 11 does but only indirectly by violating the NO_OVERWRITE contract provided when mapping constant buffers.

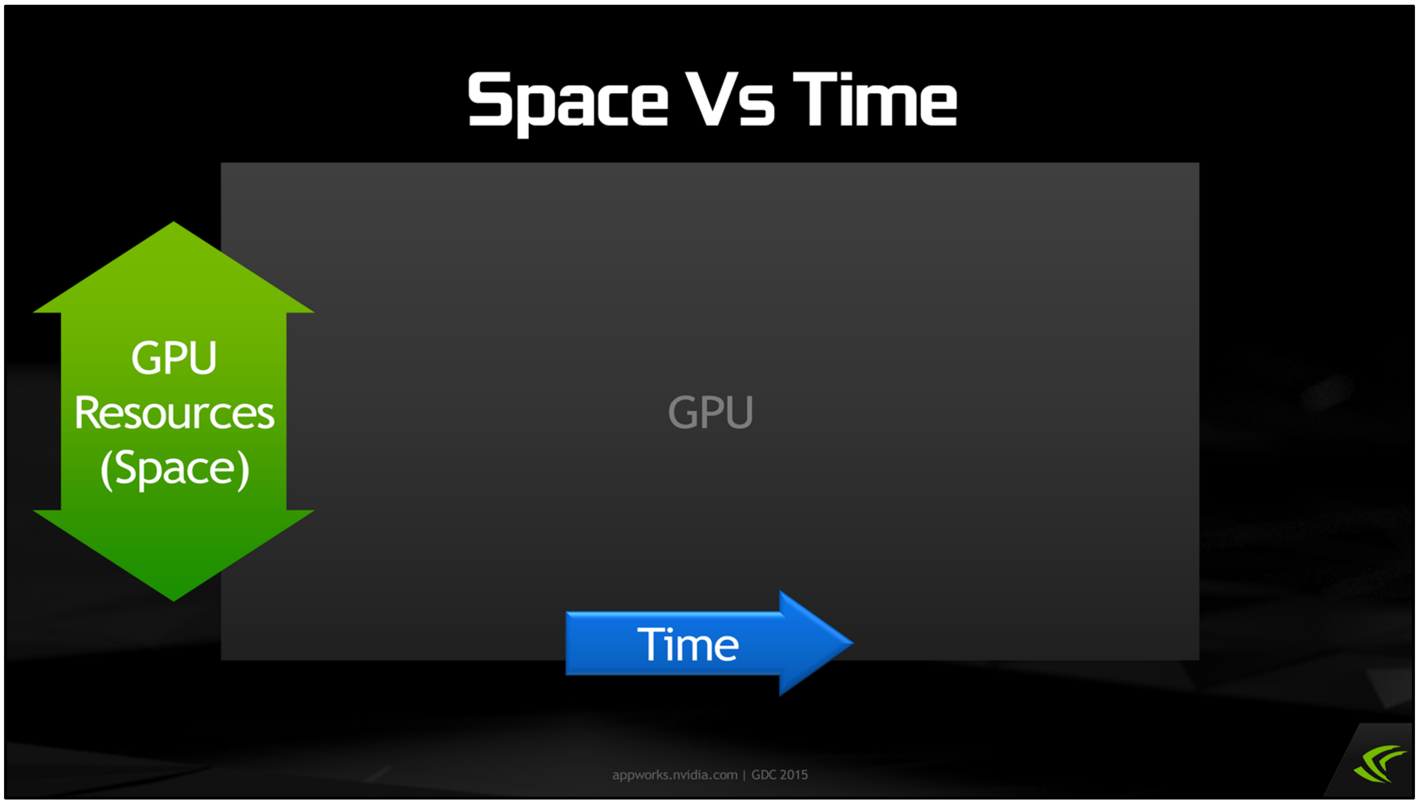
More information about late latching can be found in an Oculus blog post here: https://www.oculus.com/blog/optimizing-vr-graphics-with-late-latching/

With late-latching constants for timewarp, we can get rid of a lot of the bubbles in the CPU/GPU pipeline. However, we're still relying on the app rendering consistently at vsync rate. If the app has a hitch or the framerate drops for some reason, then we'll end up with a stuck or stuttering image on the headset. This is bad enough on an ordinary screen, but in VR it's really distracting and uncomfortable.



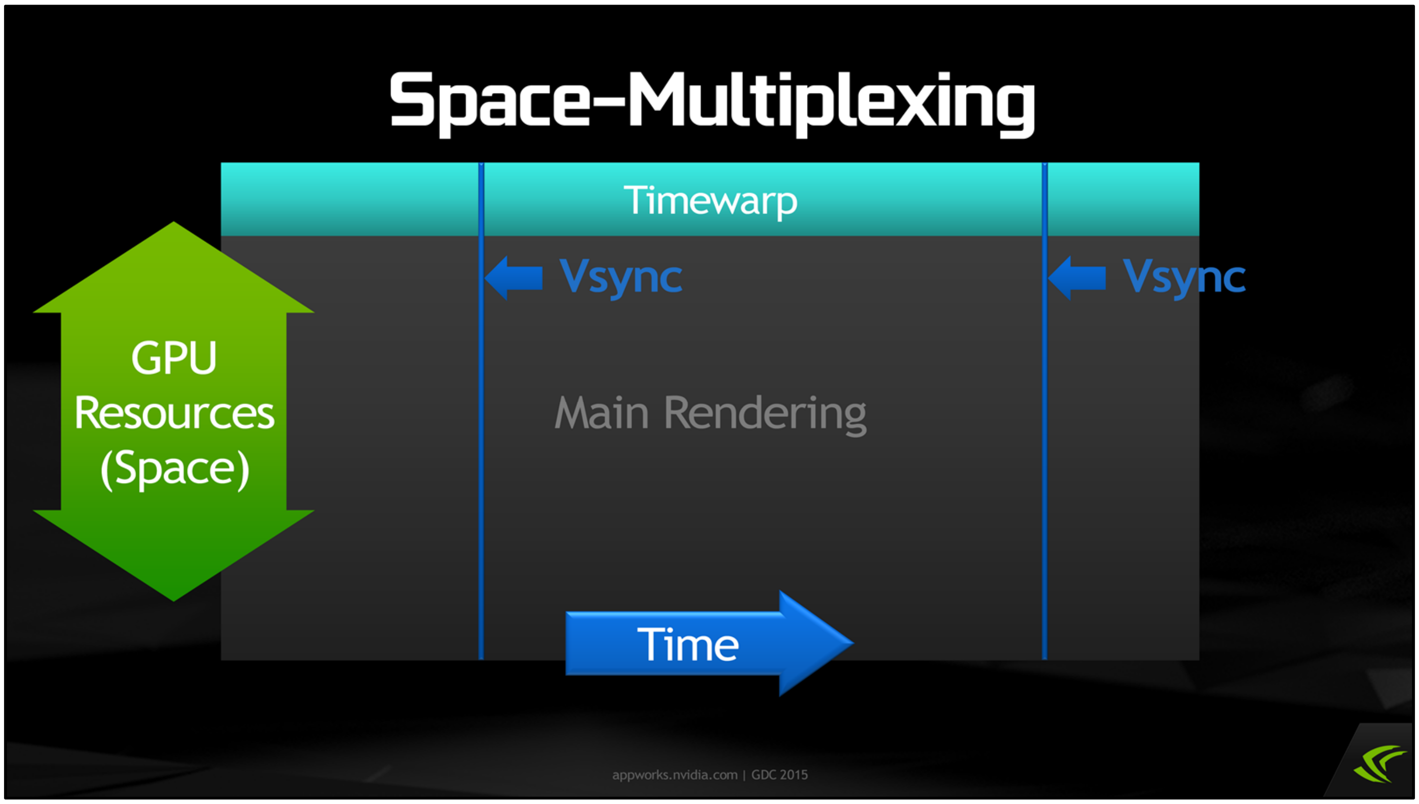
That brings us to asynchronous timewarp. The idea here is to decouple timewarp from the app's main rendering. Effectively it's like a separate process running on the GPU alongside the main rendering. It wakes up and runs right before each vsync, regardless of what else is going on with the system.

The interesting thing about this is it enables us to re-warp the previous frame if the main rendering process falls behind. So we will never end up with a stuck image on the headset, and we can continue to have low-latency rotation tracking even if the app hitches or its framerate drops.



Space Vs Time

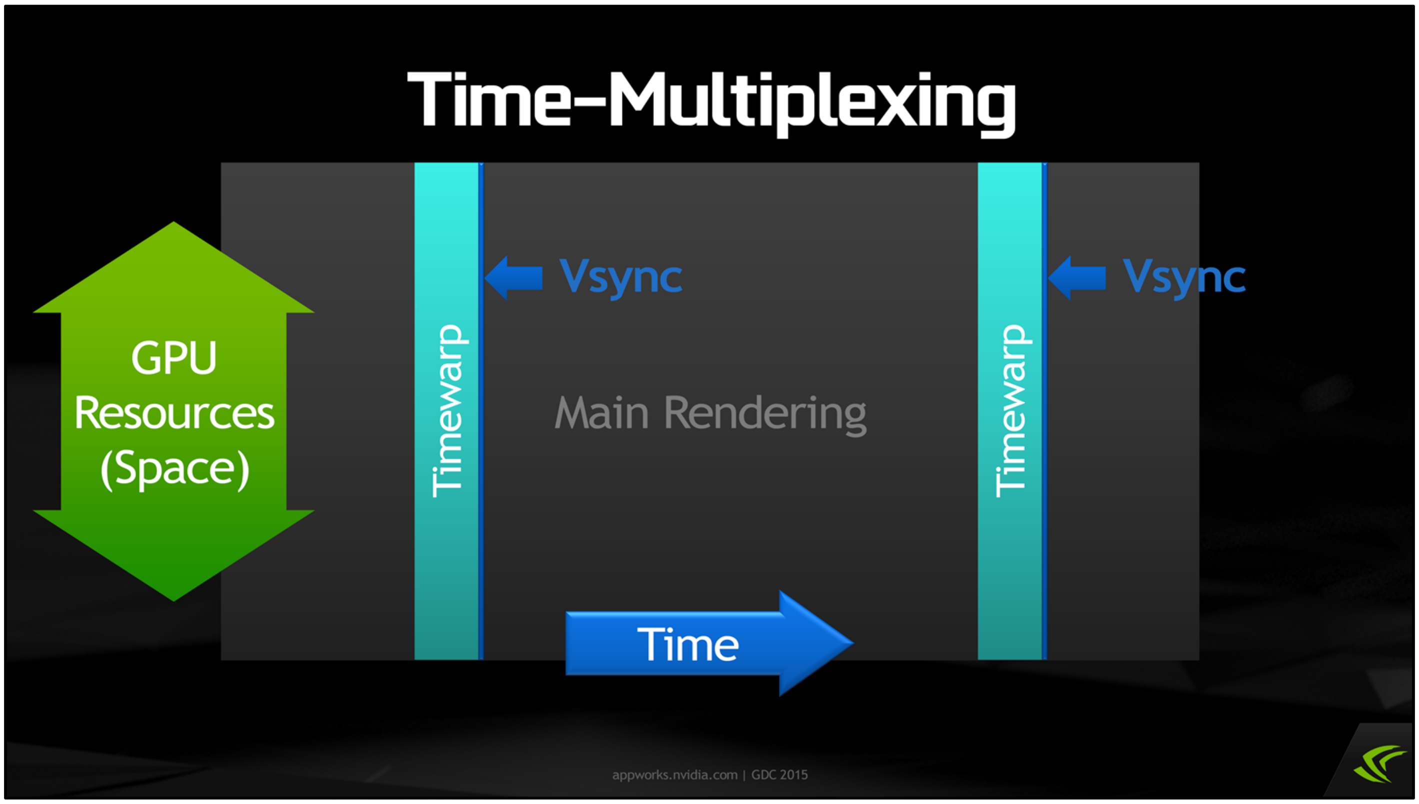
The goal is to make timewarp act like an independent process from main rendering. However, in reality, timewarp will have to share resources with the rest of the GPU. There are a range of ways to do this, depending on how we distribute the timewarp work over space and time.



Space-Multiplexing

One approach is to permanently reserve a small subset of GPU resources just for timewarp. With the minimum possible reservation, timewarp runs continuously, taking an entire frame time to complete each frame’s work.

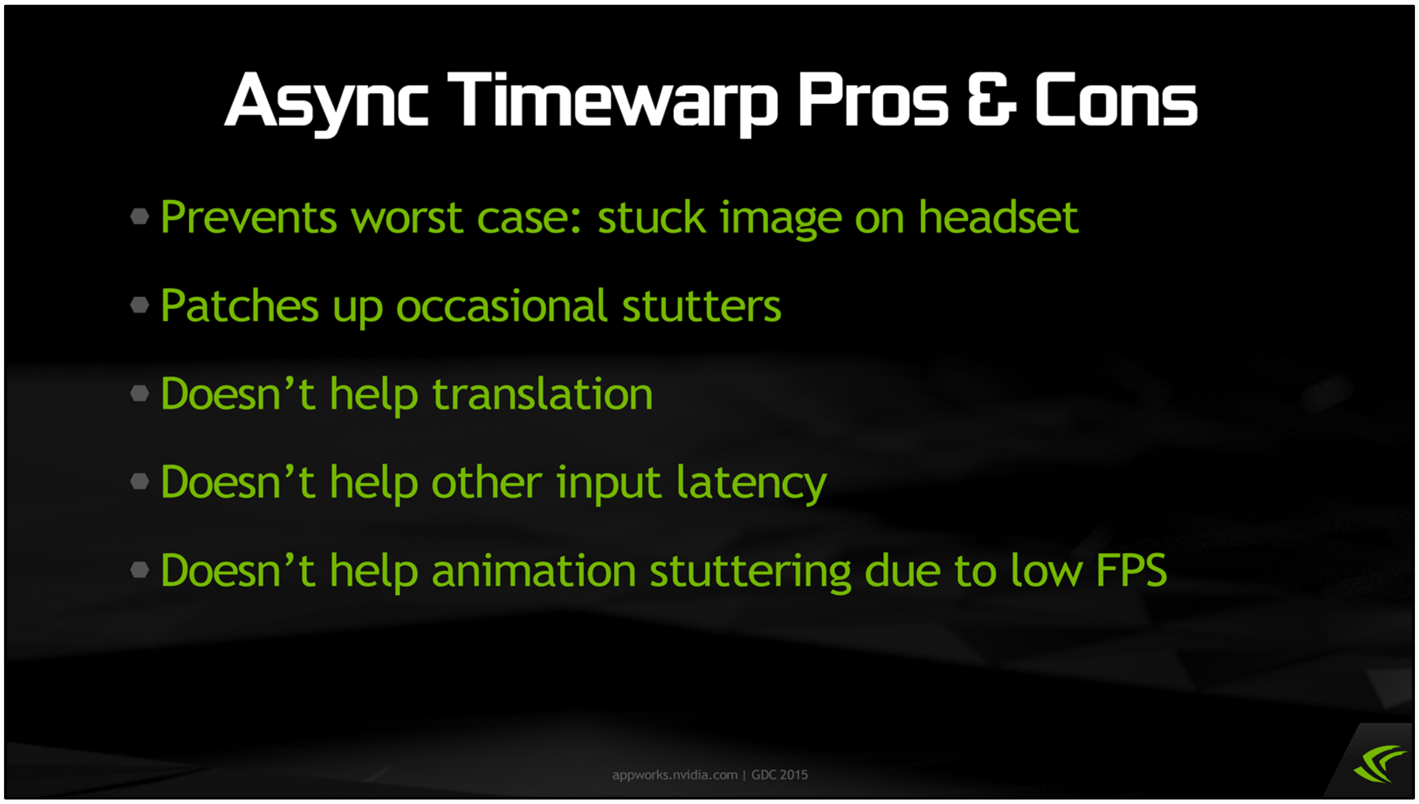
This works best in a rolling shutter display where we can race the beam. We slow down and do sections of timewarp to a front buffer just ahead of the beam. This allows us to limit the resources needed, but it’s tricky to ensure that the scheduling works consistently, and there are also cache flushing issues to contend with.



Time-Multiplexing

The other approach is to preempt and take over the entire GPU, running timewarp in the shortest possible time just before each vsync.

This involves working with (or around) the operating system in its graphics preemption system. Preemption is also limited by the hardware context switch granularity.

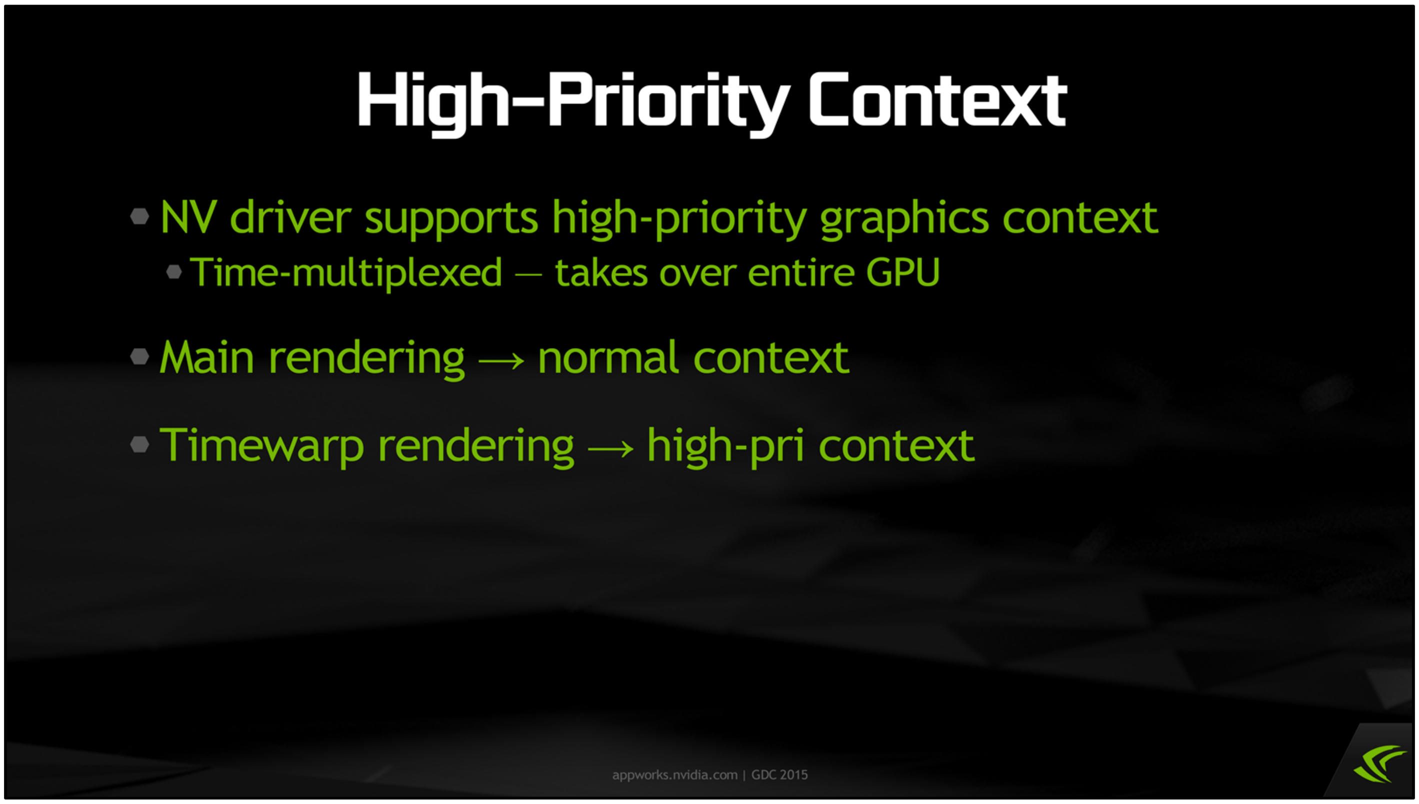


Async Timewarp Pros & Cons

- Prevents worst case: stuck image on headset
- Patches up occasional stutters
- Doesn’t help translation
- Doesn’t help other input latency
- Doesn’t help animation stuttering due to low FPS

Async timewarp is a great tool for preventing the uncomfortable scenario of a stuck image on the headset, patching up occasional stutters that cause an app to miss vsync.

However, it doesn’t address head translation (yet), latency of other inputs, or low animation frame rate.



High-Priority Context

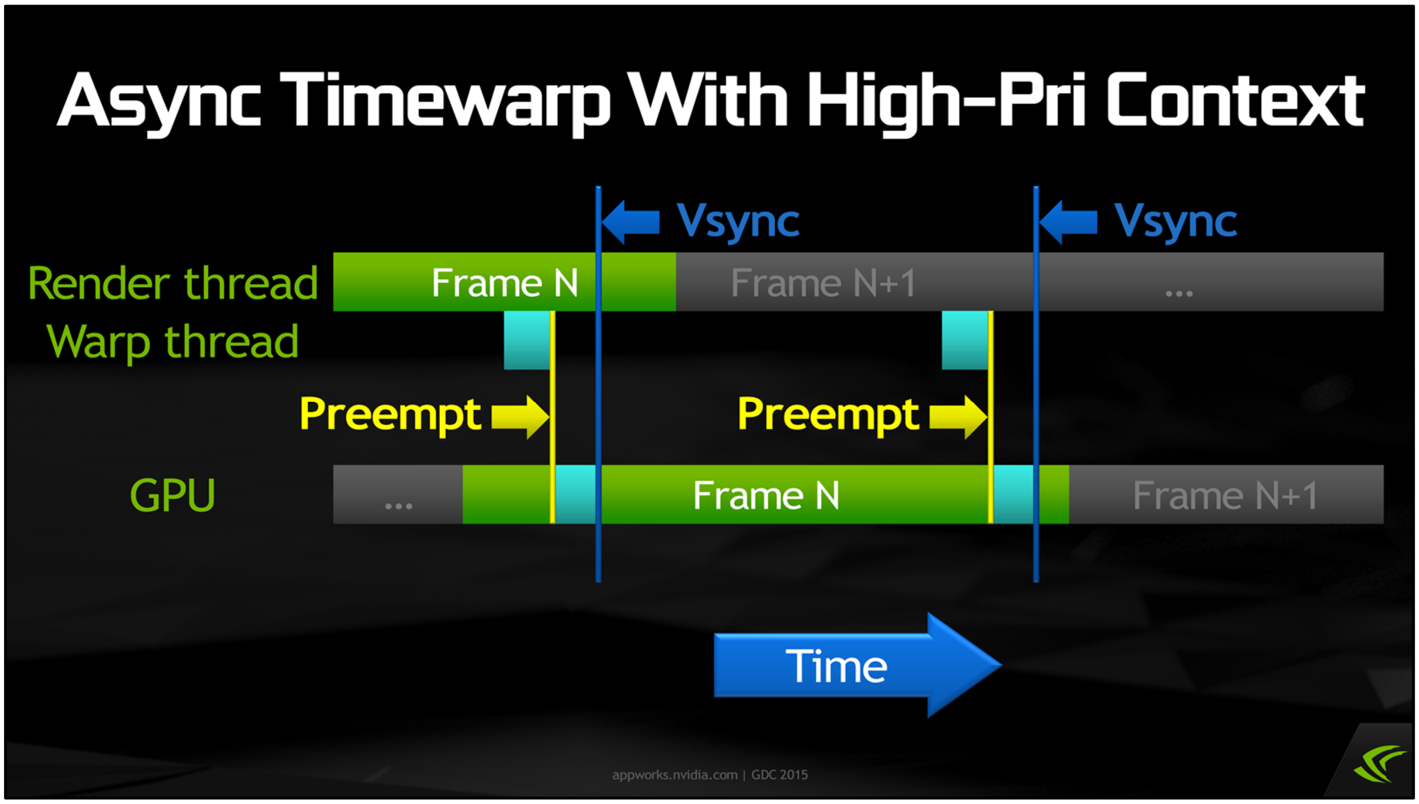
- NV driver supports high-priority graphics context
  - Time-multiplexed — takes over entire GPU
- Main rendering → normal context
- Timewarp rendering → high-pri context

appworks.nvidia.com | GDC 2015

So, Oculus has this concept of async timewarp that they've been thinking about for awhile. The question for NVIDIA is how can we actually enable Oculus to implement async timewarp on current hardware?

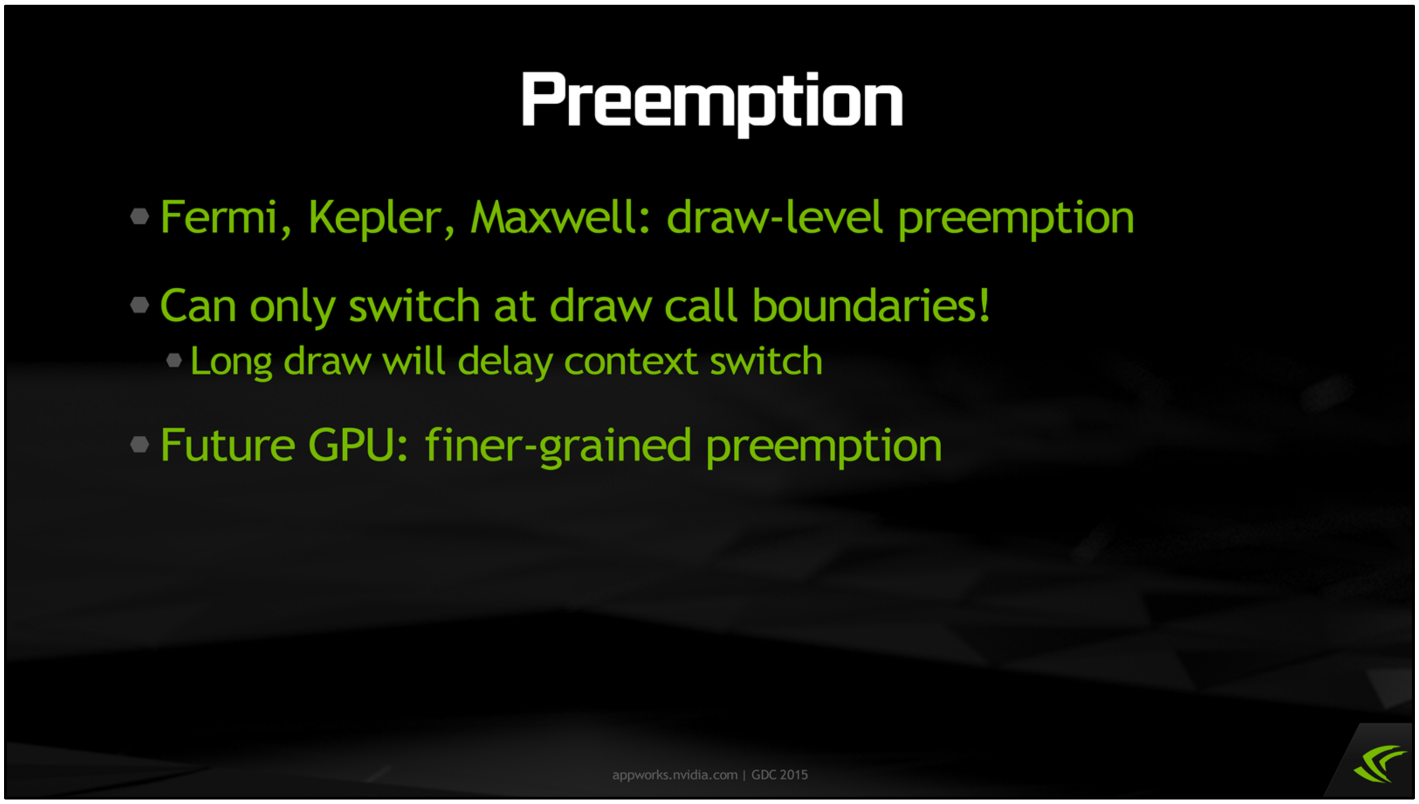
We're doing this by exposing support in our driver for a feature called a high-priority graphics context. This is very similar to what your OS desktop compositor already does to ensure the desktop can be rendered at a consistent framerate while a 3D app is running. It's a separate graphics context that takes over the entire GPU, preempting any other rendering that may already be taking place. In other words, it operates using the time-multiplexing method that was just explained.

With this feature exposed, we can have one ordinary rendering context, to which the app submits its usual work, plus a second, high-priority context where we can submit timewarp work.



Async Timewarp With High-Pri Context

Here's what async timewarp looks like in this picture. We have the app's main render thread submitting work to a main graphics context, like usual. We also have another thread running on the CPU, which sets up the high-priority context and submits the timewarp work to it a few milliseconds before each vsync. When that work hits the GPU, it preempts whatever's going on in the main context, performs the timewarp pass, then resumes what it was doing before.



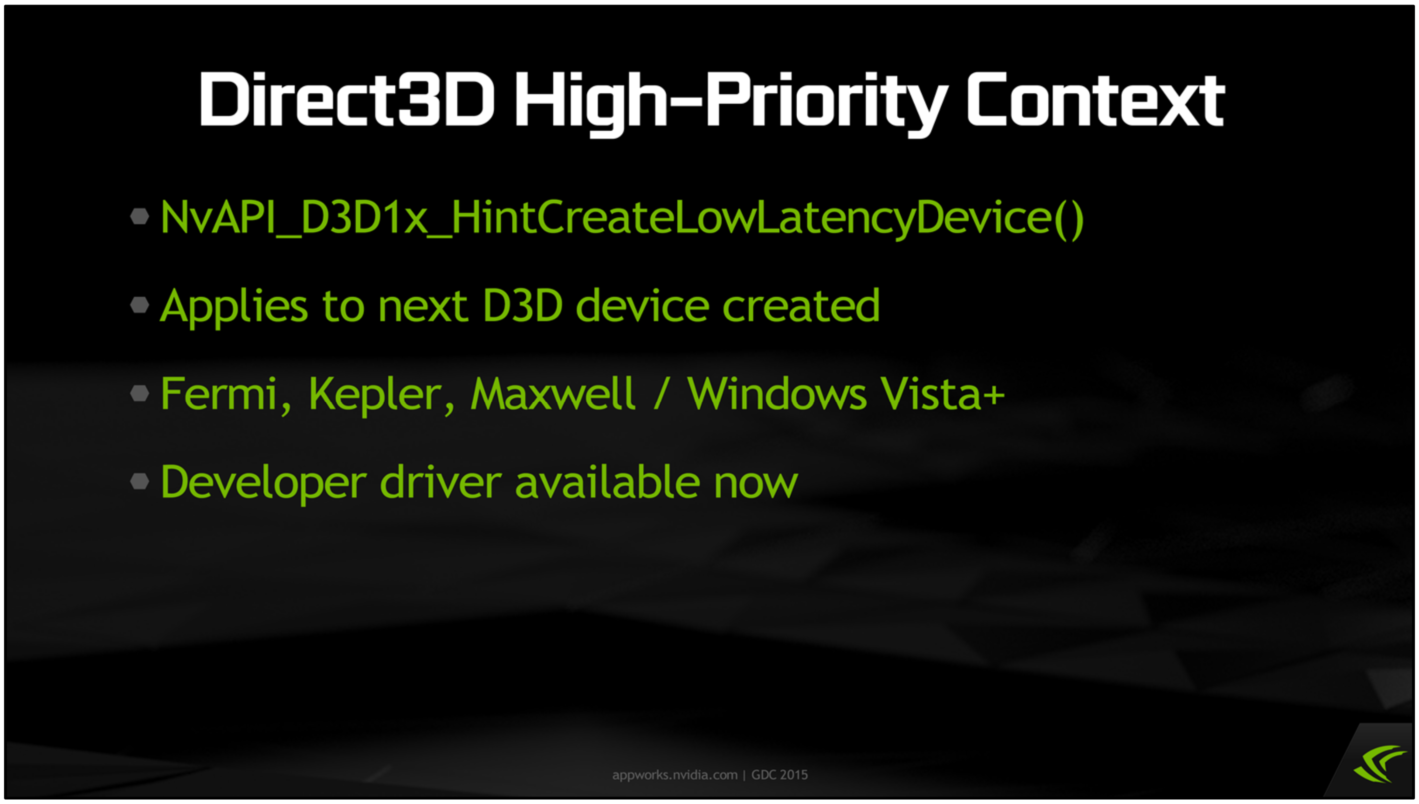
Preemption

- Fermi, Kepler, Maxwell: draw-level preemption
- Can only switch at draw call boundaries!
  - Long draw will delay context switch
- Future GPU: finer-grained preemption

appworks.nvidia.com | GDC 2015

Since we're relying on preemption, let's talk about how preemption actually works on current GPUs. Fermi, Kepler, and Maxwell GPUs — basically GeForce GTX 500 series and forward — all manage multiple contexts by time-slicing, with draw-call preemption. This means the GPU can only switch contexts at draw call boundaries! Even with the high-priority context, it's possible for that high-priority work to get stuck behind a long-running draw call on a normal context. If you have a single draw call that takes 5 ms, then async timewarp can get stuck behind it and be delayed for 5 ms, which will mean it misses vsync, causing stuttering or tearing.

On future GPUs, we're working to enable finer-grained preemption, but that's still a long way off.



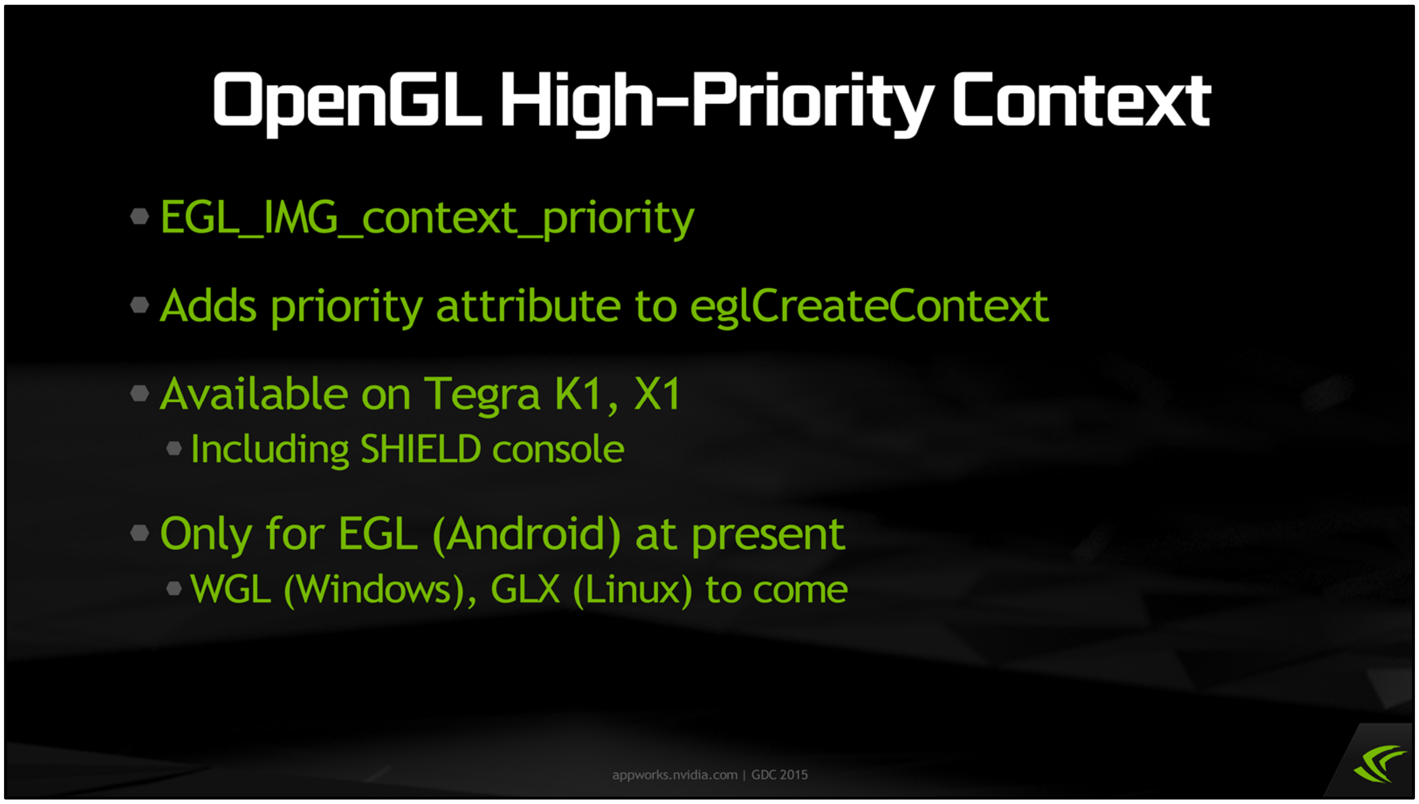
Direct3D High-Priority Context

- NvAPI_D3D1x_HintCreateLowLatencyDevice()
- Applies to next D3D device created
- Fermi, Kepler, Maxwell / Windows Vista+
- Developer driver available now

In Direct3D, we're exposing the high-priority context feature by adding an NvAPI function called "HintCreateLowLatencyDevice". This just takes a bool flag to turn the high-priority feature on and off. Any D3D devices created while the hint is set to "true" will be high-priority contexts. So, you'd turn the flag on, create the high-priority device, and turn it back off.

We have an NDA developer driver that exposes this feature available today. It works on Fermi, Kepler, and Maxwell cards on Windows Vista and up.

To be clear, game developers are not generally expected to be implementing async timewarp themselves; that's a job for Oculus. However, if you want to explore this on your own, we have the driver available.

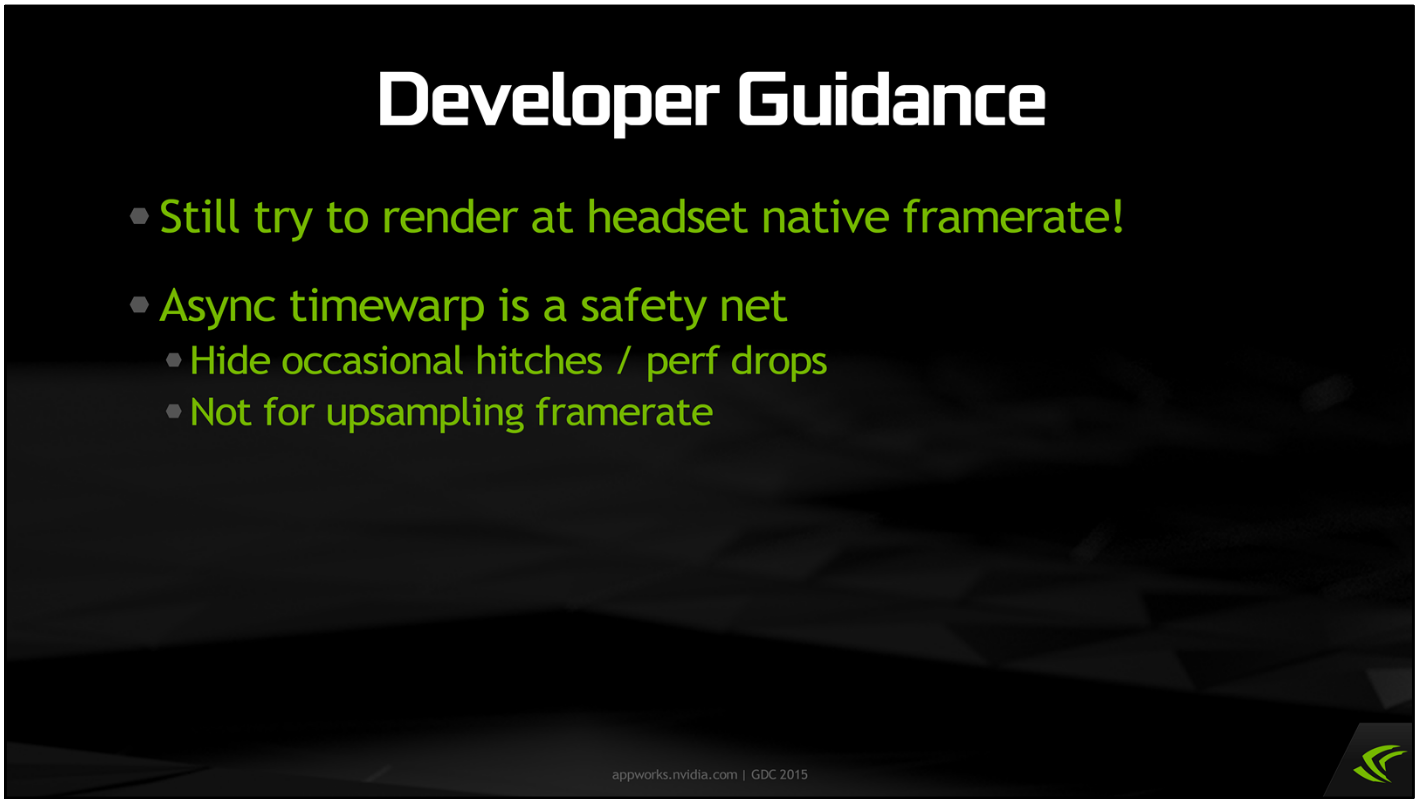


OpenGL High-Priority Context

- EGL_IMG_context_priority
- Adds priority attribute to eglCreateContext
- Available on Tegra K1, X1
  - Including SHIELD console
- Only for EGL (Android) at present
  - WGL (Windows), GLX (Linux) to come

As for OpenGL, on Android we already support the EGL_IMG_context_priority extension, which lets you select low, medium, or high priority when you create an EGL context. This is available on both Tegra K1 and Tegra X1, including on the SHIELD console.

To enable context priorities on desktop GL, we need to create corresponding extensions for WGL (Windows) and GLX (Linux). That hasn't happened yet, but we are pursuing it.



Developer Guidance

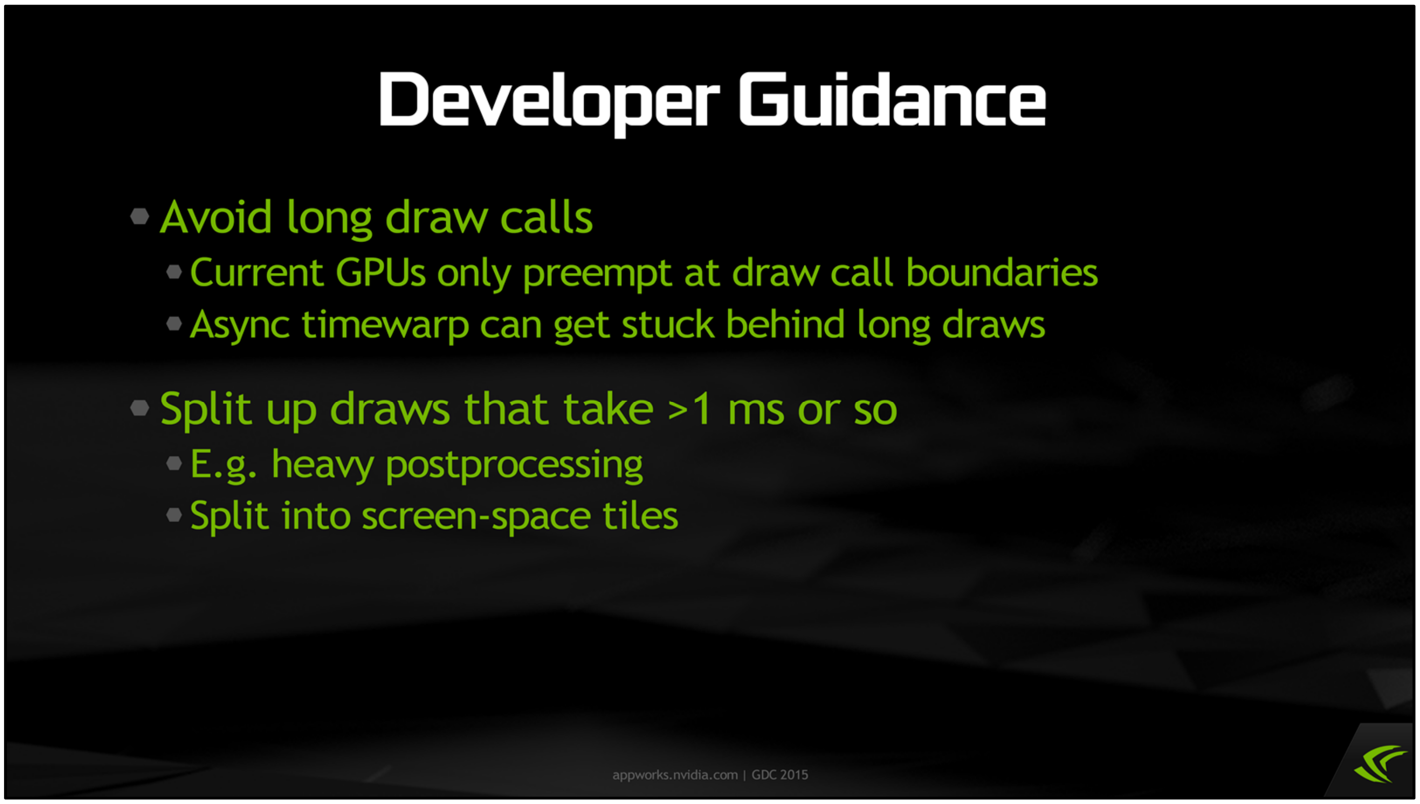
- Still try to render at headset native framerate!
- Async timewarp is a safety net
  - Hide occasional hitches / perf drops
  - Not for upsampling framerate

appworks.nvidia.com | GDC 2015

So what you, as a game developer, take away from all this?

Oculus is working on implementing async timewarp, but we want developers to understand that even async timewarp is not a magic wand to fix all latency-related issues. You really should still try to render at the headset's native framerate — 90 Hz for Crescent Bay — even though that's a difficult target to hit.

Async timewarp is really best looked at as a safety net to help cover up occasional hitches and perf drops. It is not a good tool for upsampling framerate — it's not going to work well for running 30 Hz content at 90 Hz, for example.



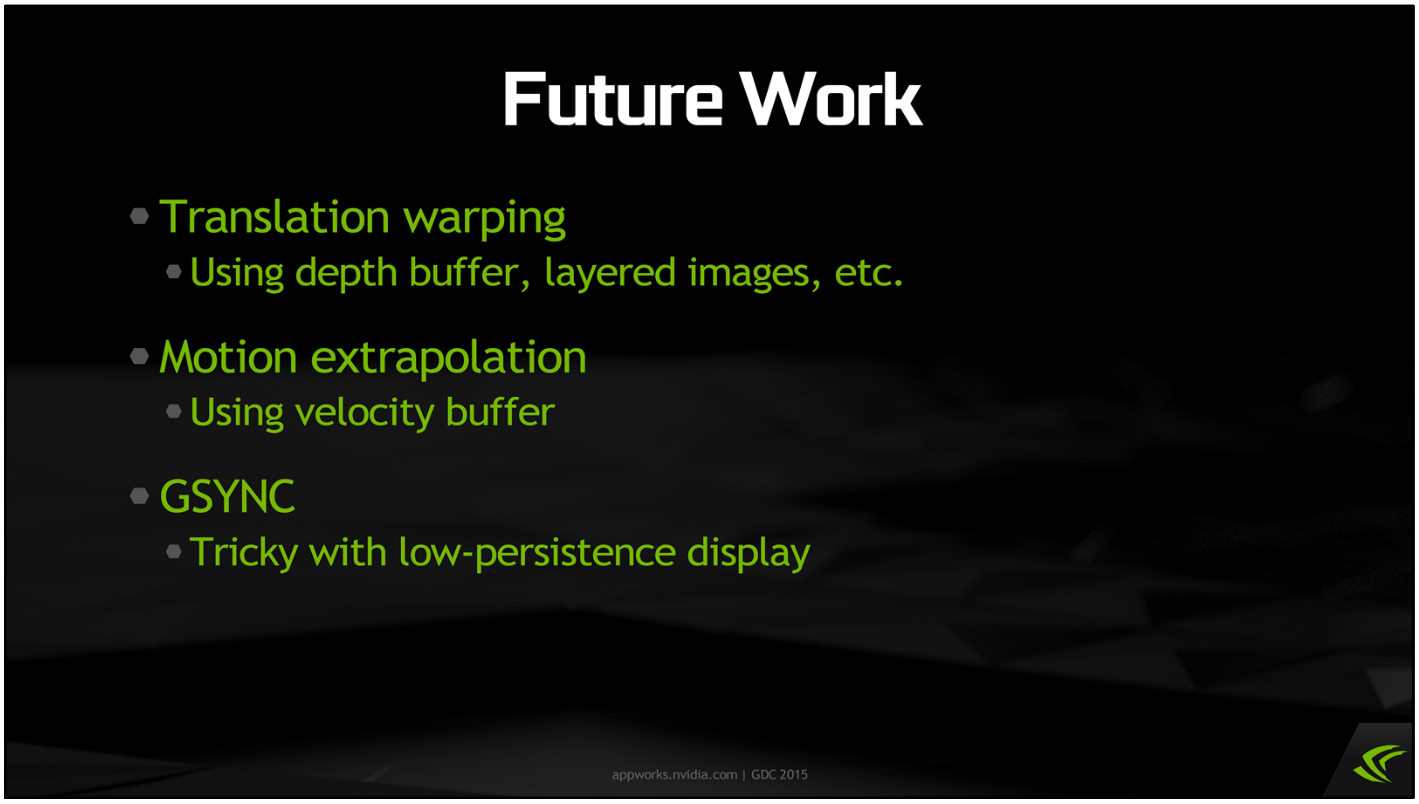
Developer Guidance

- Avoid long draw calls
  - Current GPUs only preempt at draw call boundaries
  - Async timewarp can get stuck behind long draws
- Split up draws that take >1 ms or so
  - E.g. heavy postprocessing
  - Split into screen-space tiles

appworks.nvidia.com | GDC 2015

A little while ago, we mentioned the limitations of preemption on current NVIDIA GPUs. Because we can only switch contexts at draw call boundaries, it's best for VR developers to split up long draw calls.

If you have individual draws that are taking longer than 1 ms or so, try to split them up into smaller ones. Heavy postprocessing passes are a common culprit here. For example, an SSAO pass or a depth-of-field pass where you're rendering a full-screen quad and running a heavy pixel shader may well take several milliseconds to complete. However, you can split up the screen into tiles and run the postprocessing on each tile in a separate draw call. That way, you provide the opportunity for async timewarp to come in and preempt in between those draws if it needs to.



Future Work

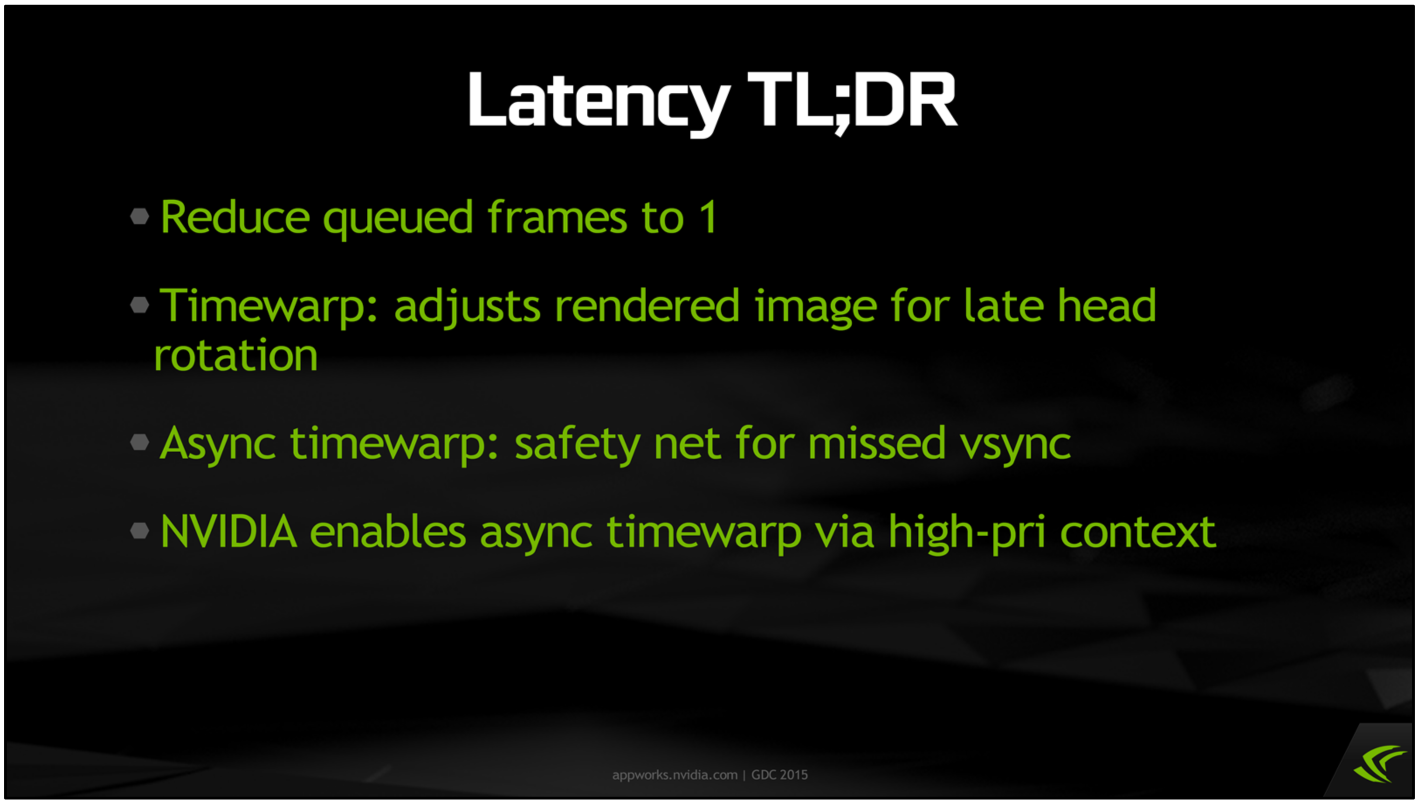
- Translation warping
  - Using depth buffer, layered images, etc.
- Motion extrapolation
  - Using velocity buffer
- GSYNC
  - Tricky with low-persistence display

Of course, we're always thinking about ways to make things better and improve timewarp and in the future. One very natural idea is to try to extend timewarp to handle head translation as well as rotation. At minimum, this needs a depth buffer to tell the timewarp kernel how far each object should parallax as you move your head. It would also be interesting to look at layered depth images, as seen in several recent research papers, to help fill in disocclusions.

Similarly, we could extend timewarp to extrapolate the motion of objects, helping to avoid animation stuttering. This could be done using velocity buffers, such as many engines already generate for motion blur.

Finally, it's interesting to consider using GSYNC, or other variable-refresh-rate technology, with VR. However, there are some non-trivial issues to solve there, relating to the interaction of GSYNC with the low-persistence display and with async timewarp.

All of these things are still very much in the research phase, but this is just a hint of some directions that VR technology might go in the future.



Latency TL;DR

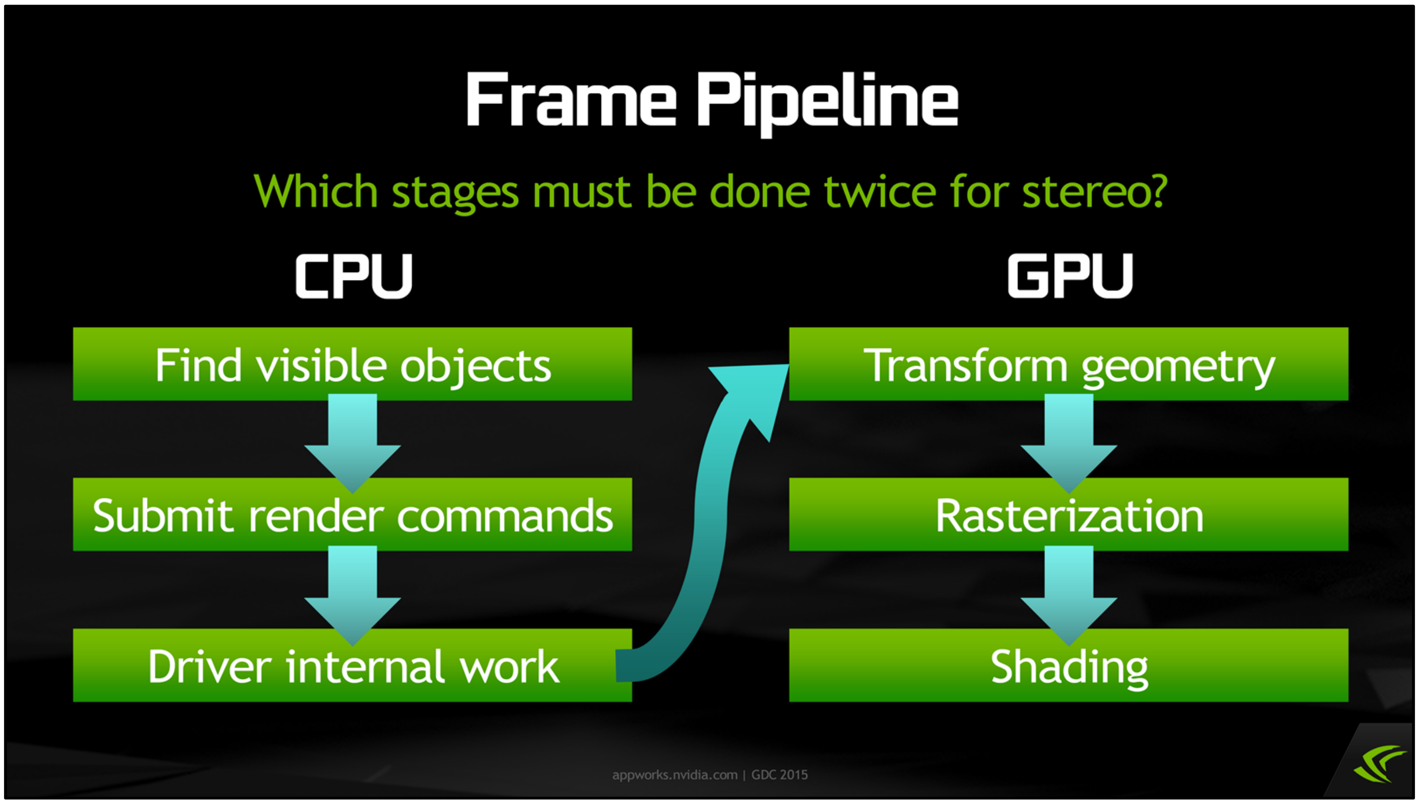
- Reduce queued frames to 1
- Timewarp: adjusts rendered image for late head rotation
- Async timewarp: safety net for missed vsync
- NVIDIA enables async timewarp via high-pri context

And that's it for the latency section of the talk. To recap: the Oculus SDK today reduces latency by limiting the size of the frame queue and by using timewarp to correct the rendered image for late head rotation. Going forward, Oculus and NVIDIA are working on asynchronous timewarp as a safety net in case a app occasionally misses vsync, and NVIDIA is enabling async timewarp on our hardware by exposing high-priority contexts as a feature.

Stereo Rendering

Multiview Rendering

VR SLI

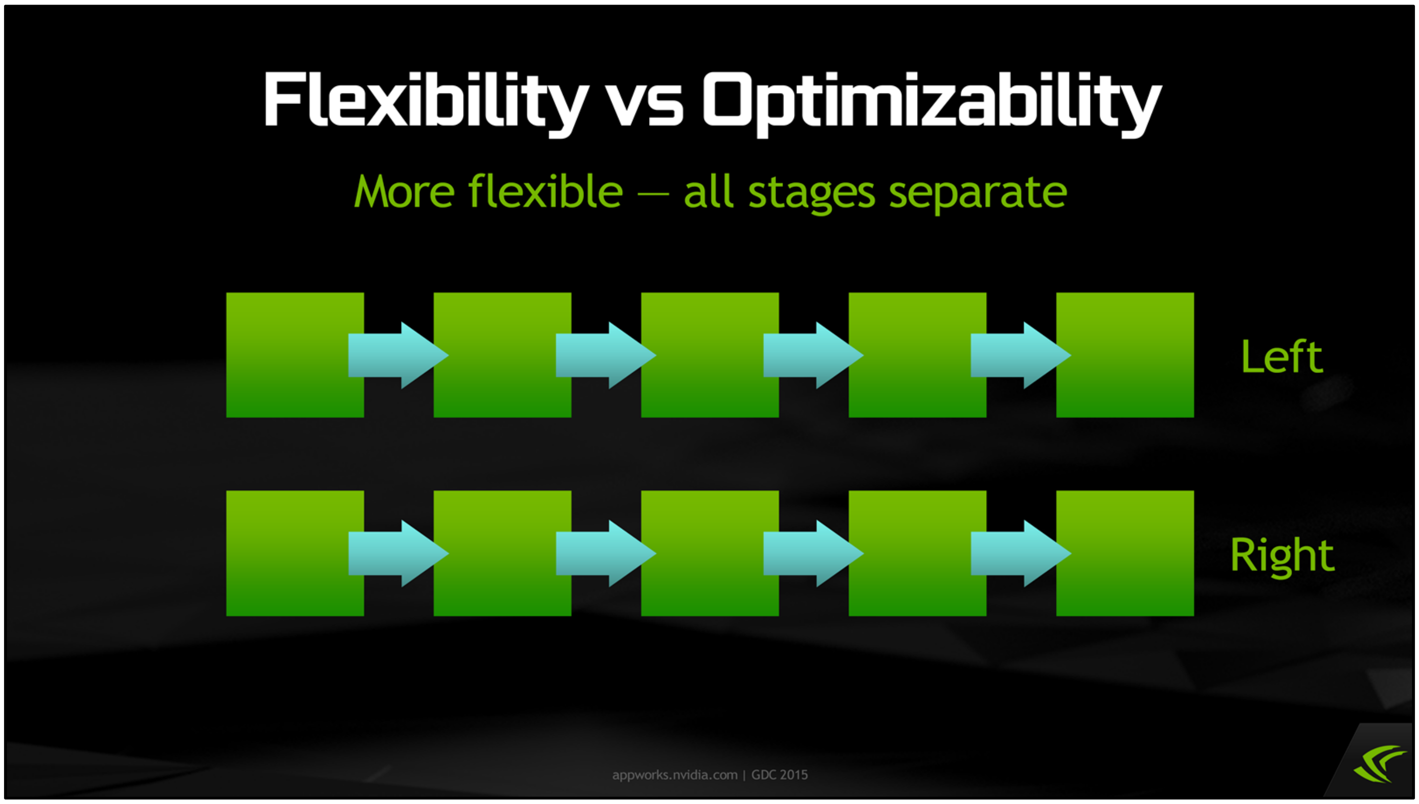


Frame Pipeline

Which stages must be done twice for stereo?

Here is the standard graphics pipeline today. The CPU processes the scene to find which objects to render, submits rendering commands, and then the driver/OS do a bunch of work validating and translating those commands, tracking and managing memory allocations, and so on. The GPU in turn transforms the geometry (or generates it, with tessellation and so forth), rasterizes it to screen space, and finally shades it.

Today, to render stereo eye views for VR, game engines are mostly running all of these stages twice. The question we want to ask is: can you save work by sharing some of these stages across views?



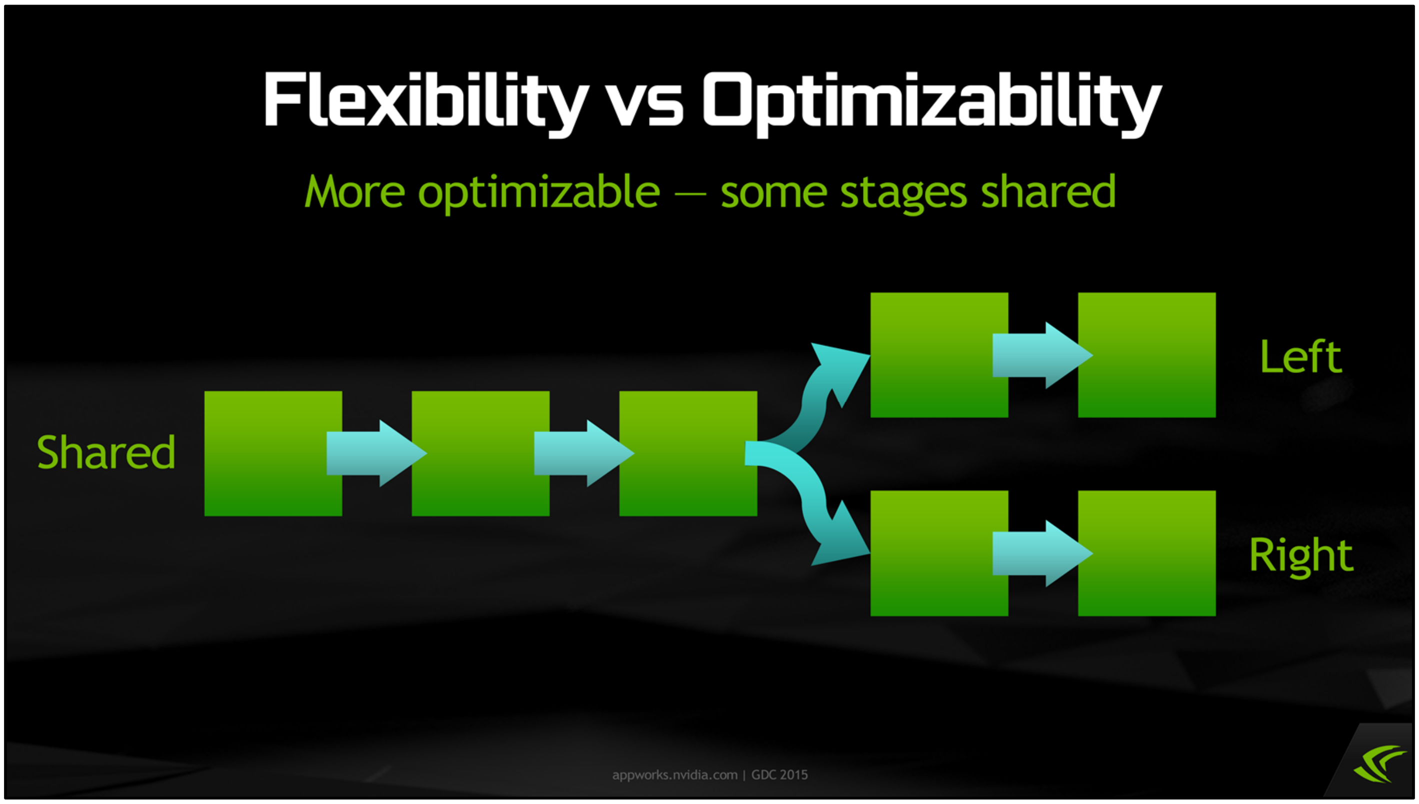
Flexibility vs Optimizability

More flexible — all stages separate

appworks.nvidia.com | GDC 2015

There are a range of different options in the stereo rendering space, and we have a trade-off between flexibility and optimizability.

At one end of the spectrum, with completely independent rendering for left and right eye, you have a lot of flexibility in how you set up your rendering, but it’s not very easy for the game engine, the driver, or GPU hardware to optimize this picture.

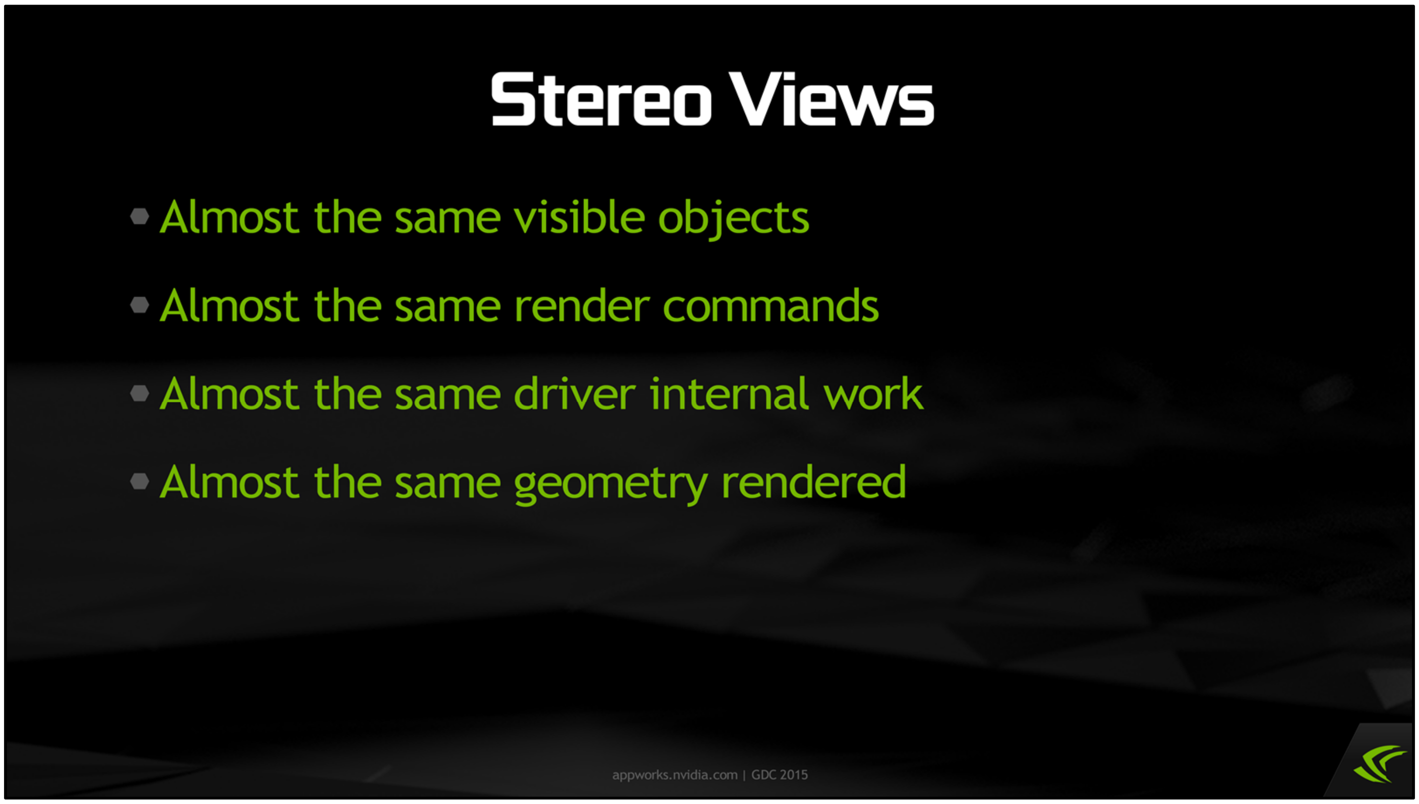


Flexibility vs Optimizability

More optimizable — some stages shared

If you give up the freedom of having completely independent rendering, you open up the opportunity to optimize, by sharing work in some of the rendering pipeline stages.

This is more restrictive, as you can't have view-dependent work happening in the earlier stages. But the farther down the pipeline you push that point where it forks in two, the more opportunities the driver and GPU have to optimize.

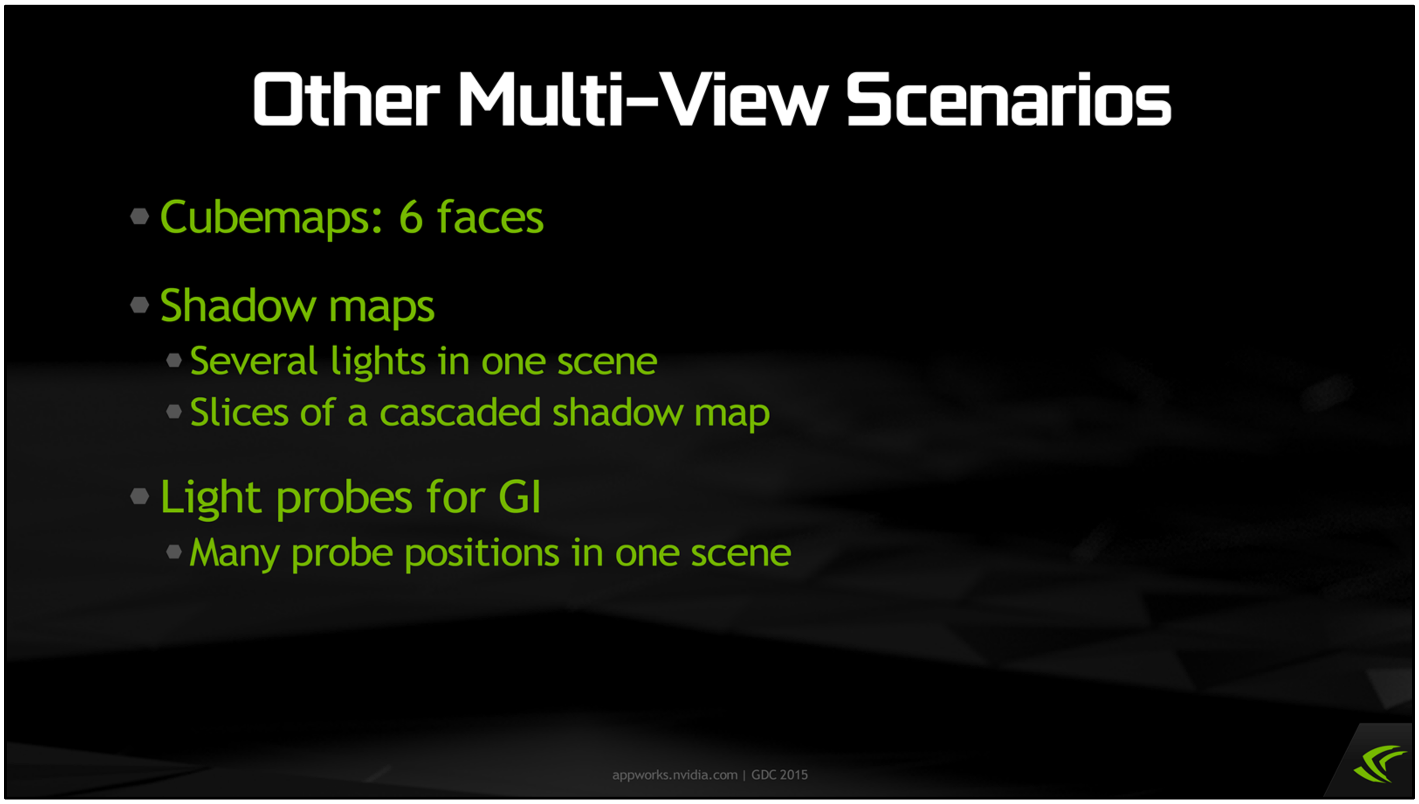


Stereo Views

- Almost the same visible objects
- Almost the same render commands
- Almost the same driver internal work
- Almost the same geometry rendered

With stereo rendering there's a great deal of redundancy between the two eye views. Visibility is almost always nearly identical. Because the same objects are visible, the same rendering commands are submitted. And because the rendering commands are the same, the driver translation/validation work is very similar as well. Finally, even if geometry is generated dynamically (e.g. adaptive tessellation), since the views are so close together, the geometry will be almost the same.

This suggests that by sharing many of the earlier stages of the rendering pipeline, we can save a lot of work while giving up minimal flexibility.

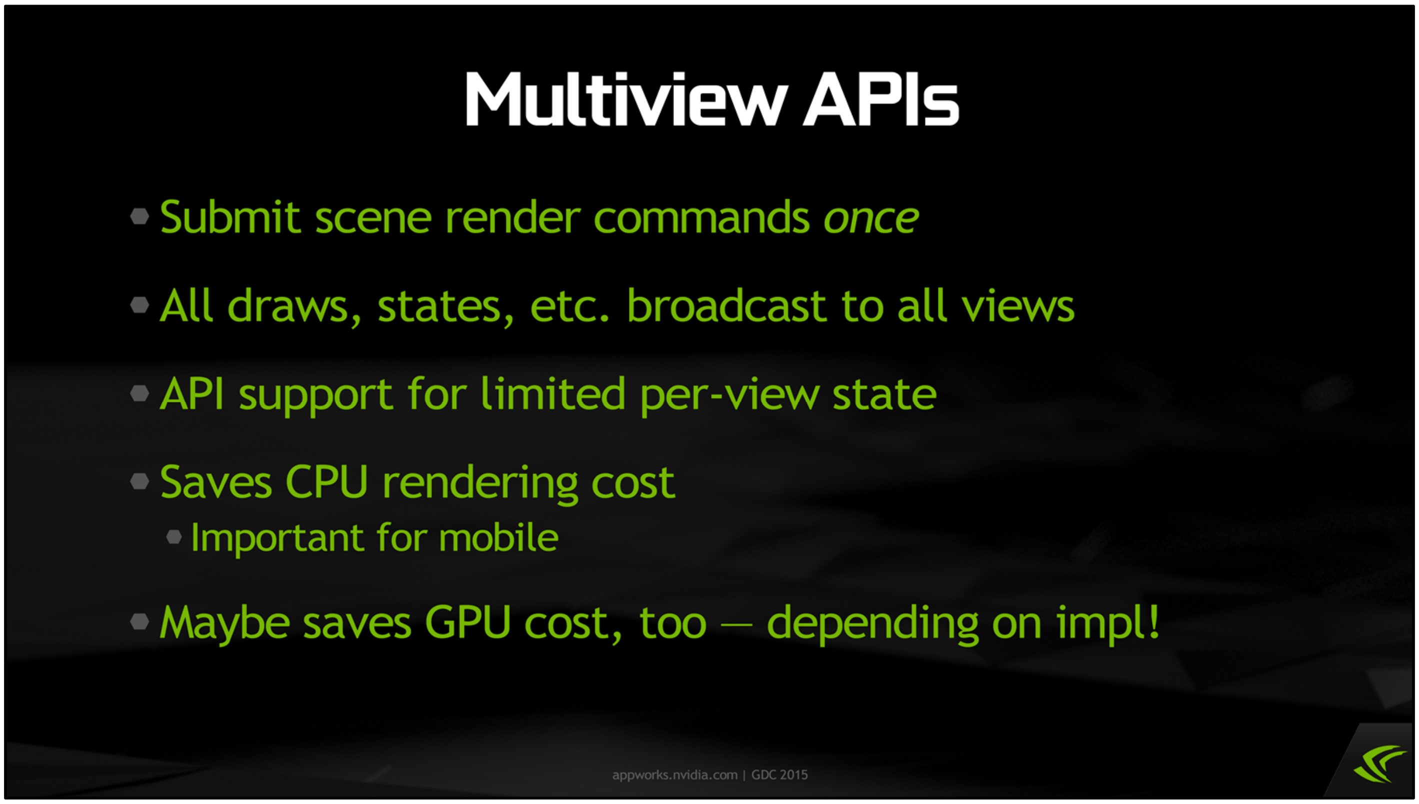


Other Multi-View Scenarios

- Cubemaps: 6 faces
- Shadow maps
  - Several lights in one scene
  - Slices of a cascaded shadow map
- Light probes for GI
  - Many probe positions in one scene

Although stereo is the primary case we're thinking about, it's worth noting that there are plenty of other scenarios where you want to render multiple views of a single scene containing the same objects and geometry — such as rendering the 6 faces of a cubemap, rendering several shadow maps (from different lights, or different slices of a cascaded shadow map), and rendering many light probes for global illumination.

In all of these cases, we'd like to be able to optimize by sharing redundant work across multiple views. So multiview rendering is more than just stereo.



Multiview APIs

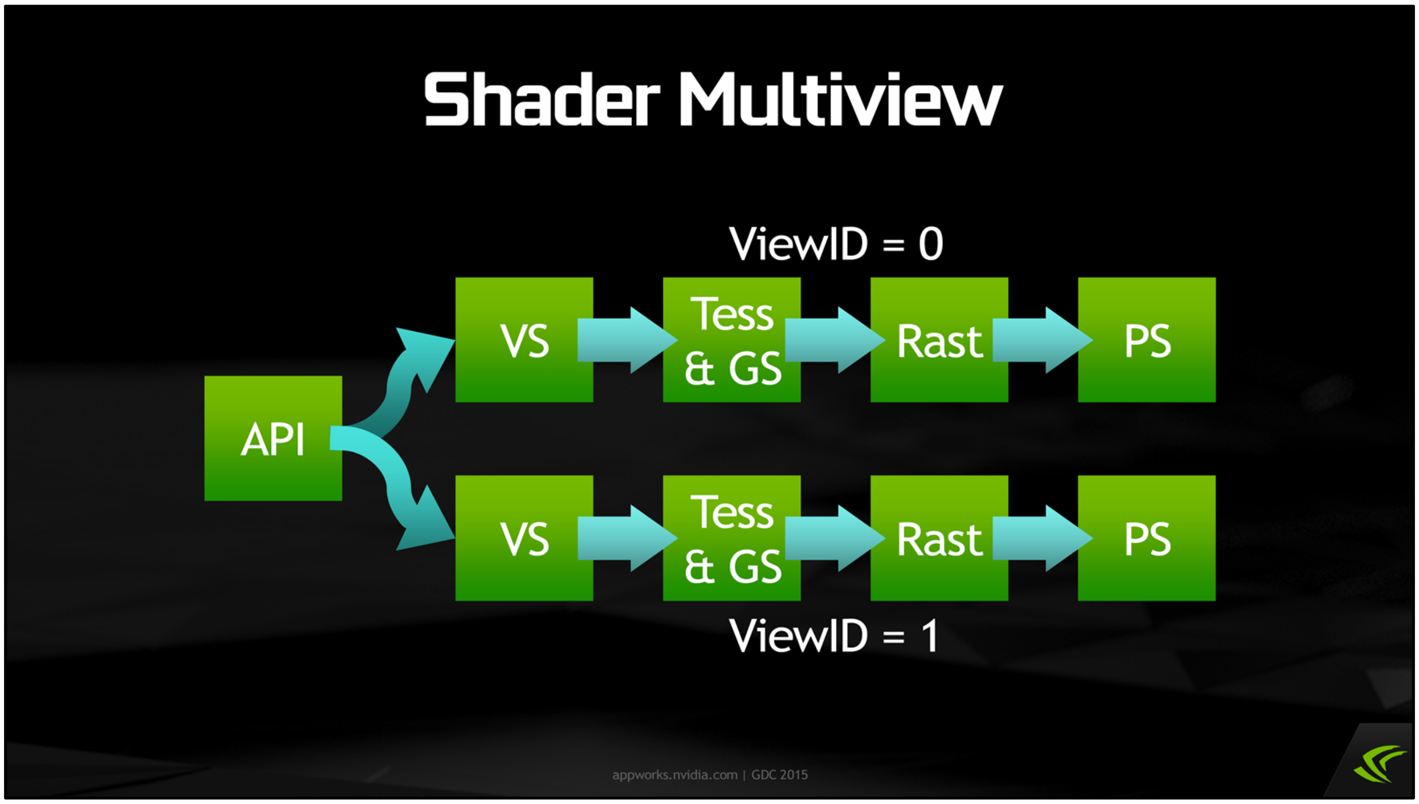
- Submit scene render commands *once*
- All draws, states, etc. broadcast to all views
- API support for limited per-view state
- Saves CPU rendering cost
  - Important for mobile
- Maybe saves GPU cost, too — depending on impl!

This is why we're looking into API-level and even hardware-level support for multiview rendering. There is a range of different proposals for how this might be done, occupying different points in that flexibility versus optimizability trade-off, but what they all have in common is that they accept a single stream of commands that renders two or more views.

All state changes, draw calls, and so on will be broadcast to all views by the implementation. There will be limited ways to configure per-view state — for example, to set different transforms for the left and right eyes.

Doing stereo rendering this way enables us to save a great deal of CPU work — both in the rendering engine and in the driver — by eliminating work that is redundant across the views. Saving CPU work is particularly important on mobile devices, but desktop devices benefit from it too.

Depending on the implementation, we may also be able to save some GPU work as well.

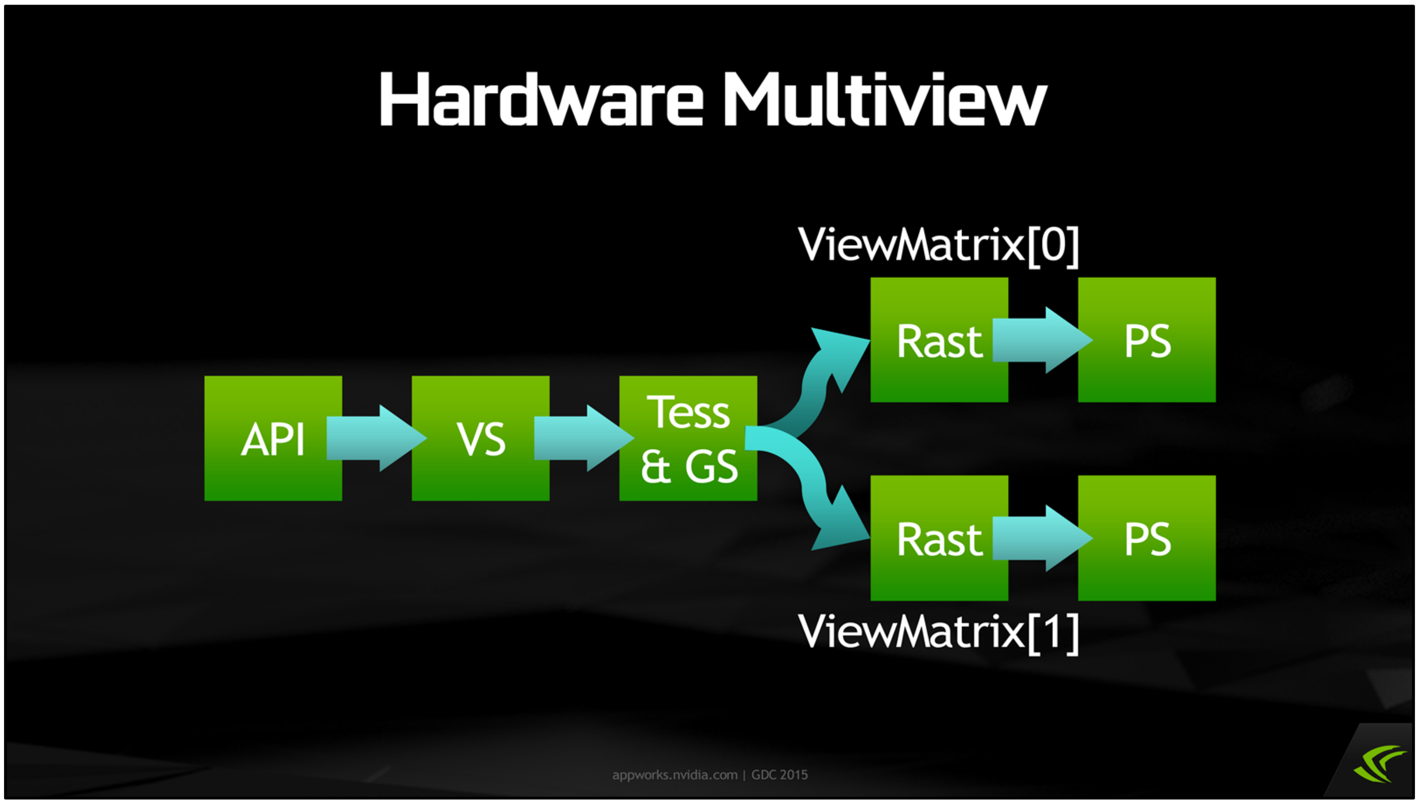


Shader Multiview

One possible implementation is to expose a ViewID variable to shaders. The GPU would launch separate shader threads for each view, with ViewID = 0 for the left eye and 1 for the right eye, for instance. This can be simulated today using geometry instancing, as described in this doc: https://docs.google.com/presentation/d/19x9XDjUvkW_9gsfsMQzt3hZbRNziVsoCEHOn4AercAc/ In the future, multiview could be explicitly supported in shading languages and APIs, as a distinct feature from instancing.

The application could pass in a constant buffer containing view-dependent information, and shaders could lookup into it based on ViewID to find transforms or whatever other view-specific data they needed. Furthermore, the shaders themselves could behave differently per ViewID.

This does imply that all the shader stages would have to be run separately for each view.



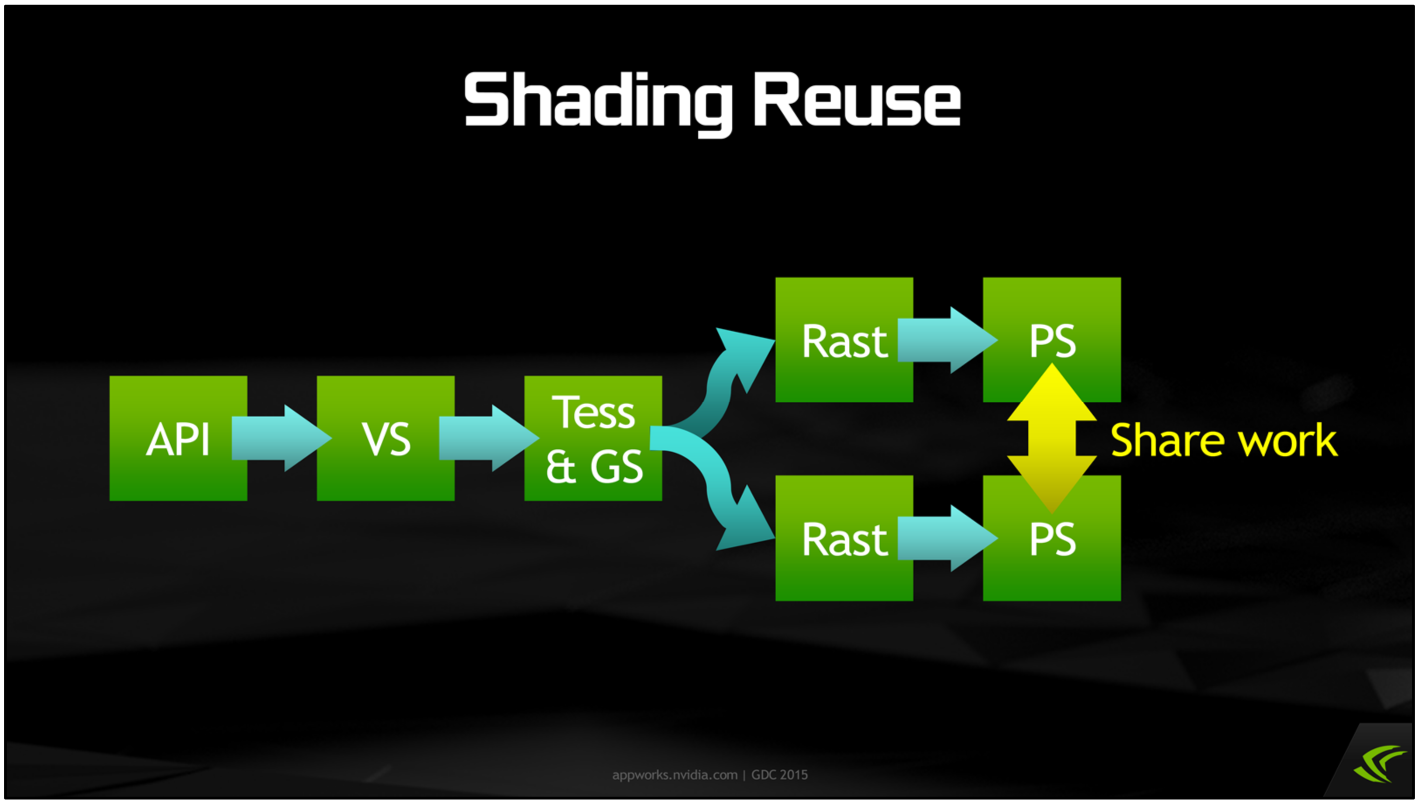
Hardware Multiview

Another possible implementation is to run the geometry pipeline once, including vertex, tessellation, and geometry shaders, and add hardware that can split off the views at the rasterization stage. This implies that we'd render exactly the same geometry in all views.

To generate appropriately-transformed vertex positions for the rasterizer for each view, we'd introduce a fixed-function 4×4 matrix (blast from the past!) configured per view. The output position from the geometry pipeline would get transformed by these matrices before going into rasterization and pixel shading. (We could do this replication and per-view transform using a geometry shader, but geometry shaders are really slow and not a good fit for this.) Pixel shaders would have access to a ViewID variable, similar to the previous slide.

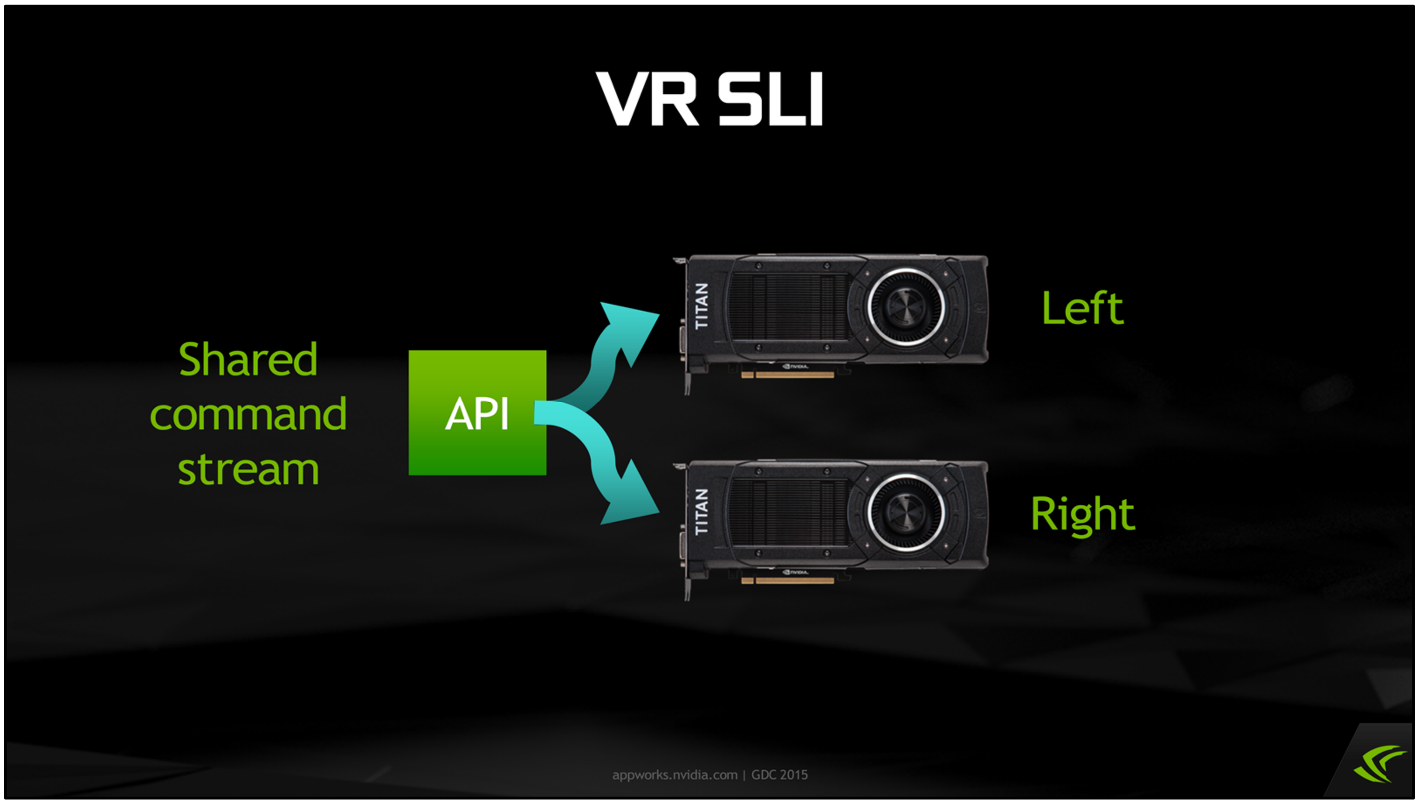
A complication with this approach is what to do about other interpolated attributes. If shaders are outputting other vectors or data that's view-dependent in some way, the implementation isn't going to know how to transform them. Either the shaders will have to explicitly generate per-view attribute values, or those calculations will need to be moved to the pixel shader.

Since it requires GPU hardware changes to fully realize this approach, it's a long way off. However, if APIs are designed today with this possibility in mind, the API can automatically fall back to a shader-based approach (as on the previous slide) on hardware that doesn't support multiview directly.



Shading Reuse

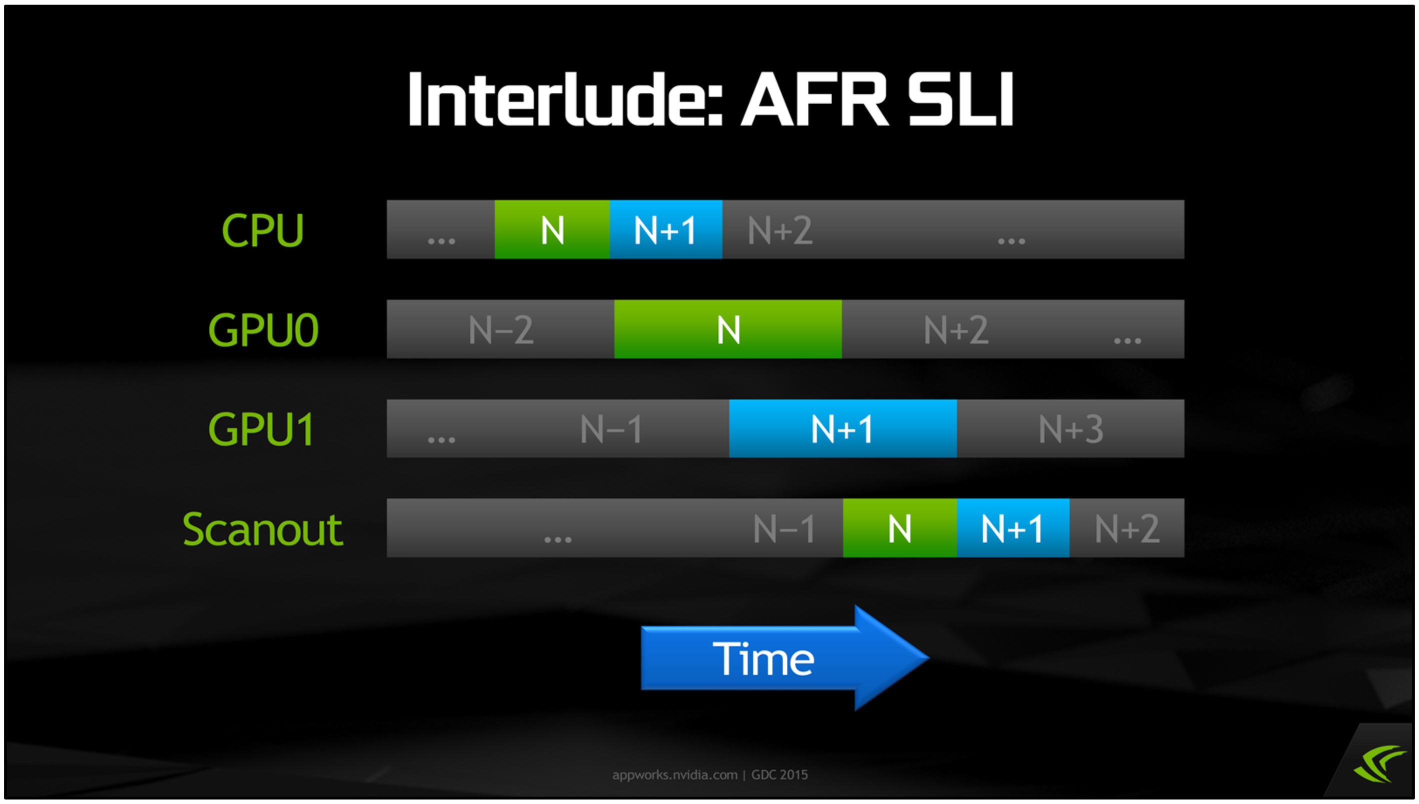
Although this is somewhat of a handwavy future possibility, it’s also interesting to explore possibilities for reusing shading work between views. We can already do this to some extent with texture-space shading. However, the obvious complication is view-dependent shading phenomena such as specular reflections.



VR SLI

One very natural extension of these multiview rendering ideas we've been talking about is to distribute the views across multiple GPUs. In the case of stereo, you'd have two GPUs, each rendering one eye view, but driven by a single command stream from the application. We call this VR SLI, and we've created an API to enable it.

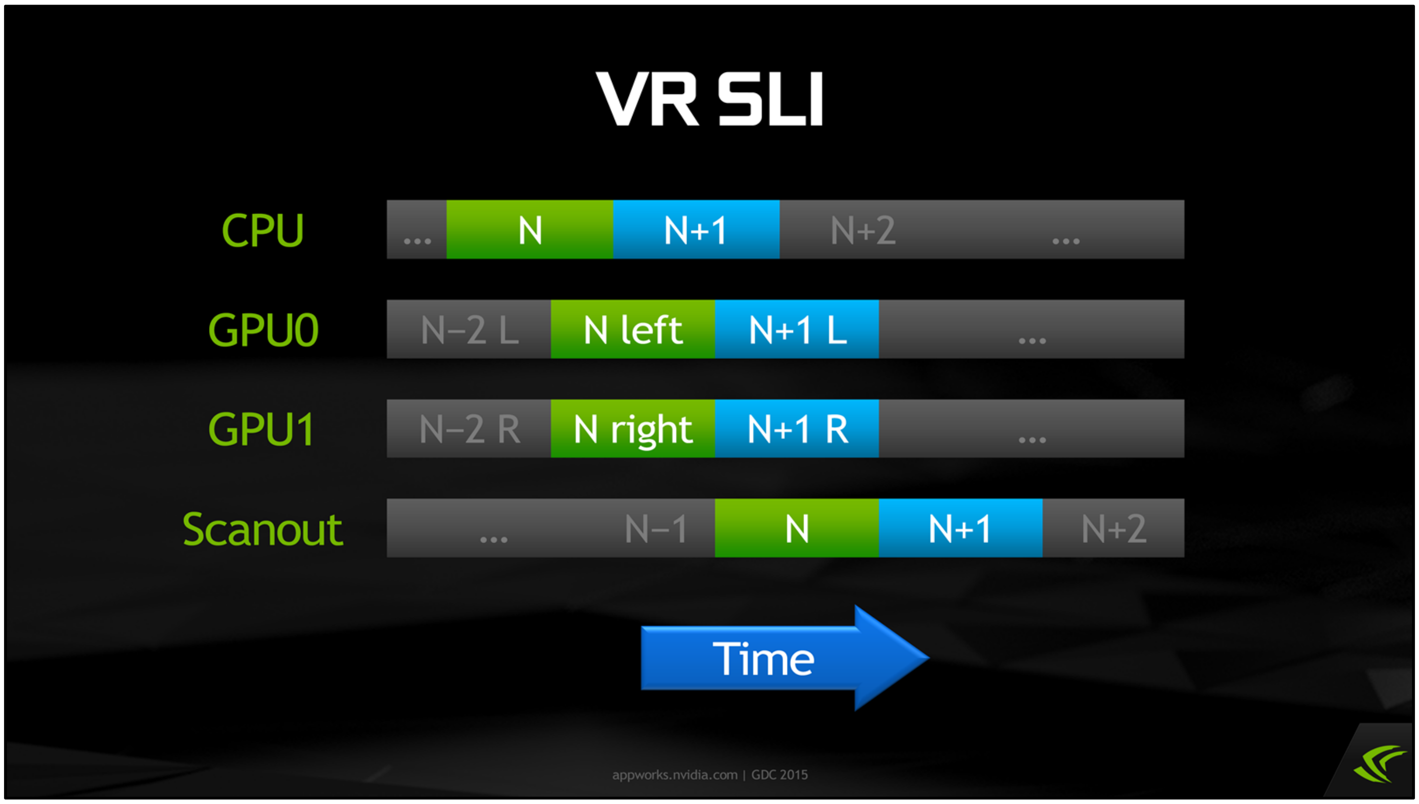
Now, one way to feed two GPUs is to let the application provide two completely independent command streams and send totally different rendering to each GPU. That's a very flexible solution, but the problem is you then need to burn the CPU time to generate those two command streams in your engine and process them through the driver. If we trade off the flexibility of independent command streams, by producing one command stream for both GPUs, we can save a lot of CPU work.



Interlude: AFR SLI

Before we dig into VR SLI, as a quick interlude, let me first explain how SLI normally works. For years, NVIDIA has had alternate-frame rendering, or AFR SLI, in which the GPUs trade off whole frames. In the case of two GPUs, one renders the even frames and the other the odd frames. The GPU start times are staggered half a frame apart to try to maintain regular frame delivery to the display.

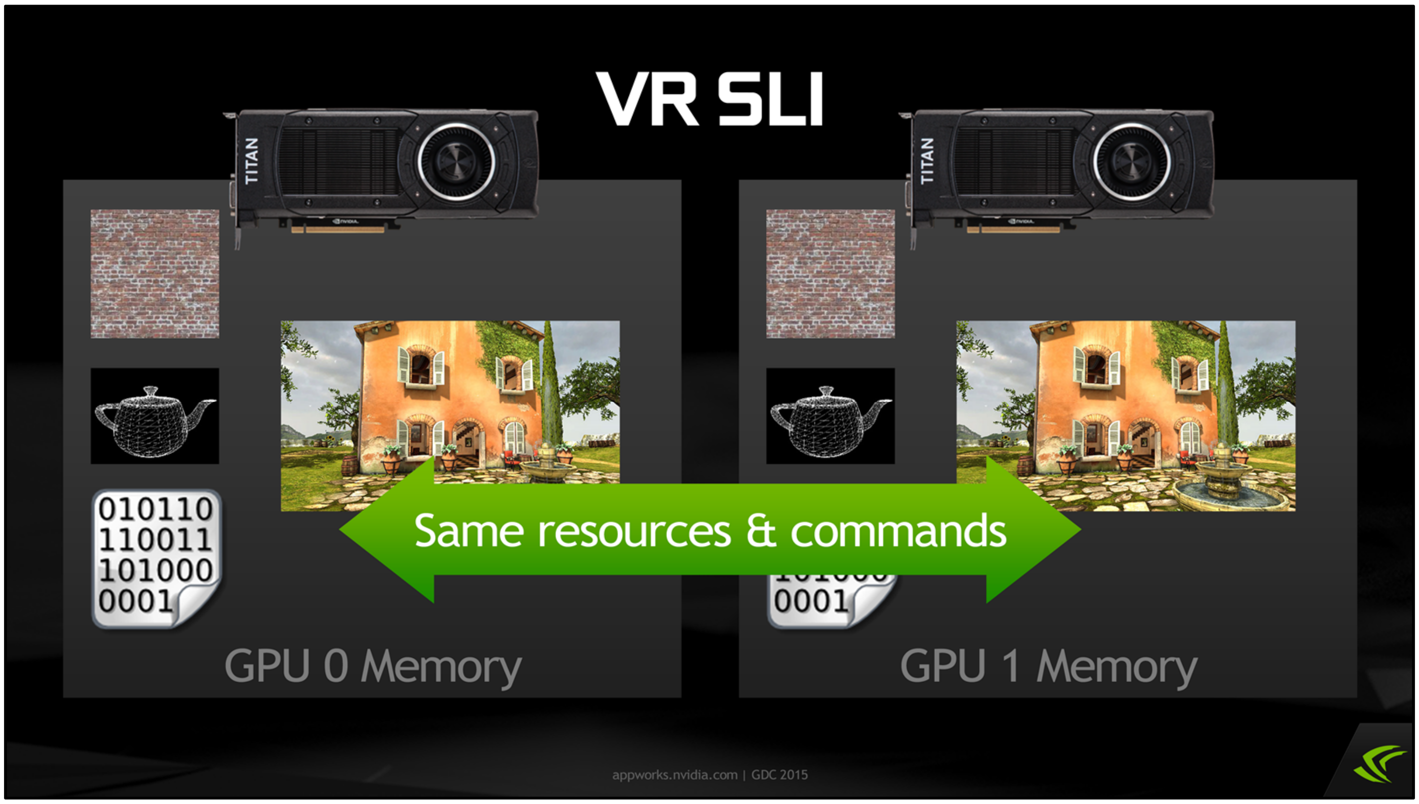
This works well to increase framerate (as long as you don’t have interframe dependencies, anyway), but it doesn’t help with latency. In fact, it might actually hurt latency in some scenarios. So this isn’t the right model for VR.



VR SLI

A better way to use two GPUs for VR rendering is to split the work of drawing a single frame across them — rendering each eye on one GPU. This has the nice property that since the two eyes see essentially the same objects and have essentially the same command stream, the work distribution is nice and even across the two GPUs.

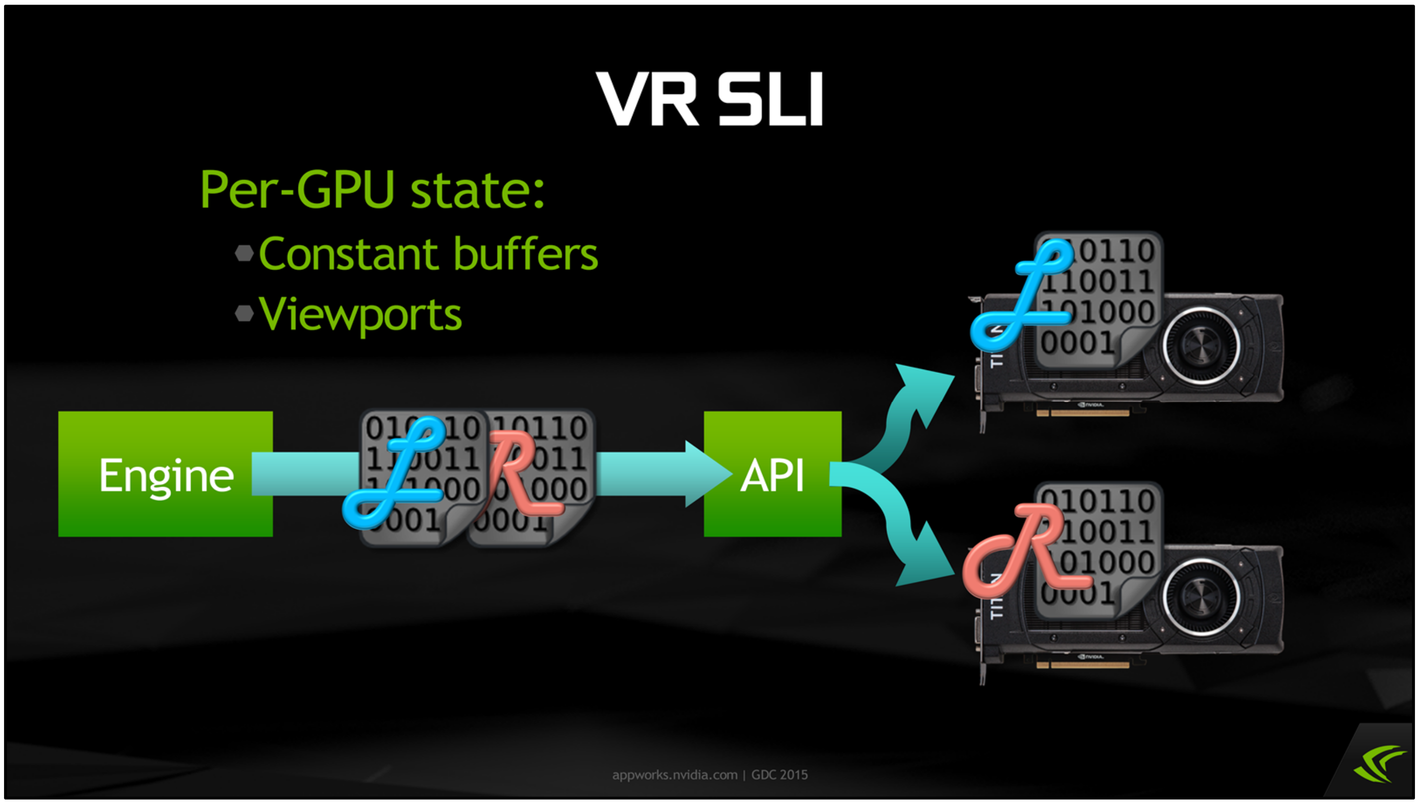
This allows you to improve both framerate *and* latency, since each GPU does less work per frame.



Let's see how VR SLI works in more detail. First of all, keep in mind that in SLI, each GPU has its own copy of every resource. (This is true in standard AFR SLI as well as in VR SLI.) Every time you upload a texture, a model, a constant buffer, or any other resource, it goes to both GPUs — and in fact to the same memory address on both GPUs.

In VR SLI, the driver will also submit the same command stream to both GPUs. Every state setting and draw command your app performs will be executed on both GPUs, using its own copy of every resource referenced.

Because the GPUs have the same input data and execute the same command stream, they generate the same output.



VR SLI

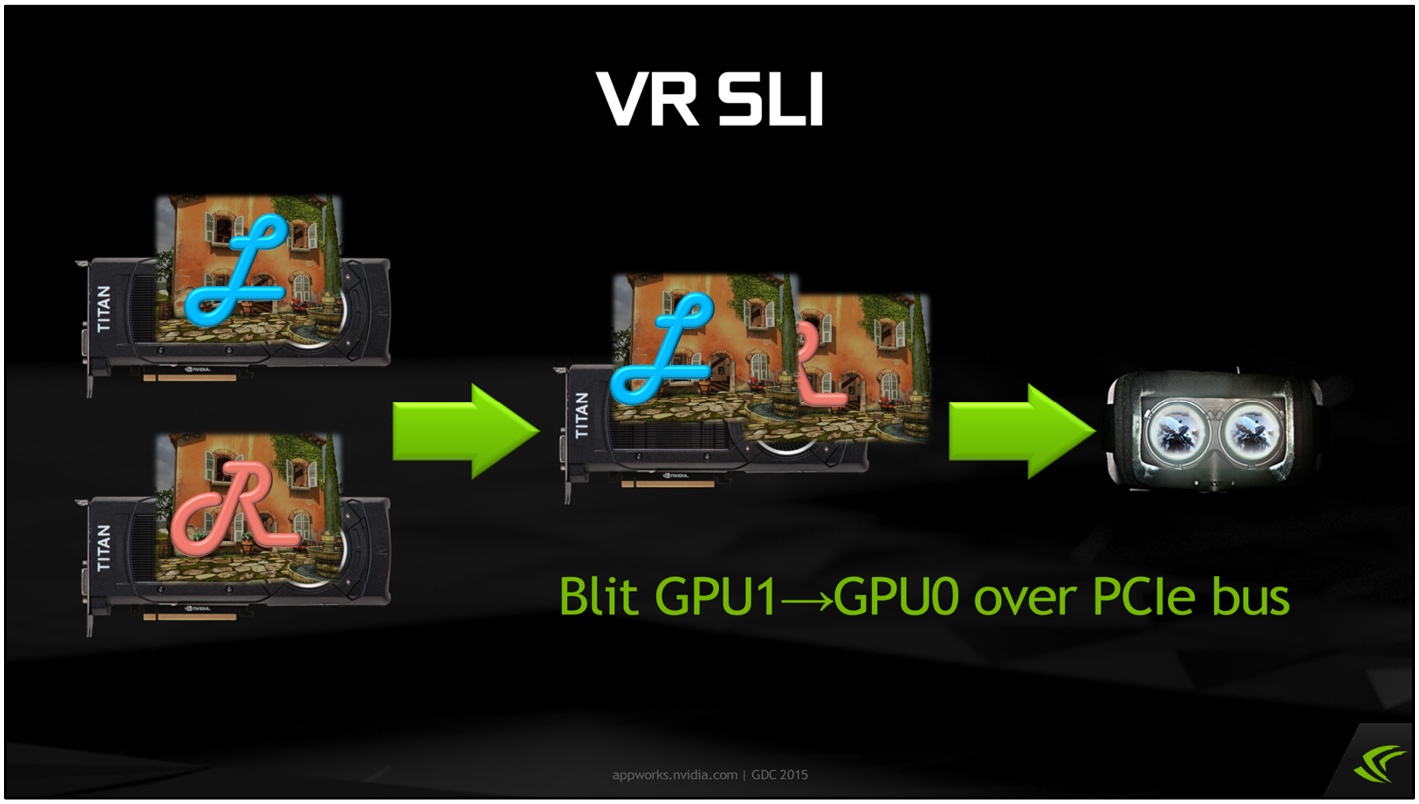
Per-GPU state:

- Constant buffers
- Viewports

Of course, we don't actually want them to generate the same output; we want them to render a stereo pair. So, in our API we have ways to specify a very limited set of state that can differ between the two GPUs: namely, constant buffer bindings and viewports.

So, you can prepare a constant buffer that contains the left eye view matrix, and another buffer with the right eye view matrix. There is a SetConstantBuffers call in the API that takes both the left and right eye constant buffers and sends them to the respective GPUs.

Similarly, you can set up a render target with left and right eye viewports. You don't have to do that — remember, each GPU is rendering to its own copy of the render target, so they're not overwriting each other in any case — but it can make things easier for an engine to set up different viewports per eye.



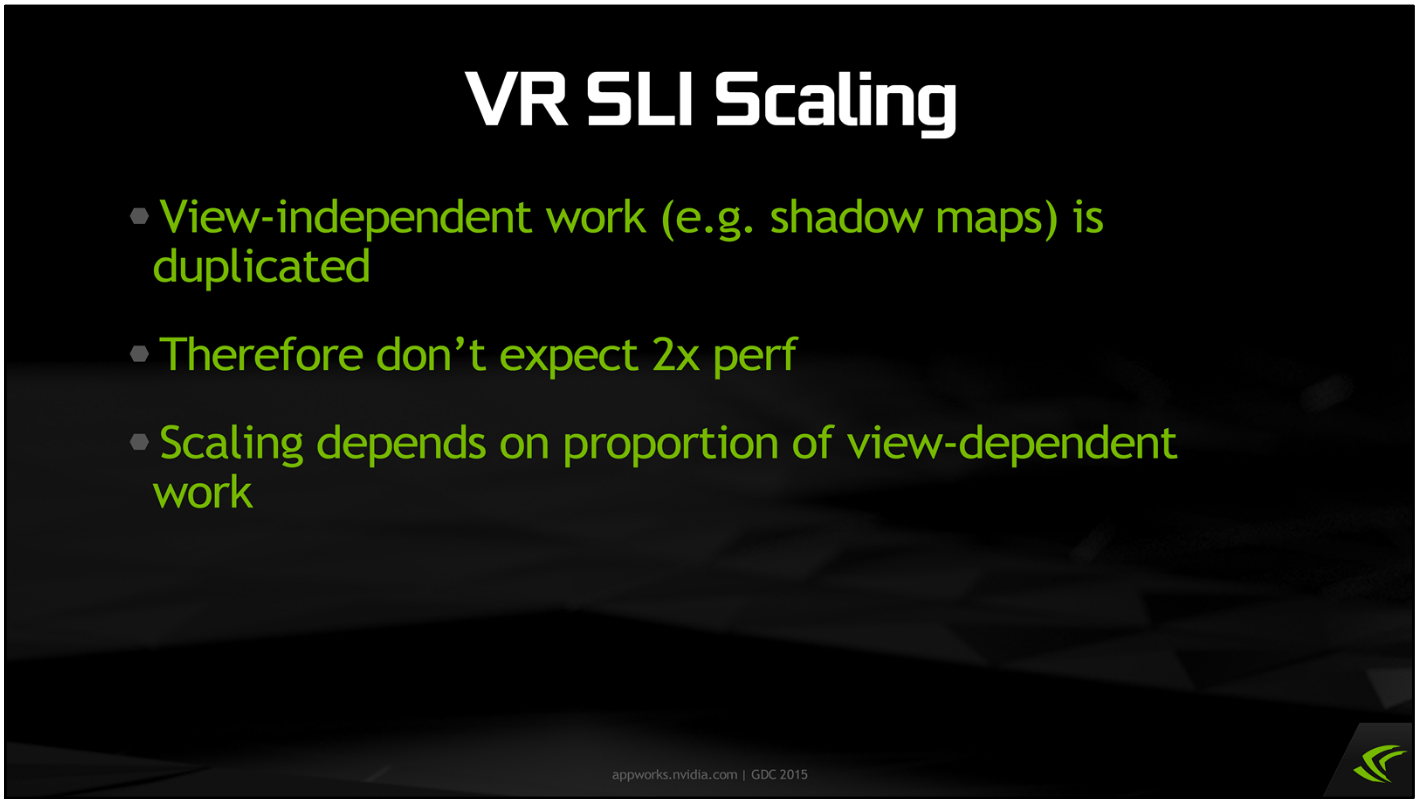
VR SLI

Blit GPU1→GPU0 over PCIe bus

Finally, at some point we have to get both halves of the stereo pair back on a single GPU to present to the display. So we have an API call that synchronizes the GPUs and blits an image from one to the other, over the PCI Express bus.

That way, at the end of the frame we can blit the right eye image back to GPU0 and present both eyes from there.

We can also blit things between GPUs at other points in the frame, if needed — with the proviso that each blit introduces a sync point between the GPUs, which can be harmful for performance.



VR SLI Scaling

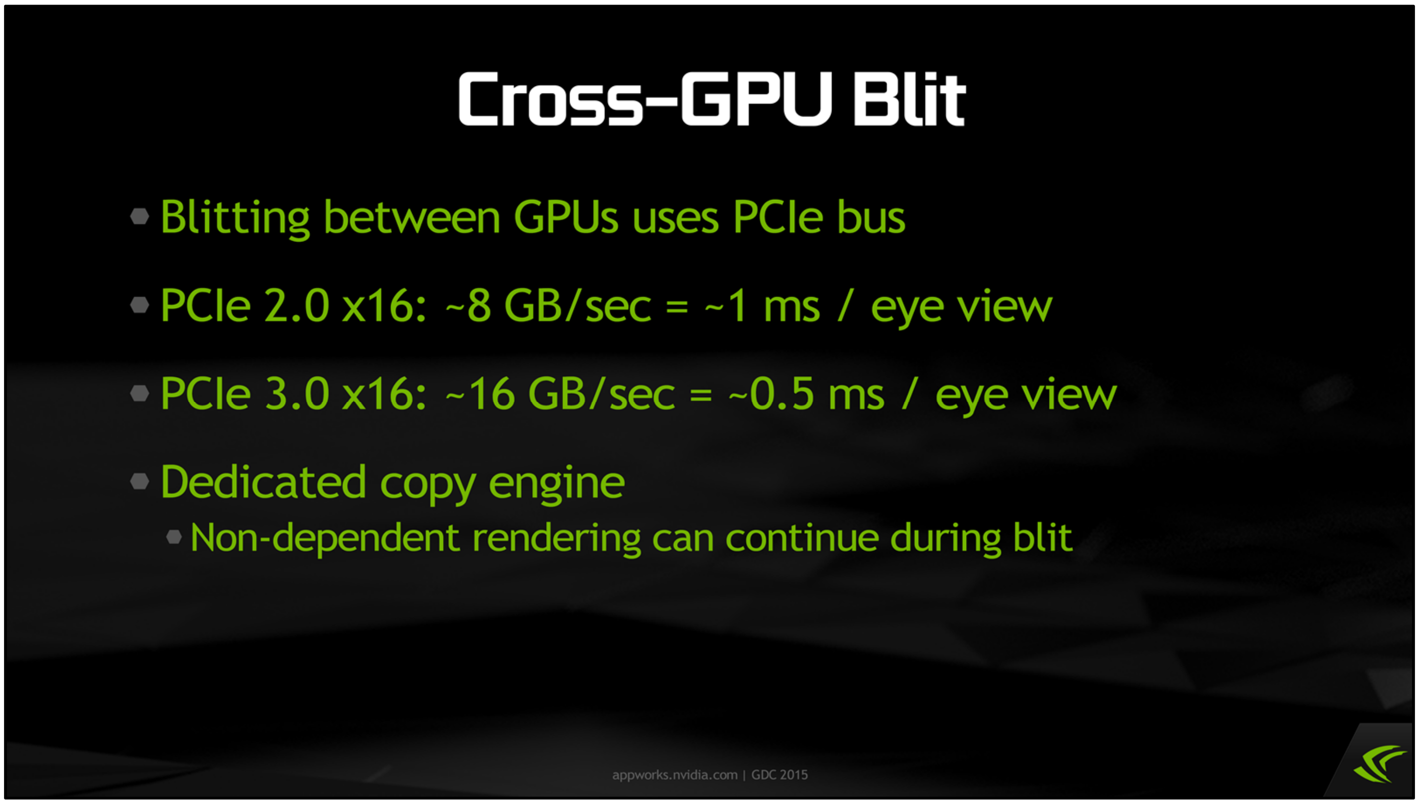
- View-independent work (e.g. shadow maps) is duplicated
- Therefore don't expect 2x perf
- Scaling depends on proportion of view-dependent work

appworks.nvidia.com | GDC 2015

Because of the way VR SLI works, all view-independent work gets duplicated on both GPUs. For example, both GPUs render the shadow maps. Presumably, your shadow maps aren't going to be any different between the left and right eyes, so this is a waste — the two GPUs will be generating exactly the same shadow map results. We're trading this inefficiency in exchange for lower latency.

Therefore we can't expect to get perfect 2x scaling. The scaling we see will depend on what proportion of the frame is taken up by view-dependent work, such as the actual eye views, versus view-independent work such as physics simulations and shadow maps. I would argue that if you see 40% to 50% scaling with VR SLI, you're doing pretty well. However, if you have much less view-independent work you would get closer to 2x scaling.

You could also consider splitting up the physics sims, shadow maps and so on and trying to do half of the work on each GPU, then syncing and blitting the results between GPUs before going on. This can work, but it's tricky to ensure an even work distribution, and as mentioned, the sync points can potentially hurt performance.



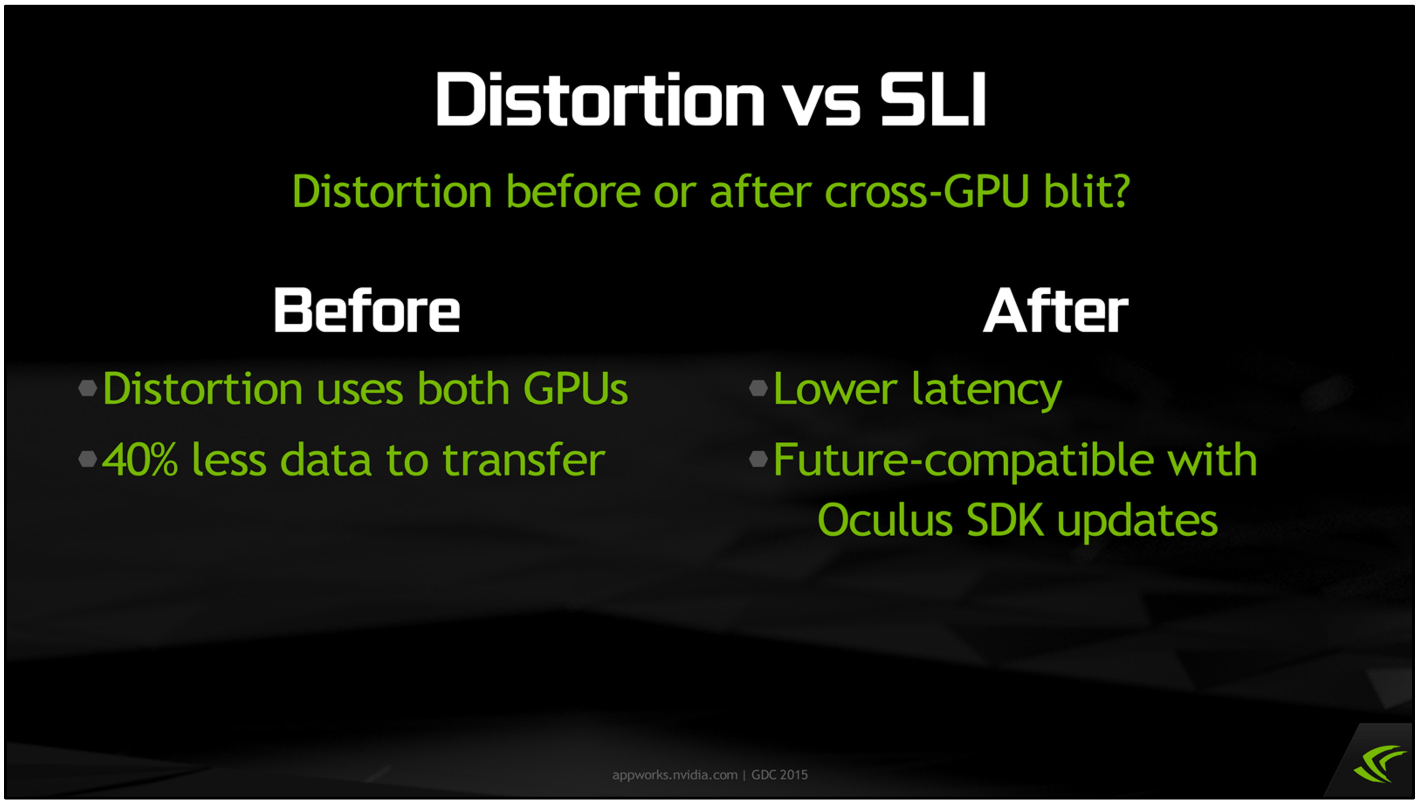
Cross-GPU Blit

- Blitting between GPUs uses PCIe bus
- PCIe 2.0 x16: ~8 GB/sec = ~1 ms / eye view
- PCIe 3.0 x16: ~16 GB/sec = ~0.5 ms / eye view
- Dedicated copy engine
  - Non-dependent rendering can continue during blit

appworks.nvidia.com | GDC 2015

When you do blit between GPUs, it goes across the PCI Express bus, so just be aware of what kind of speed you can expect from that. With PCIe 2.0 x16, you get about 8 GB/sec of bandwidth. A single eye view for an Oculus headset is about 2 megapixels, so will take about 1 ms to transfer (at 32 bits per pixel). With PCIe 3.0 you'll get double the bandwidth, so half the time. This is something that needs to be kept in mind when planning a rendering pipeline: the transfer time will eat up some of the benefit you got by distributing rendering across GPUs.

One piece of good news here is that the blits are done using a dedicated copy engine. This is an engine on the GPU that just shoves bits from one GPU's memory into the other, without consuming any computational resources. So, other rendering work can continue during the blit, as long as it's not reading or writing the same render targets that are being blitted.



Distortion vs SLI

Distortion before or after cross-GPU blit?

Before

- Distortion uses both GPUs
- 40% less data to transfer

After

- Lower latency
- Future-compatible with Oculus SDK updates

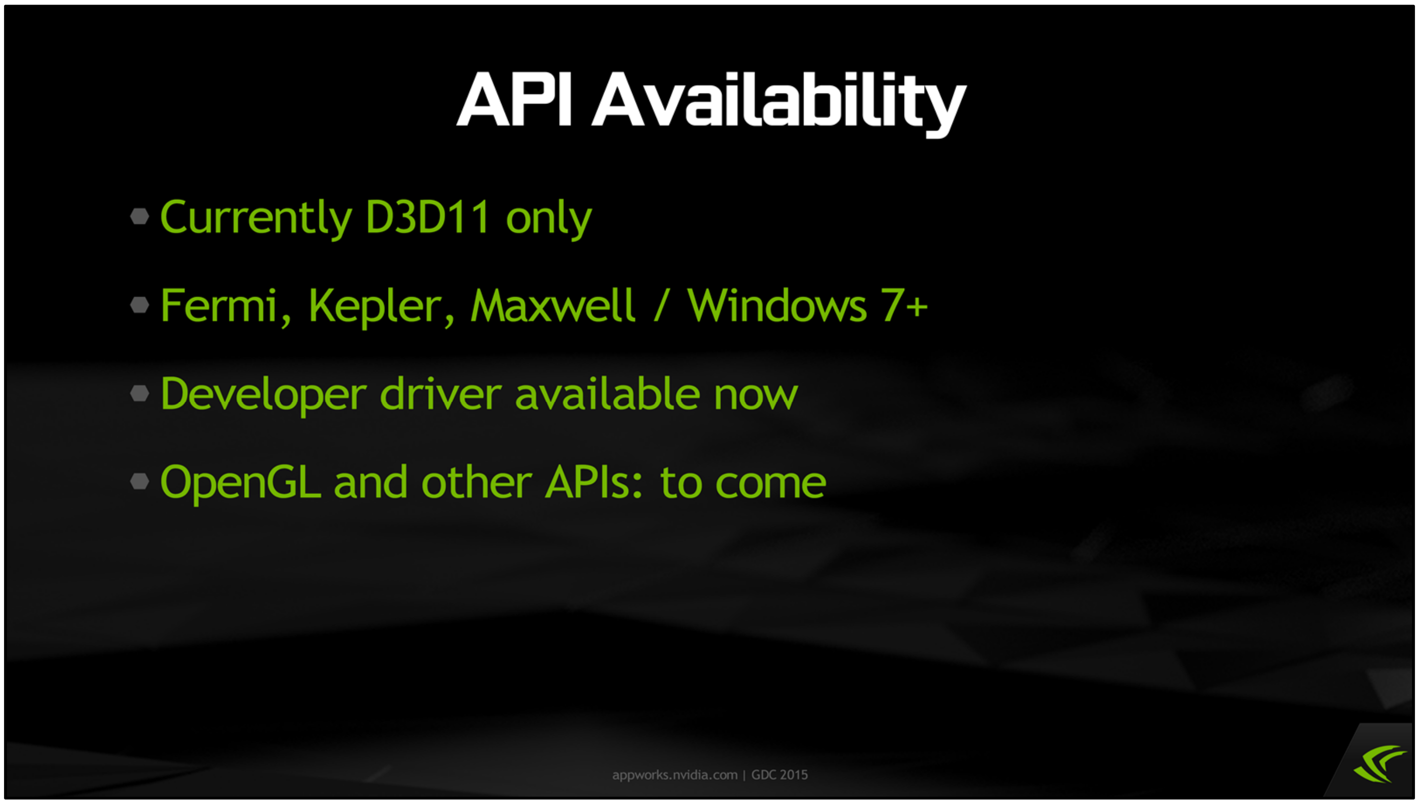
appworks.nvidia.com | GDC 2015

In light of the fact that cross-GPU blits are somewhat slow, an interesting question comes up regarding the distortion pass. We know that at the end of the frame we have to blit the right eye view to GPU0 so it can be presented on the display. The question is, should we do this blit before or after distortion? In other words, should we have each GPU perform distortion for one eye and then blit across the distorted image? Or should we first blit the undistorted image, then run the distortion for both eyes on GPU0?

There are arguments to be made both ways. If you do the distortion before the blit, the distortion gets to take advantage of both GPUs. Also, there will be less data to transfer overall, because the post-distortion render target is smaller, by about 40% of the pixel count, than the pre-distortion render target. On the other hand, you've introduced the extra latency of the sync and blit between when the distortion pass runs and when the image gets presented on the display.

If you do the distortion after the blit, you don't have that extra latency. There's also the benefit that you're just using the standard Oculus SDK distortion, which means you'll be compatible with any future changes to the distortion pass that Oculus makes. As opposed to the per-eye distortion, where you'd have to copy the Oculus distortion code out of their SDK and modify it to run per-eye.

Still, it's not entirely clear which approach might be the better one in general.



API Availability

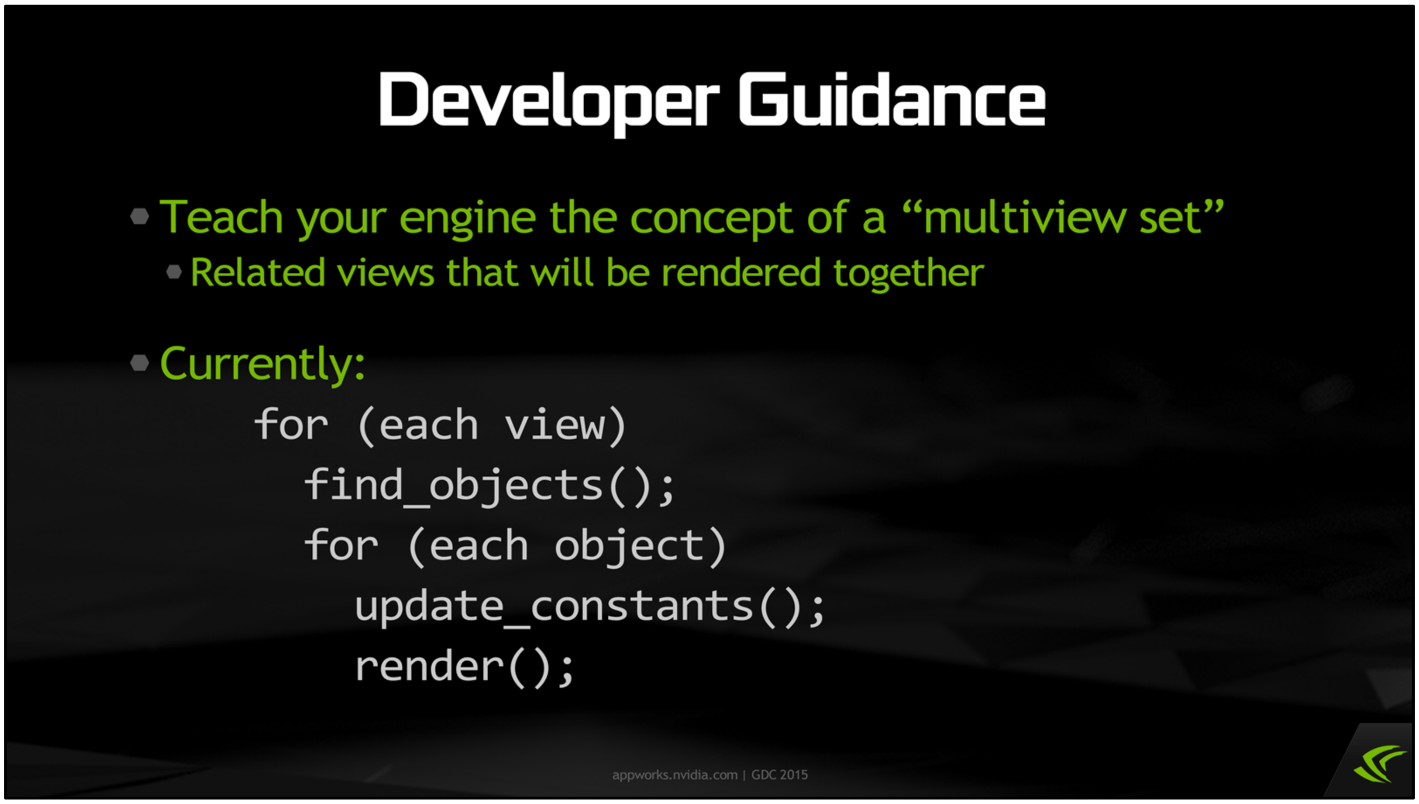
- Currently D3D11 only
- Fermi, Kepler, Maxwell / Windows 7+
- Developer driver available now
- OpenGL and other APIs: to come

appworks.nvidia.com | GDC 2015

VR SLI is currently implemented for D3D11 only. It works on Fermi, Kepler, and Maxwell GPUs, on Windows 7 and up.

We have an NDA developer driver with VR SLI support available now, so if you're interested in looking at this, get in touch with us and we'll set you up with a driver.

We're also working on support for VR SLI, and multiview rendering more generally, in OpenGL and other APIs — but that's still down the road a bit.



Developer Guidance

- Teach your engine the concept of a "multiview set"
  - Related views that will be rendered together
- Currently:

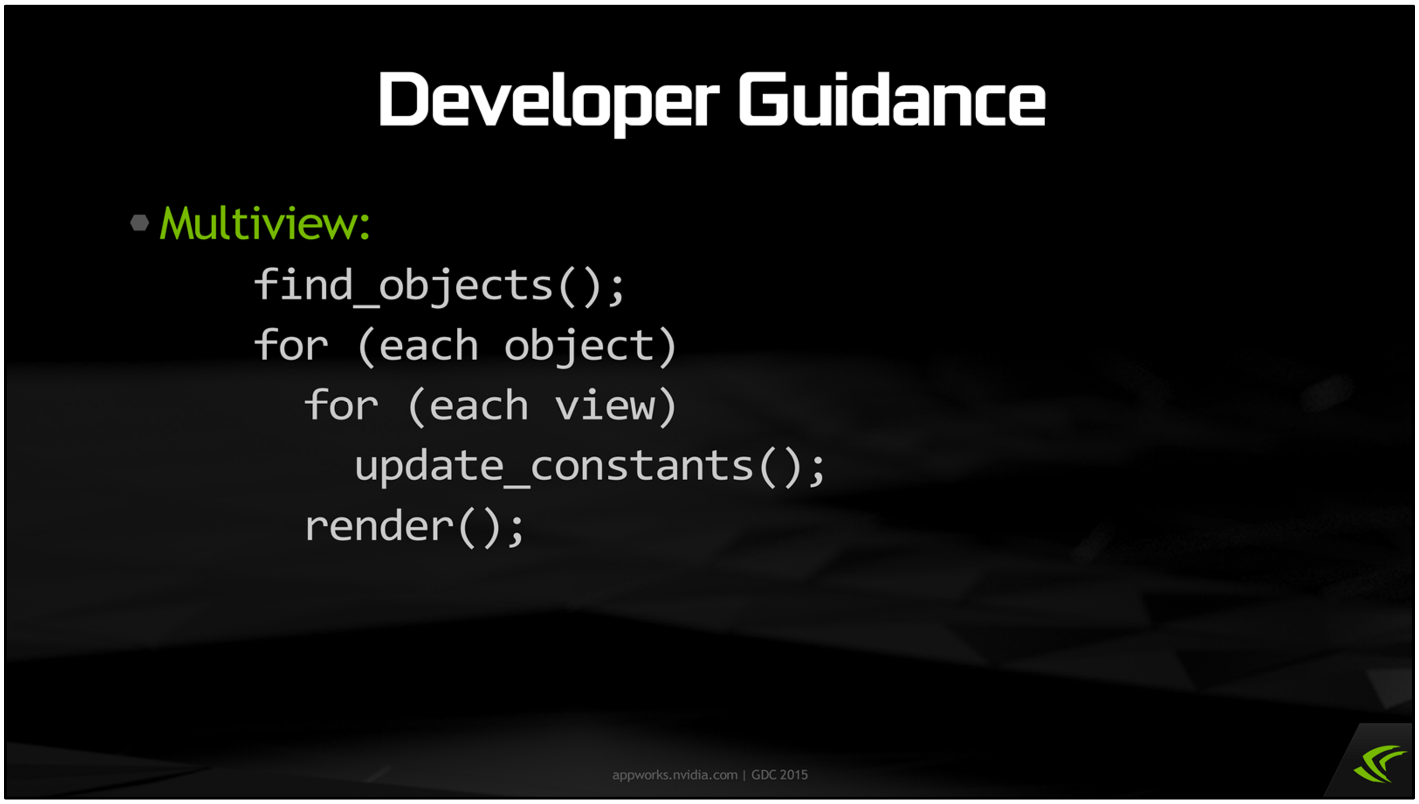
```
for (each view)
  find_objects();
  for (each object)
    update_constants();
    render();
```

appworks.nvidia.com | GDC 2015

We've talked a lot about the nuts and bolts of how multiview rendering and VR SLI operate, but as a game developer, what should you take away from this?

The most important thing is that you don't want to process individual views one at a time anymore. Instead, you want to teach your engine the concept of what you might call a multiview set — a set of related views that will be rendered together, such as the two eyes of a stereo pair, or the six faces of a cubemap.

Currently, most engines are doing something like this pseudocode snippet here at a high level. They're iterating over views as an outer loop. For each view, they build a list of visible objects, then they calculate the appropriate constants for those objects and generate the rendering commands for them.



Developer Guidance

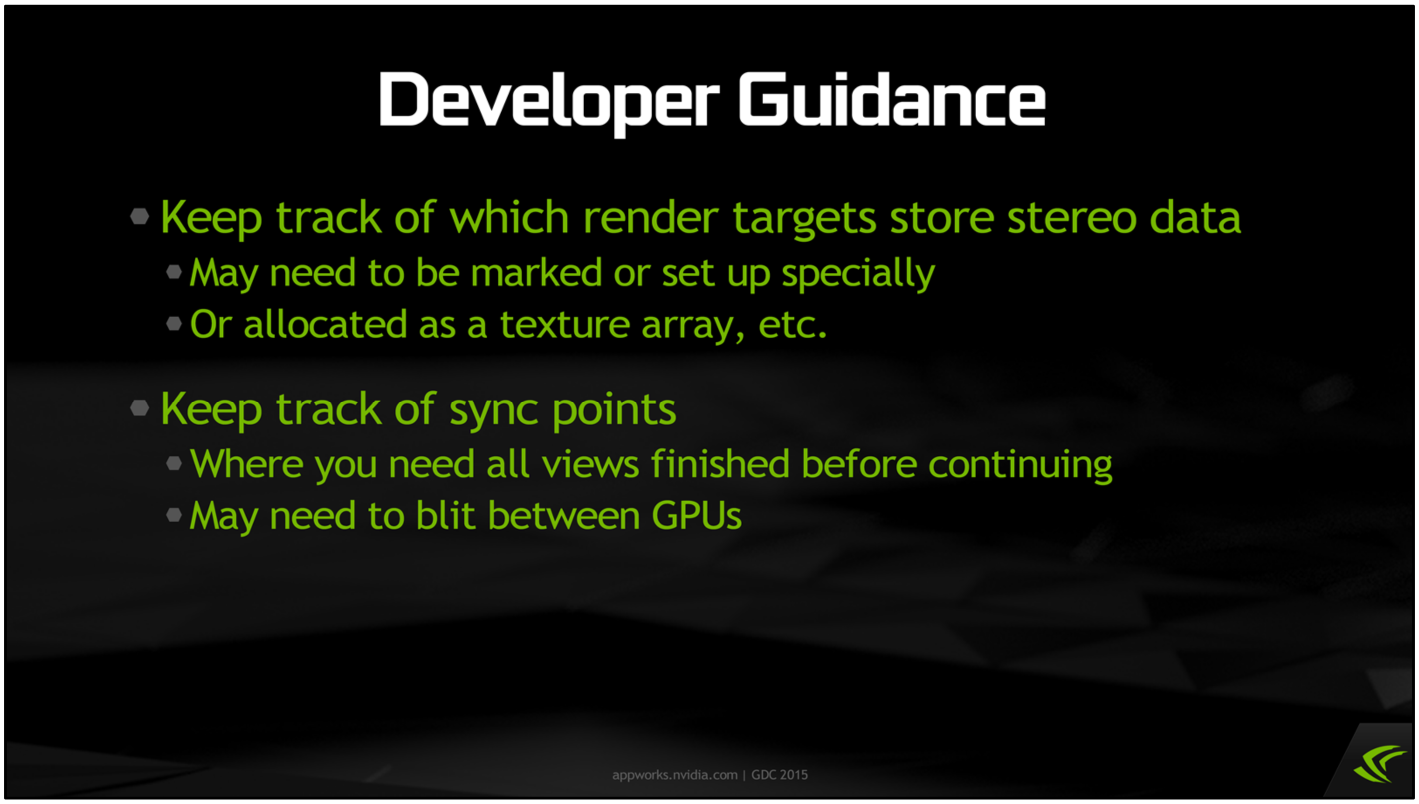
- Multiview:

```
find_objects();
for (each object)
  for (each view)
    update_constants();
  render();
```

appworks.nvidia.com | GDC 2015

In multiview, what you want is a little bit different. First, you want to build a list of visible objects that is the union of all the objects visible from all the views in the set. Then, when you calculate the constants for an object, you build a set of constants for all views in the set. For example, in stereo you might build a left eye and right eye matrix for the same object. Those per-view constants should be stored together, in the same data structure, so they can be submitted to the multiview API. Finally, you'll submit the rendering commands just once for the whole set of views.

So, with a multiview API, rendering a stereo pair of views is really almost like rendering a single view. All you have to do differently is calculate and submit a few extra matrices in your constant buffers, to set up the appropriate transforms for the left and right eye views.

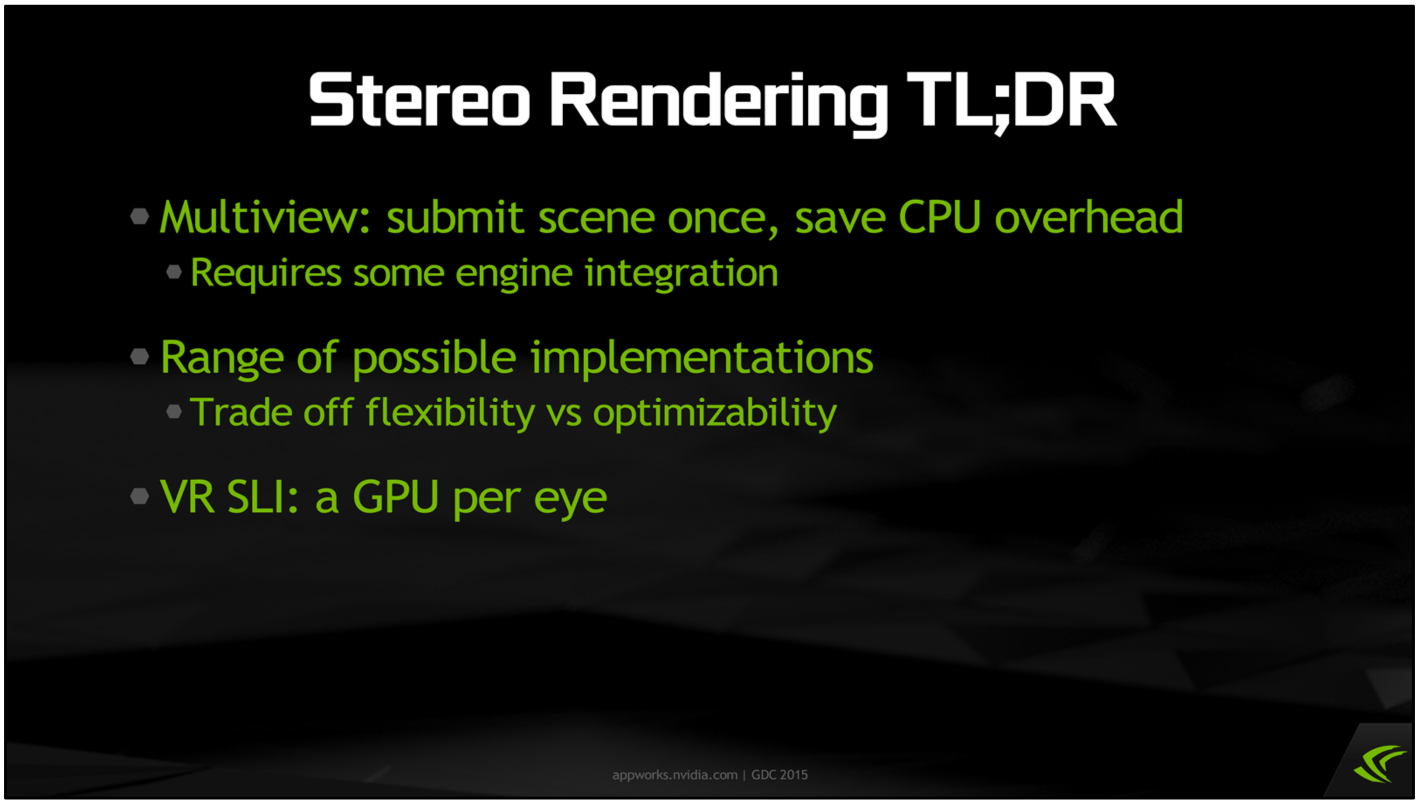


Developer Guidance

- Keep track of which render targets store stereo data
  - May need to be marked or set up specially
  - Or allocated as a texture array, etc.
- Keep track of sync points
  - Where you need all views finished before continuing
  - May need to blit between GPUs

Just a couple of other notes. If you're preparing your engine for multiview APIs, you're going to want to keep track of which render targets are intended to store stereo data. G-buffers and scene color buffers would be in stereo, while things like shadow maps and physics simulation results would not be. Depending on the details of the multiview API, you may need to configure stereo render targets differently, or allocate them as slices of a texture array, or something like that.

You'll also want to keep track of sync points between the views — places where you need all the views' rendering to be finished before continuing. In VR SLI, such places may need to sync the GPUs and blit data between them, so it's a good idea to keep track of where those places are.



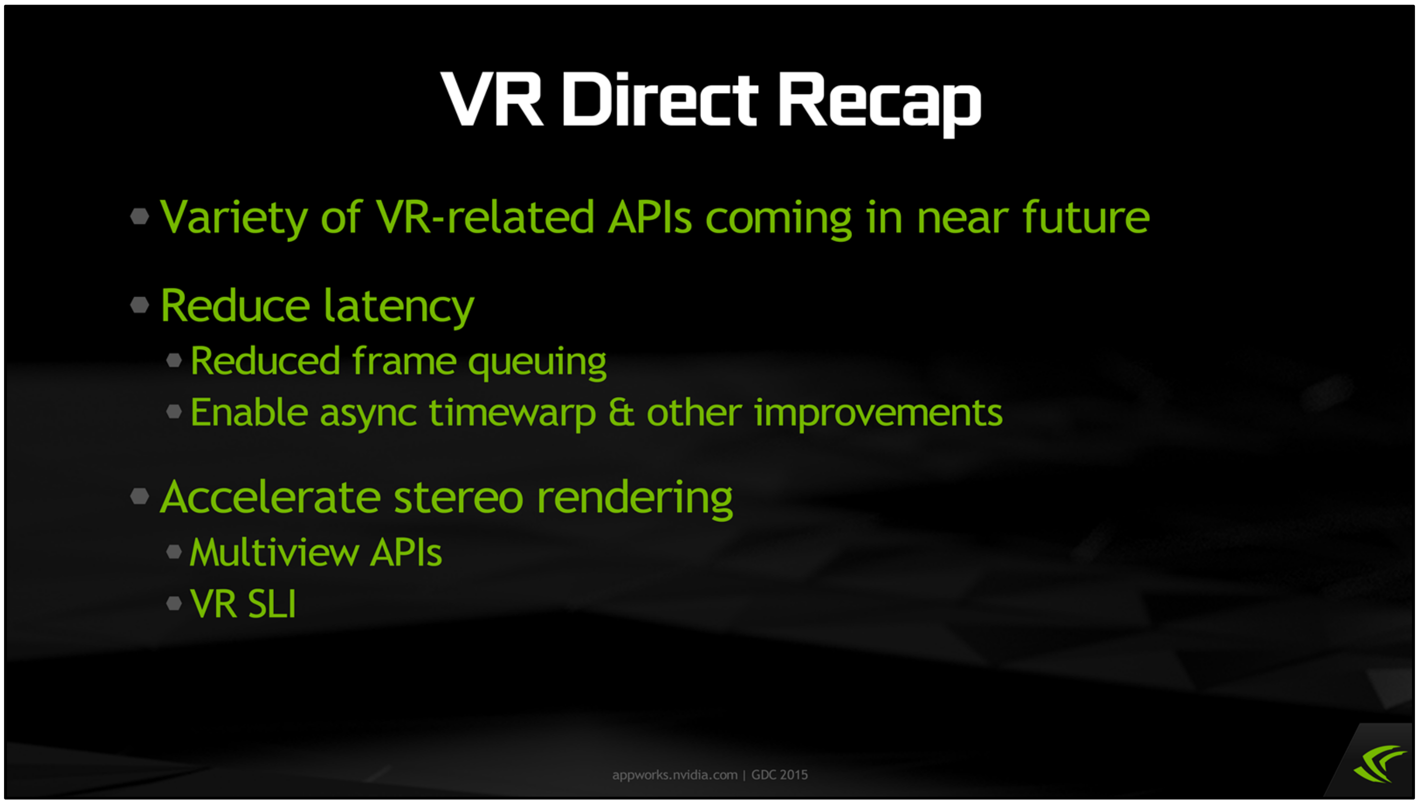
Stereo Rendering TL;DR

- Multiview: submit scene once, save CPU overhead
  - Requires some engine integration
- Range of possible implementations
  - Trade off flexibility vs optimizability
- VR SLI: a GPU per eye

That’s about all we have to say on stereo rendering today. To review, we’ve explained the concept of multiview rendering, in which you submit a single stream of rendering commands that contains all the information necessary to render several different views. Having a single command stream lets you save CPU work, both in the engine and in the driver, versus completely independent command streams.

There are a range of possible implementations of multiview rendering, and some of them enable you to save some portion of GPU work by sharing some stages of the rendering pipeline across views. There is a trade-off between flexibility and optimizability here, and no one is quite sure yet exactly what the right point is to hit on that spectrum.

Finally, we discussed VR SLI, a special case of multiview that’s implemented today. VR SLI works by broadcasting a command stream to two GPUs, letting you render each eye on its own GPU and accomplishing stereo rendering with hardly any more overhead than single-view rendering.

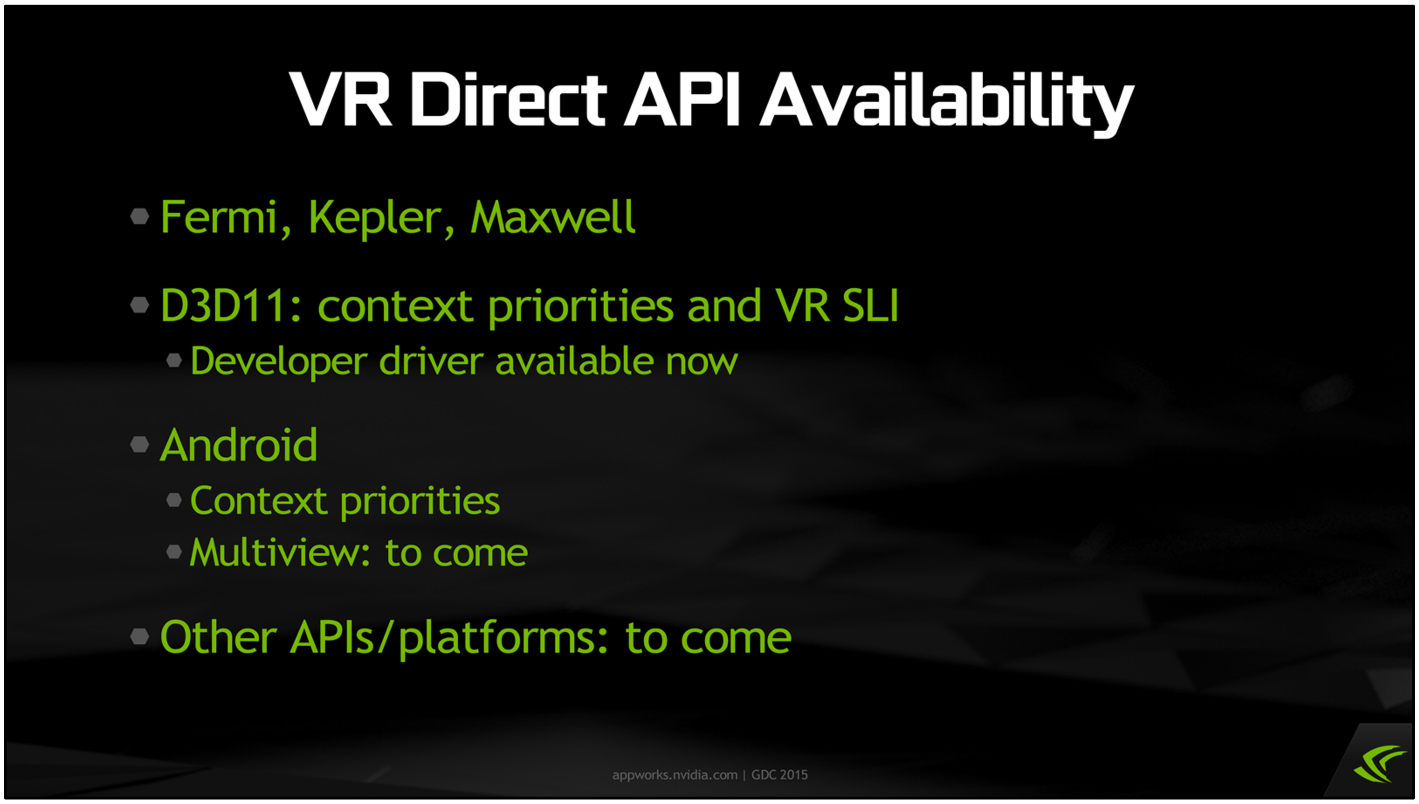


VR Direct Recap

- Variety of VR-related APIs coming in near future
- Reduce latency
  - Reduced frame queuing
  - Enable async timewarp & other improvements
- Accelerate stereo rendering
  - Multiview APIs
  - VR SLI

appworks.nvidia.com | GDC 2015

And that pretty much concludes the talk. We've discussed two major components of VR Direct: asynchronous timewarp and VR SLI, for which we have APIs working today. Again, VR Direct is a collection of a variety of technologies, all aimed at reducing latency and accelerating stereo rendering, so you'll be hearing about it from NVIDIA again in the future.



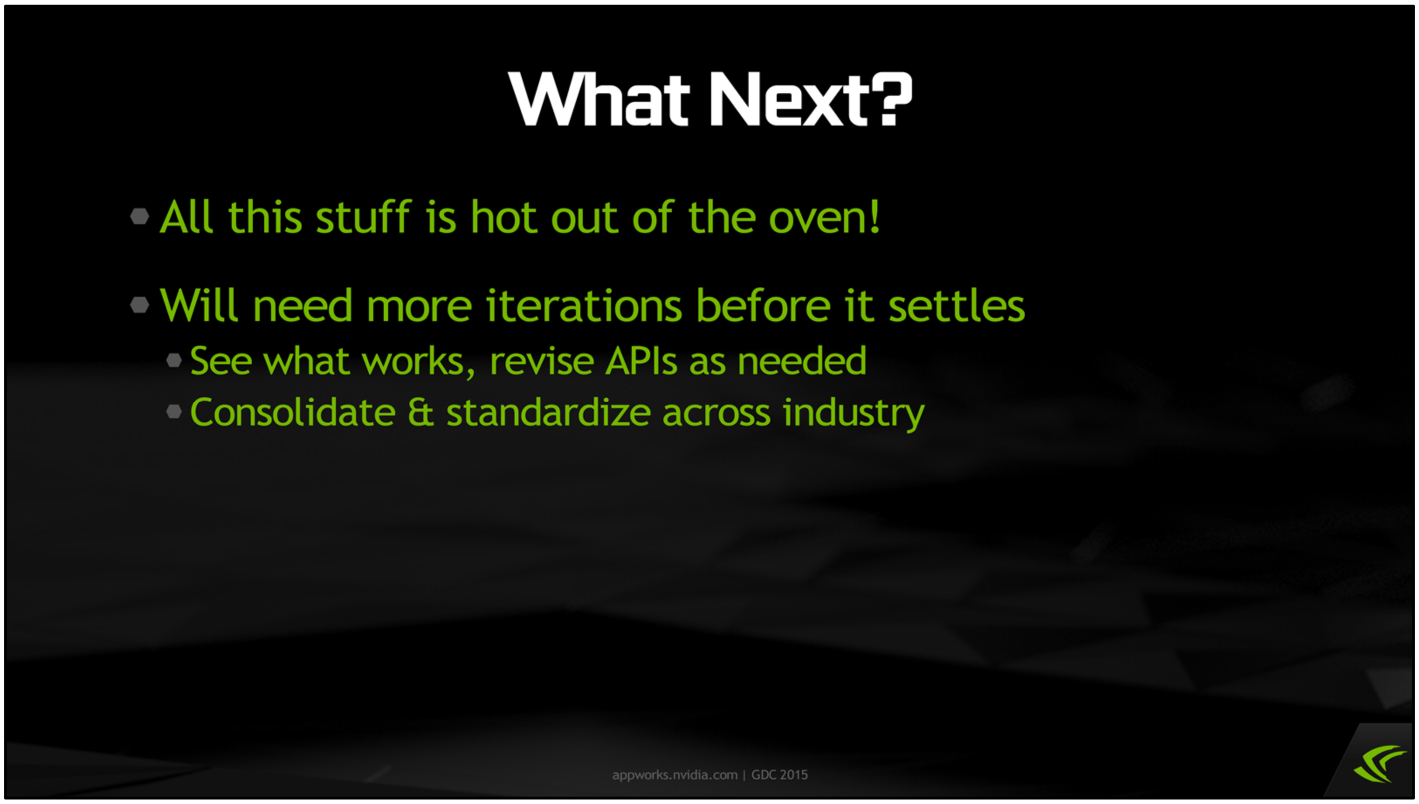
VR Direct API Availability

- Fermi, Kepler, Maxwell
- D3D11: context priorities and VR SLI
  - Developer driver available now
- Android
  - Context priorities
  - Multiview: to come
- Other APIs/platforms: to come

appworks.nvidia.com | GDC 2015

Again, the context priorities and VR SLI APIs we talked about today are available now in an NDA developer driver, so if you're really interested in working with these APIs, come talk to us, and we'll get you set up with a driver.

Both context priorities and VR SLI on Fermi, Kepler, and Maxwell hardware. The APIs are currently implemented for D3D11, and in addition, we have support for context priorities on Android already. In the future we will gradually fill out the rest of the API/platform matrix — such as desktop GL, and eventually new APIs such as D3D12 and Vulkan.



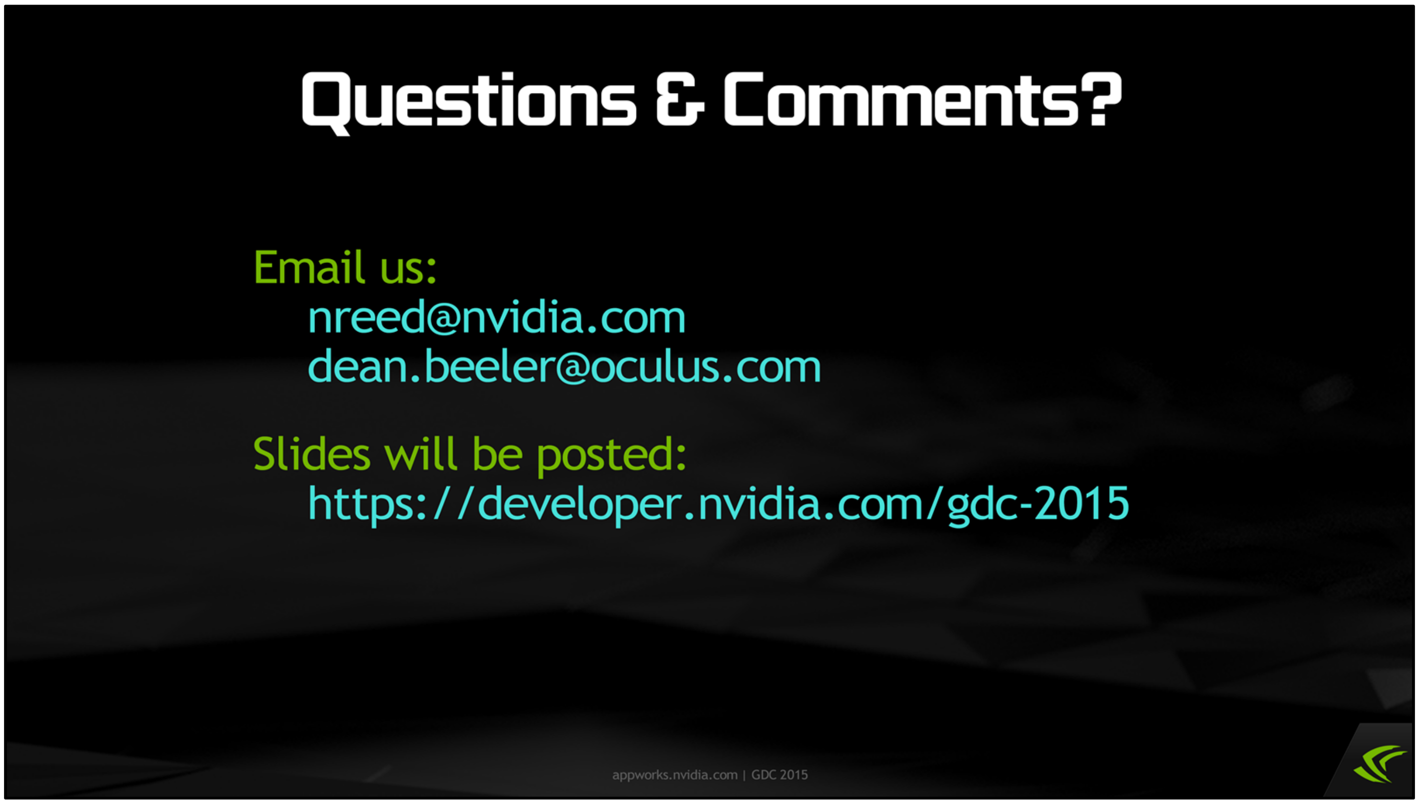
What Next?

- All this stuff is hot out of the oven!
- Will need more iterations before it settles
  - See what works, revise APIs as needed
  - Consolidate & standardize across industry

appworks.nvidia.com | GDC 2015

Before we conclude, I'd just like to make a quick note. All of the things we talked about today are hot out of the oven. Some of the APIs that I'm talking about here literally did not exist six weeks ago. They should be looked at as rough drafts, not finished products.

We are going to need more iterations before all this stuff settles out. As an industry, we will need to try different things out, see what works and what doesn't, and revise our APIs as needed. Eventually we should consolidate and standardize on some common APIs and feature sets for things like multiview rendering — but we're not there yet; we're still experimenting and learning. So things will be changing rapidly for awhile!



Questions & Comments?

Email us:
nreed@nvidia.com
dean.beeler@oculus.com

Slides will be posted:
https://developer.nvidia.com/gdc-2015

And that concludes our talk. We'll have some time for questions and comments now, and you can email us as well. The slides will be posted on NVIDIA's developer website. Thank you for listening.