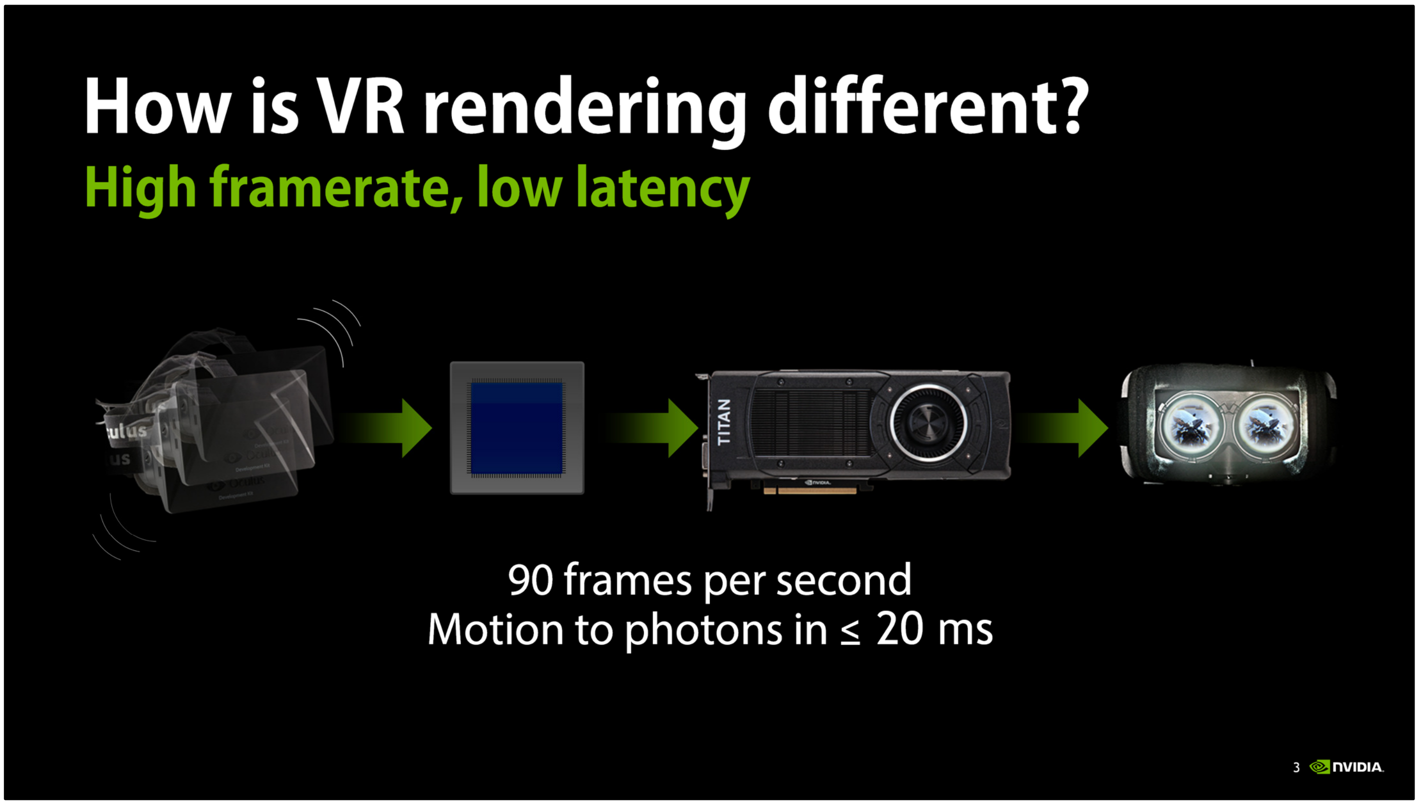
GameWorks VR

Nathan Reed, VR Software Engineer

How is VR rendering different?

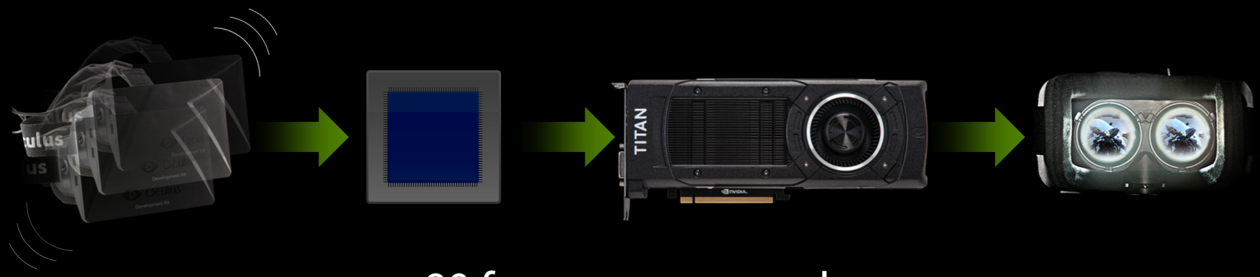
2

To set the stage, first I want to mention a few ways that virtual reality rendering differs from the more familiar kind of GPU rendering that real-time 3D apps and games have been doing up to now.



How is VR rendering different?

High framerate, low latency



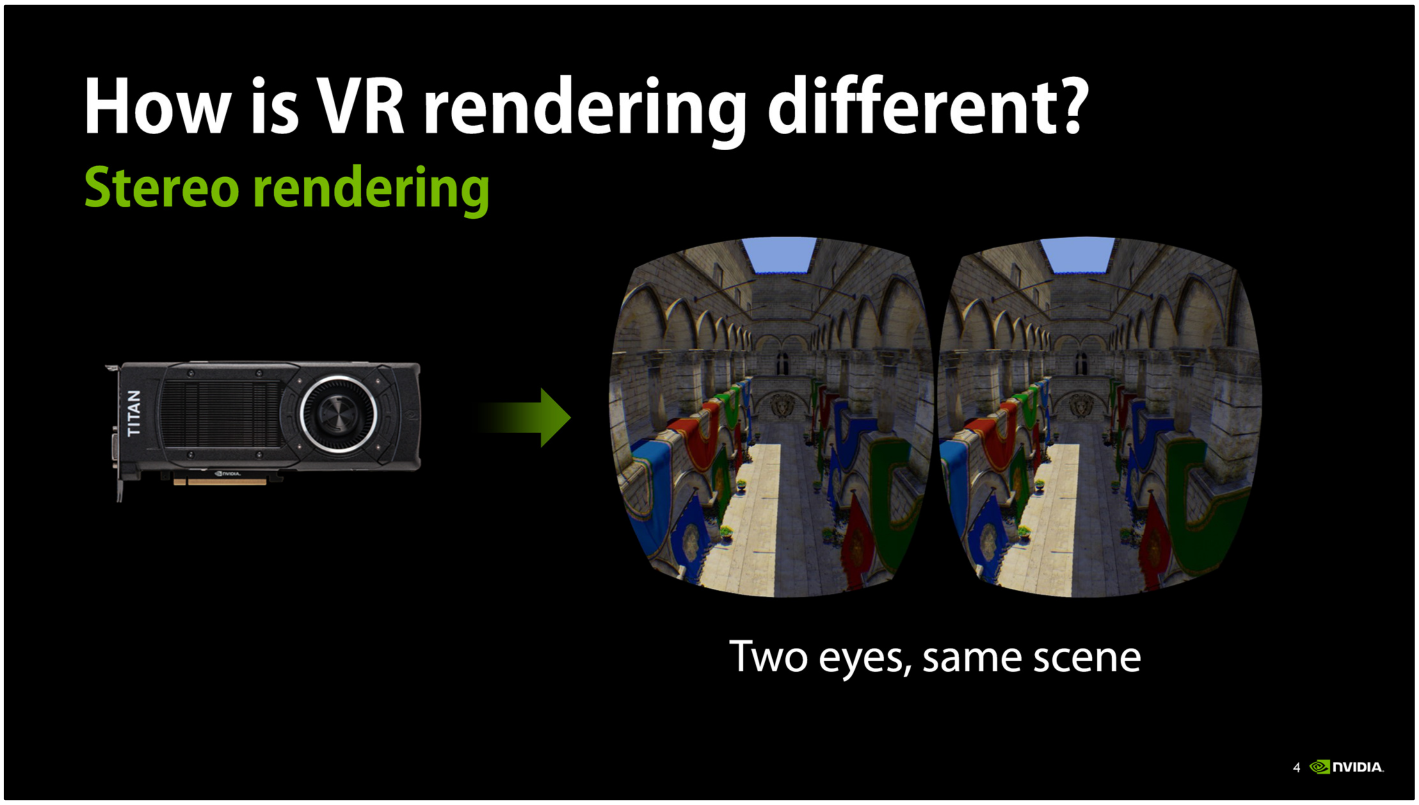
90 frames per second
Motion to photons in ≤ 20 ms

3

First, virtual reality is extremely demanding with respect to rendering performance. Both the Oculus Rift and HTC Vive headsets require 90 frames per second, which is much higher than the 60 fps that's usually considered the gold standard for real-time rendering.

We also need to hit this framerate while maintaining low latency between head motion and display updates. Research indicates that the total motion-to-photons latency should be at most 20 milliseconds to ensure that the experience is comfortable for players. This isn't trivial to achieve, because we have a long pipeline, where input has to be first processed by the CPU, then a new frame has to be submitted to the GPU and rendered, then finally scanned out to the display.

Traditional real-time rendering pipelines have not been optimized to minimize latency, so this goal requires us to change our mindset a little bit.



How is VR rendering different?

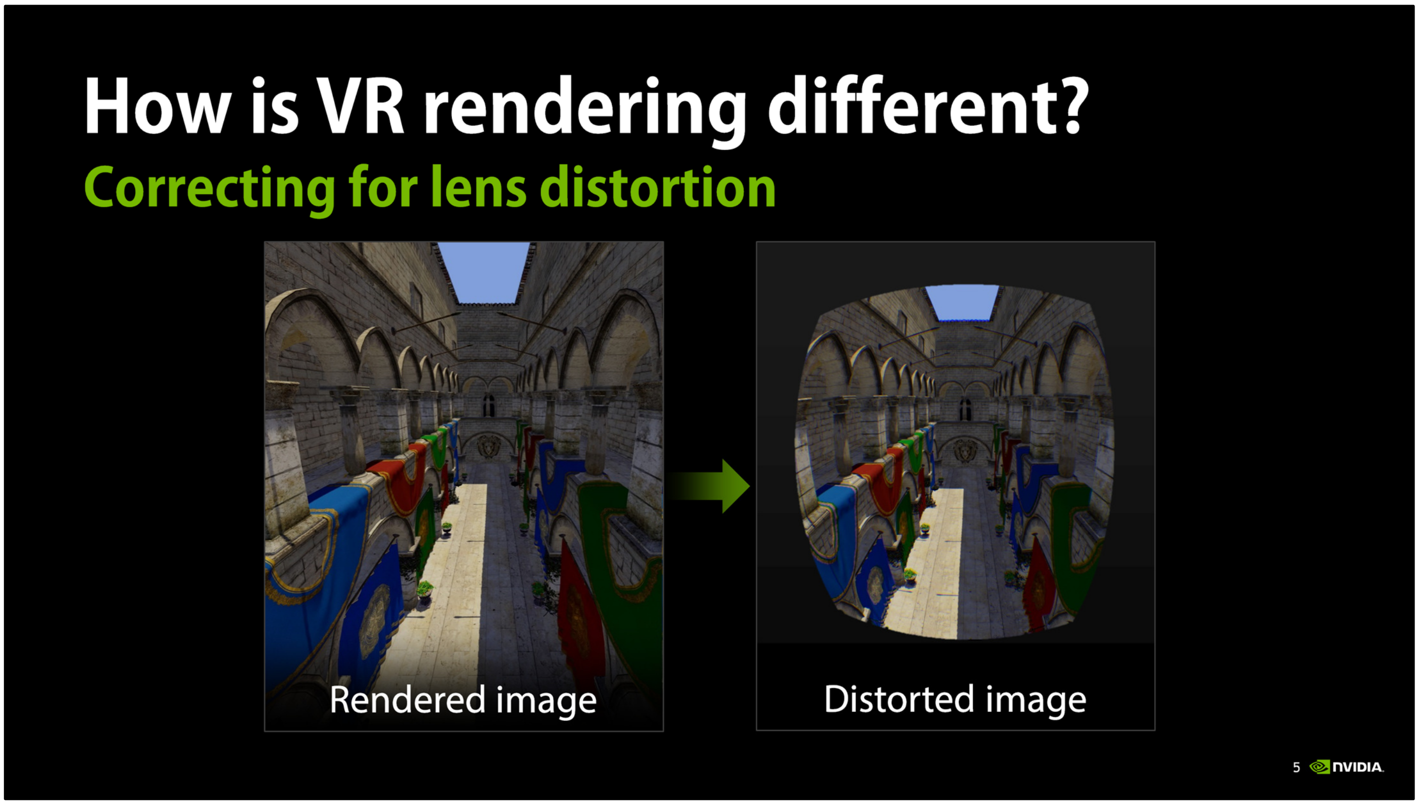
Stereo rendering

Two eyes, same scene

4

Another 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.



How is VR rendering different?

Correcting for lens distortion

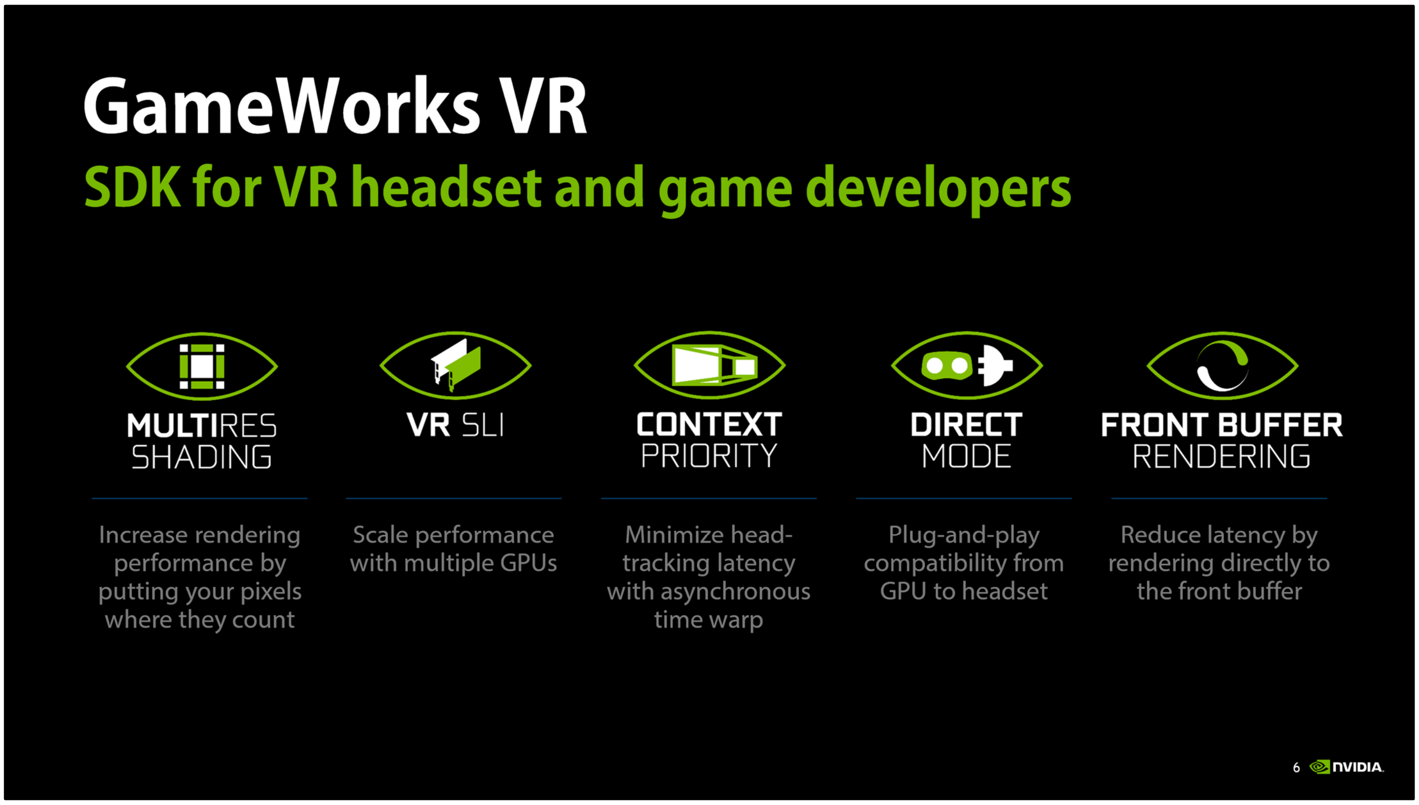
Rendered image

Distorted image

5

Finally, another unusual feature of rendering for a VR headset is that the image we present has to be barrel-distorted to counteract the optical effects of the lenses.

The trouble is that GPUs can’t natively render into a nonlinearly distorted view like this. Current VR software solves this problem by first rendering a normal perspective projection (left), then resampling to the distorted view (right) as a postprocess.



GameWorks VR

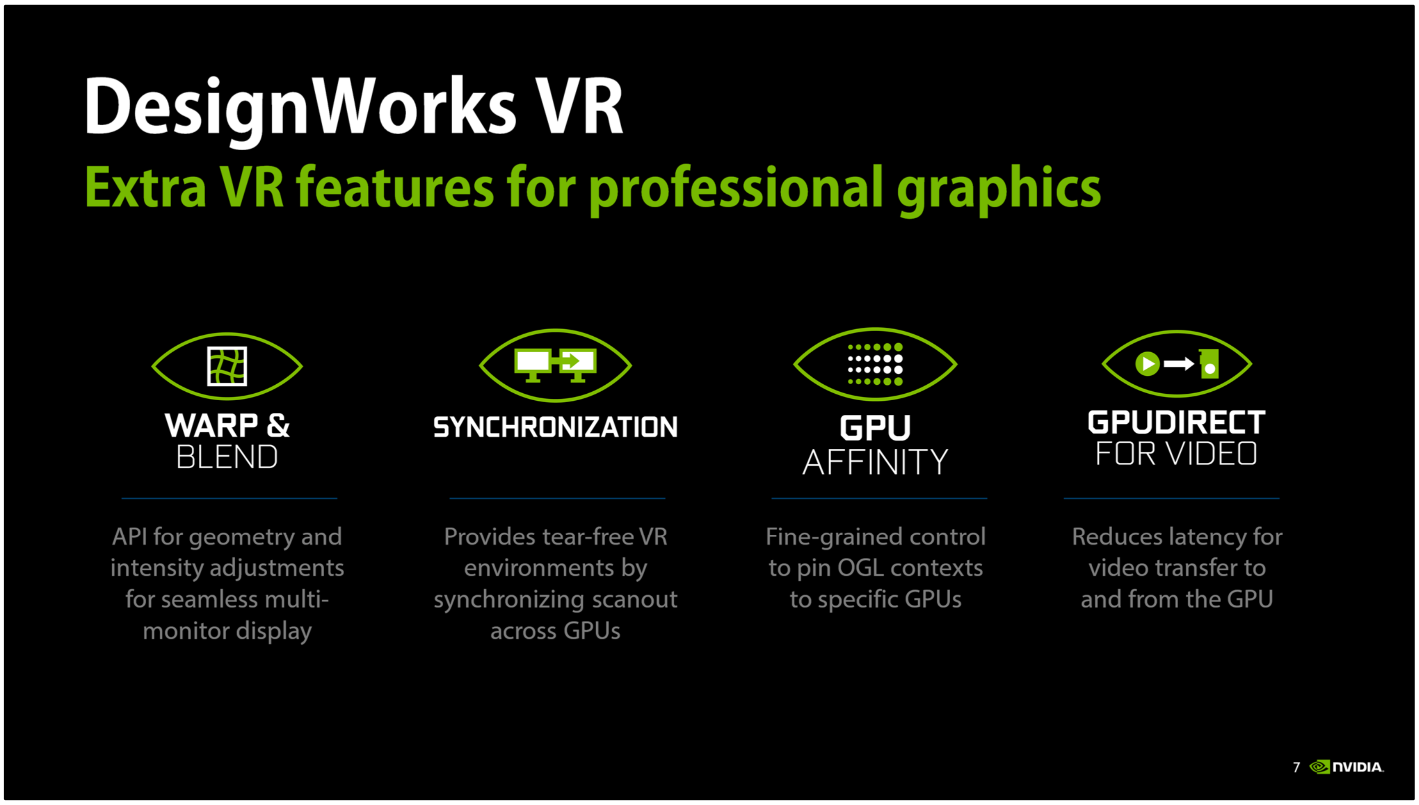
SDK for VR headset and game developers

MULTIRES SHADING	VR SLI	CONTEXT PRIORITY	DIRECT MODE	FRONT BUFFER RENDERING
Increase rendering performance by putting your pixels where they count	Scale performance with multiple GPUs	Minimize head-tracking latency with asynchronous time warp	Plug-and-play compatibility from GPU to headset	Reduce latency by rendering directly to the front buffer

6 NVIDIA

As a GPU company, of course NVIDIA is going to do all we can to help VR game and headset developers use our GPUs to create the best VR experiences. To that end, we've built—and are continuing to build—GameWorks VR. GameWorks VR is the name for a suite of technologies we're developing to tackle the challenges I've just mentioned—high-framerate, low-latency, stereo, and distorted rendering.

It has several different components, which we'll go through in this talk. The first two features, VR SLI and multi-res shading, are targeted more at game and engine developers. The last three are more low-level features, intended for VR headset developers to use in their software stack.



DesignWorks VR

Extra VR features for professional graphics

WARP & BLEND

API for geometry and intensity adjustments for seamless multi-monitor display

SYNCHRONIZATION

Provides tear-free VR environments by synchronizing scanout across GPUs

GPU AFFINITY

Fine-grained control to pin OGL contexts to specific GPUs

GPUDIRECT FOR VIDEO

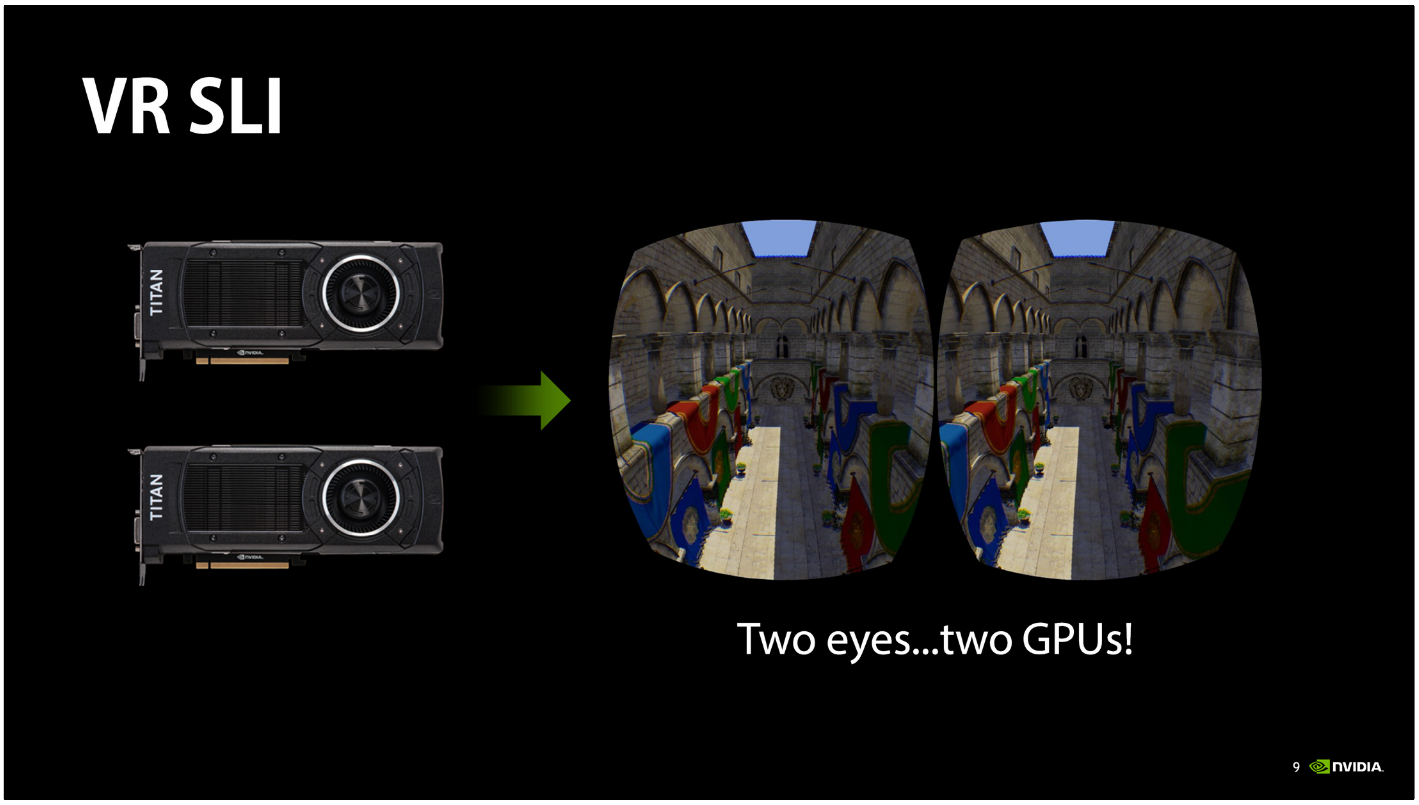
Reduces latency for video transfer to and from the GPU

7

Besides GameWorks VR, we've just announced today another suite of technologies called DesignWorks VR. These are some extra features just for our Quadro line, and they're targeted more at CAVEs, cluster rendering and things like that, rather than VR headsets. I'm not going to be covering these in those session, though—there'll be more information about DesignWorks VR in the coming days. And all the GameWorks VR features that I'm speaking about today are also available on Quadro with DesignWorks VR.

VR SLI

8



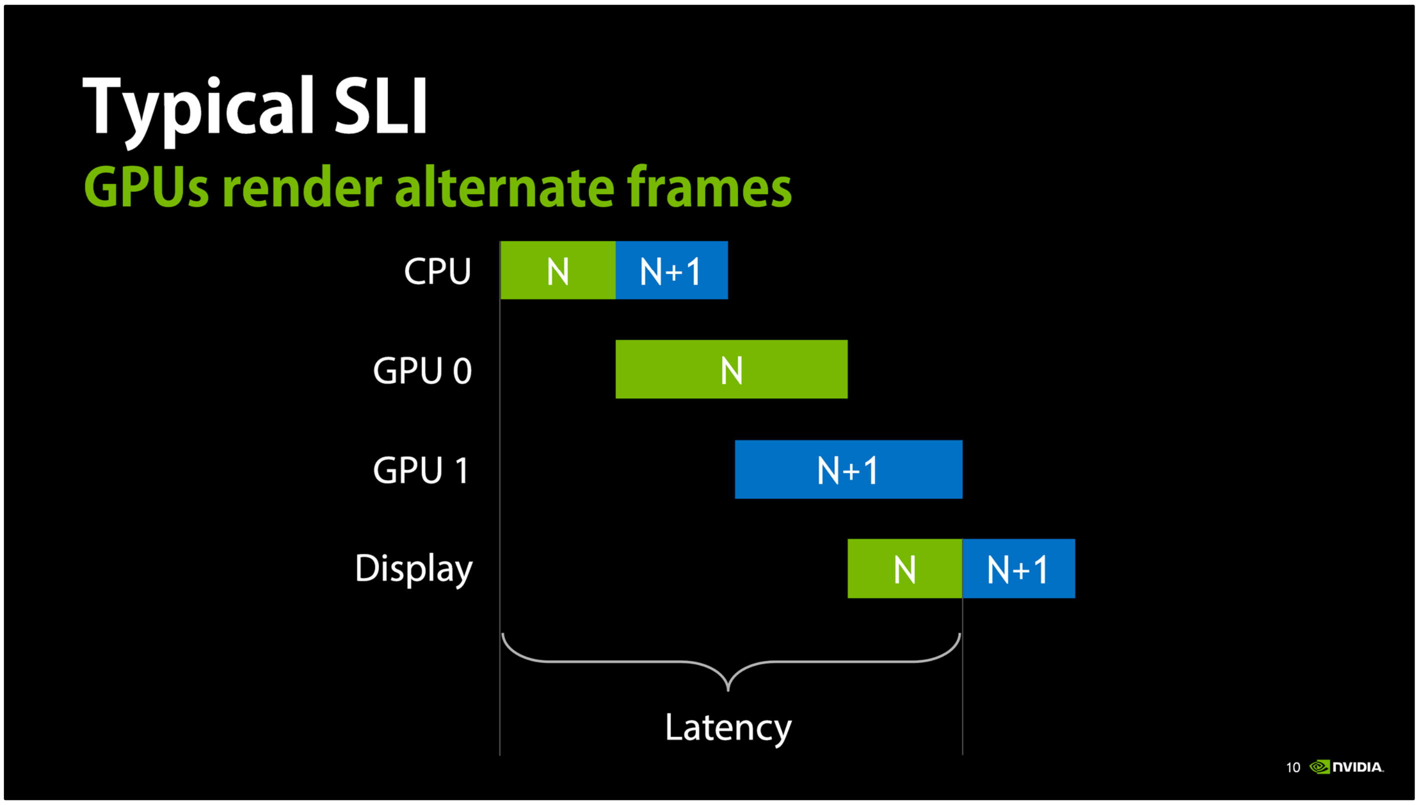
VR SLI

TITAN
TITAN

Two eyes...two GPUs!

Given that the two stereo views are independent of each other, it's intuitively obvious that you can parallelize the rendering of them across two GPUs to get a massive improvement in performance.

In other words, you render one eye on each GPU, and combine both images together into a single frame to send out to the headset. This reduces the amount of work each GPU is doing, and thus improves your framerate—or alternatively, it allows you to use higher graphics settings while staying above the headset's 90 FPS refresh rate, and without hurting latency at all.

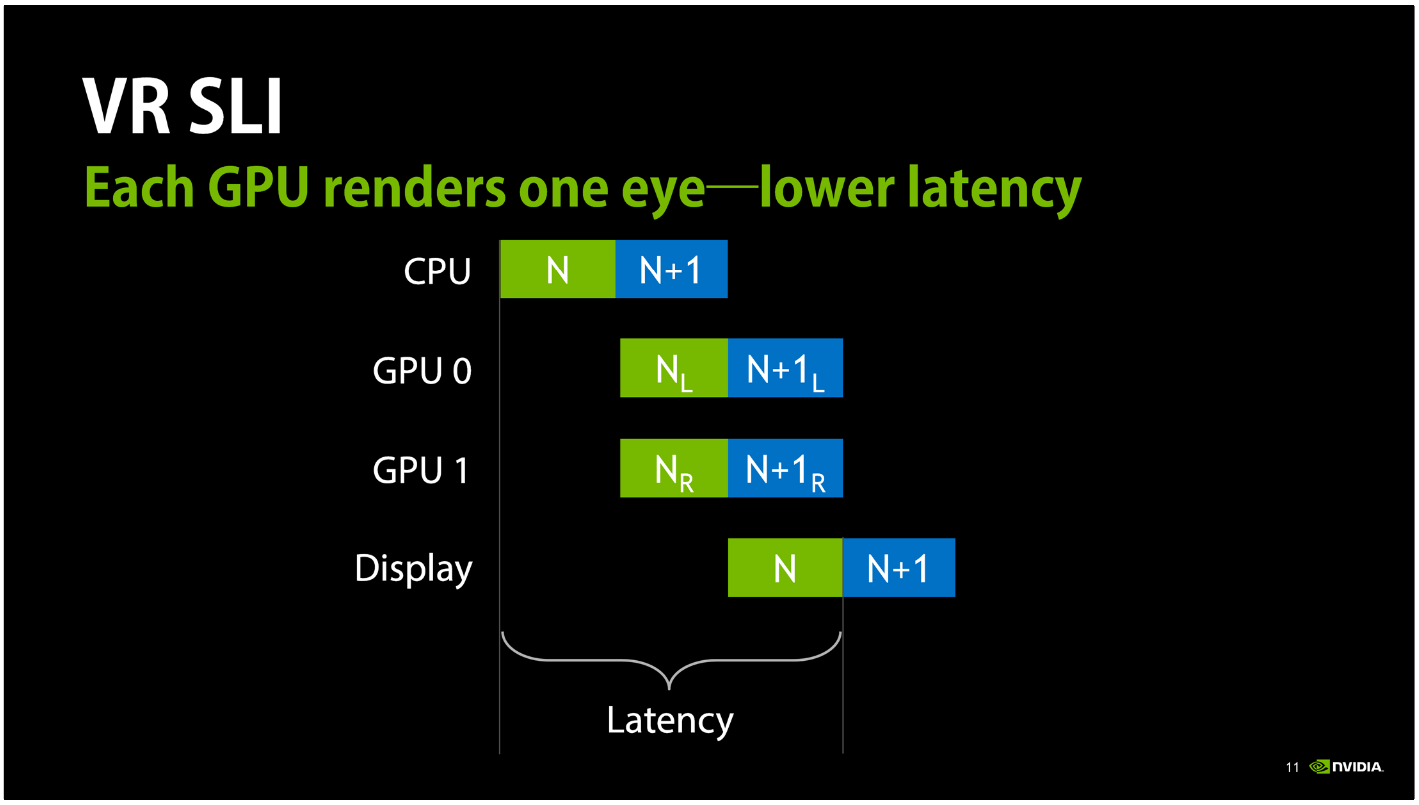


Typical SLI

GPUs render alternate frames

Before we dig into VR SLI, as a quick interlude, let me first explain how ordinary SLI normally works. For years, we've had alternate-frame SLI, in which the GPUs trade off 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.

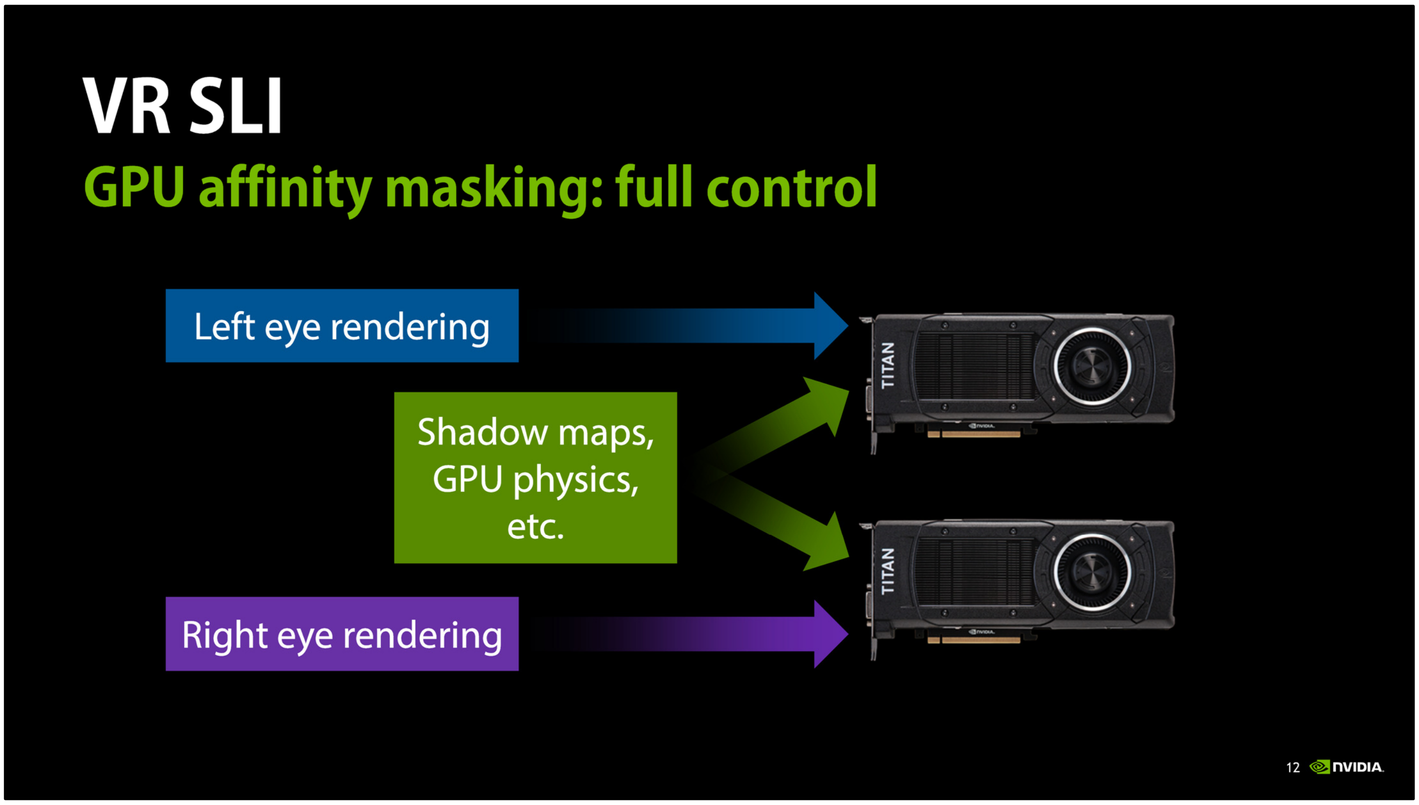
This works reasonably well to increase framerate relative to a single-GPU system, but it doesn't help with latency. So this isn't the best model for VR.



VR SLI

Each GPU renders one eye—lower latency

A better way to use two GPUs for VR rendering is to split the work of drawing a single frame across them—namely, by rendering each eye on one GPU. This has the nice property that it improves both framerate *and* latency relative to a single-GPU system.



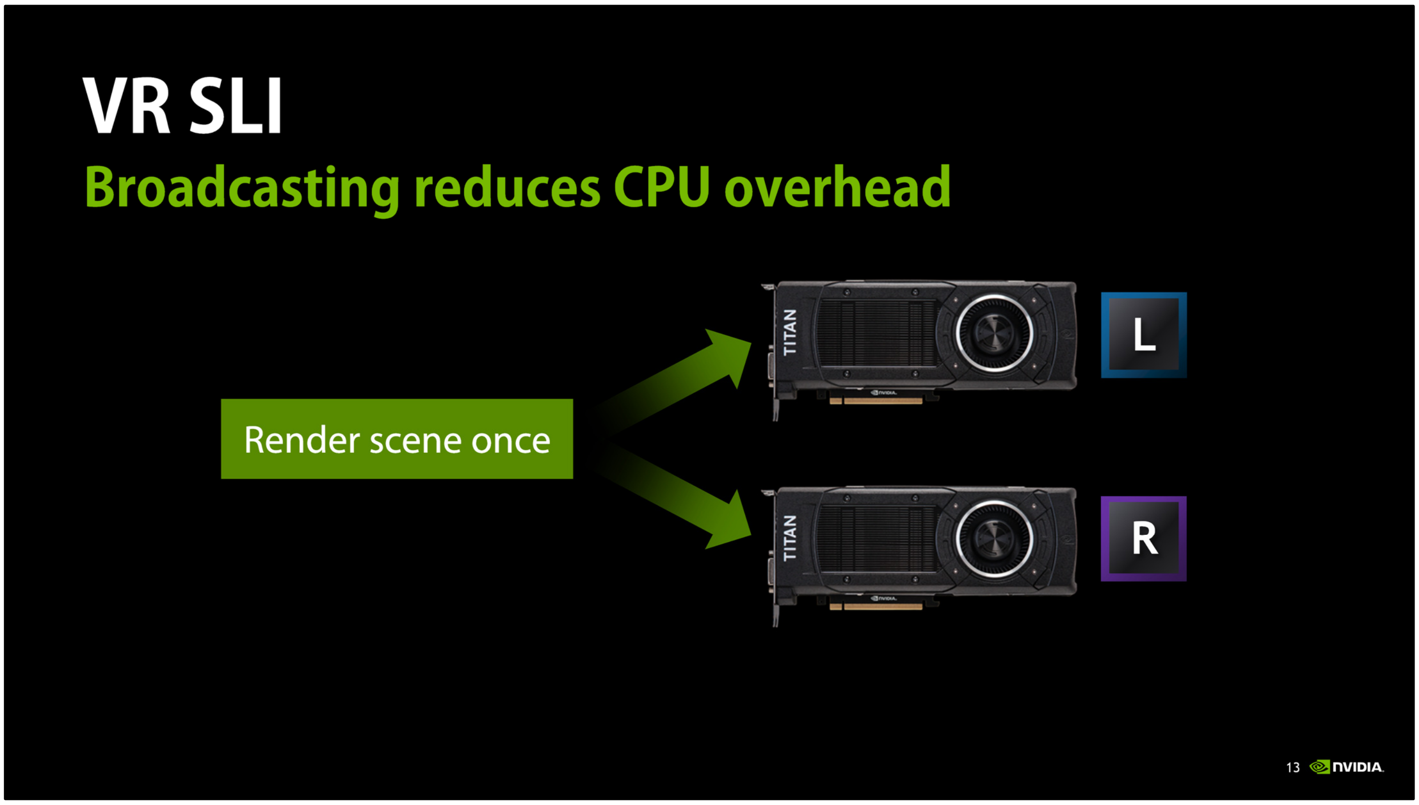
VR SLI

GPU affinity masking: full control

I'll touch on some of the main features of our VR SLI API. First, it enables GPU affinity masking: the ability to select which GPUs a set of draw calls will go to. With our API, you can do this with a simple API call that sets a bitmask of active GPUs. Then all draw calls you issue will be sent to those GPUs, until you change the mask again.

With this feature, if an engine already supports sequential stereo rendering, it's very easy to enable dual-GPU support. All you have to do is add a few lines of code to set the mask to the first GPU before rendering the left eye, then set the mask to the second GPU before rendering the right eye. For things like shadow maps, or GPU physics simulations where the data will be used by both GPUs, you can set the mask to include both GPUs, and the draw calls will be broadcast to them. It really is that simple, and incredibly easy to integrate in an engine.

By the way, all of this extends to as many GPUs as you have in your machine, not just two. So you can use affinity masking to explicitly control how work gets divided across 4 or 8 GPUs, as well.



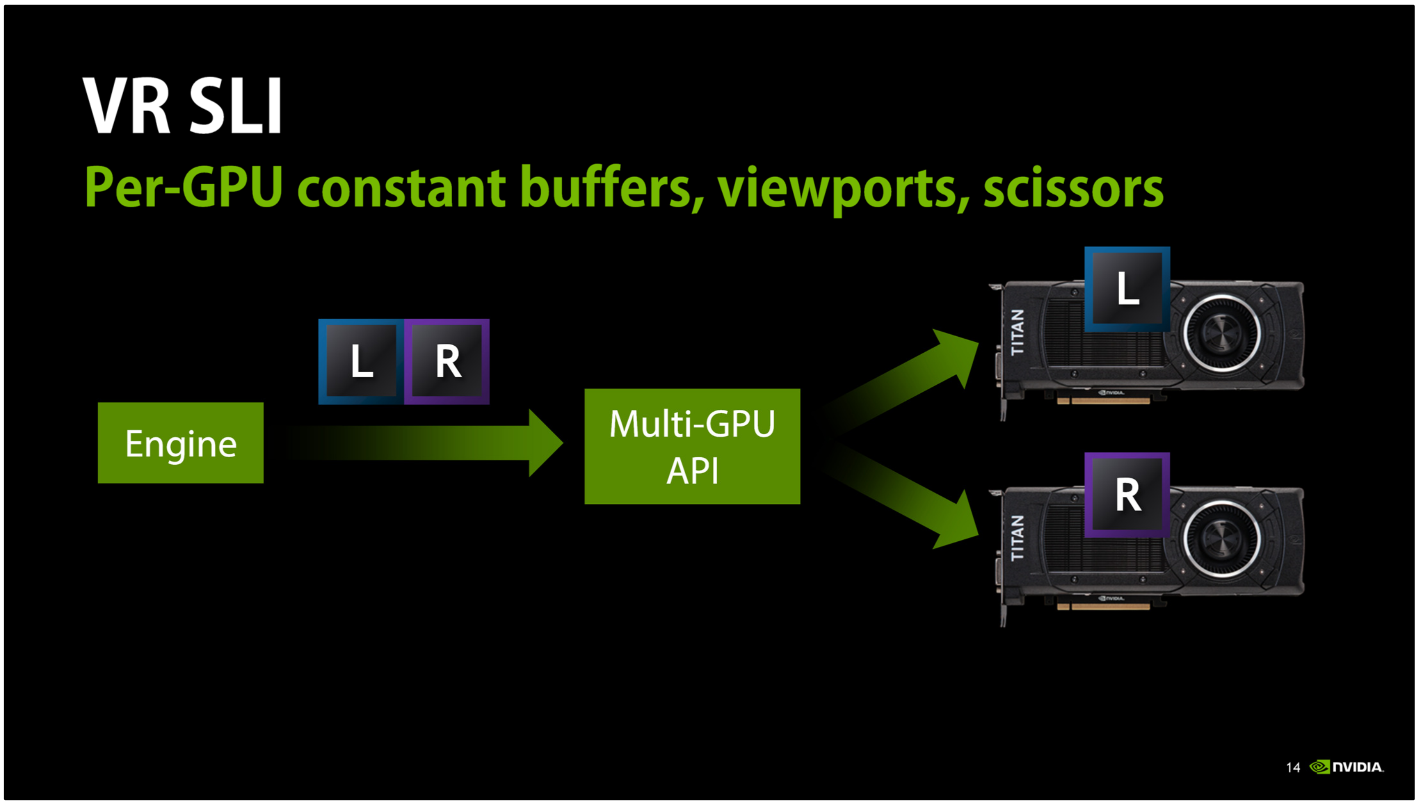
VR SLI

Broadcasting reduces CPU overhead

GPU affinity masking is a great way to get started adding VR SLI support to your engine. However, note that with affinity masking you're still paying the CPU cost for rendering both eyes. After splitting the app's rendering work across two GPUs, your top performance bottleneck can easily shift to the CPU.

To alleviate this, VR SLI supports a second style of use, which we call broadcasting. This allows you to render both eye views using a single set of draw calls, rather than submitting entirely separate draw calls for each eye. Thus, it cuts the number of draw calls per frame—and their associated CPU overhead—roughly in half.

This works because the draw calls for the two eyes are almost completely the same to begin with. Both eyes can see the same objects, are rendering the same geometry, with the same shaders, textures, and so on. So when you render them separately, you're doing a lot of redundant work on the CPU.



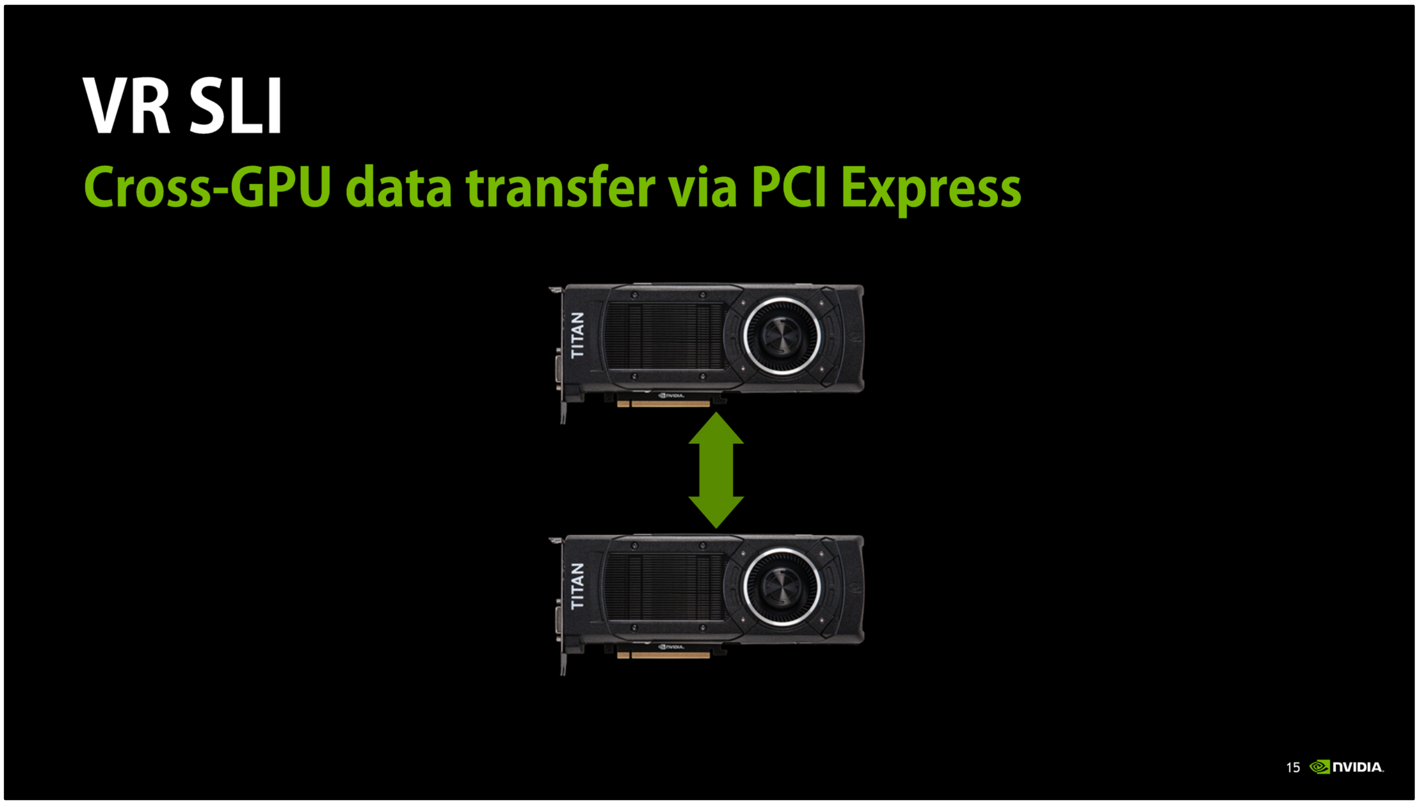
VR SLI

Per-GPU constant buffers, viewports, scissors

The only difference between the eyes is their view position—just a few numbers in a constant buffer. So, VR SLI lets you send different constant buffers to each GPU, so that each eye view is rendered from its correct position when the draw calls are broadcast.

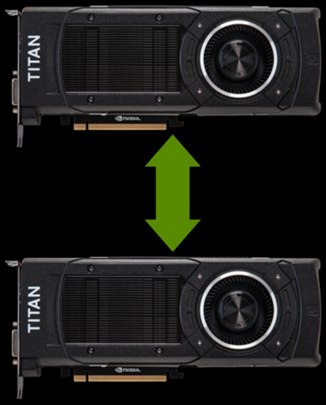
So, you can prepare one constant buffer that contains the left eye view matrix, and another buffer with the right eye view matrix. Then, in our API we have a SetConstantBuffers call that takes both the left and right eye constant buffers at once and sends them to the respective GPUs. Similarly, you can set up the GPUs with different viewports and scissor rectangles.

Altogether, this allows you to render your scene only once, broadcasting those draw calls to both GPUs, and using a handful of per-GPU state settings. This lets you render both eyes with hardly any more CPU overhead then it would cost to render a single view.



VR SLI

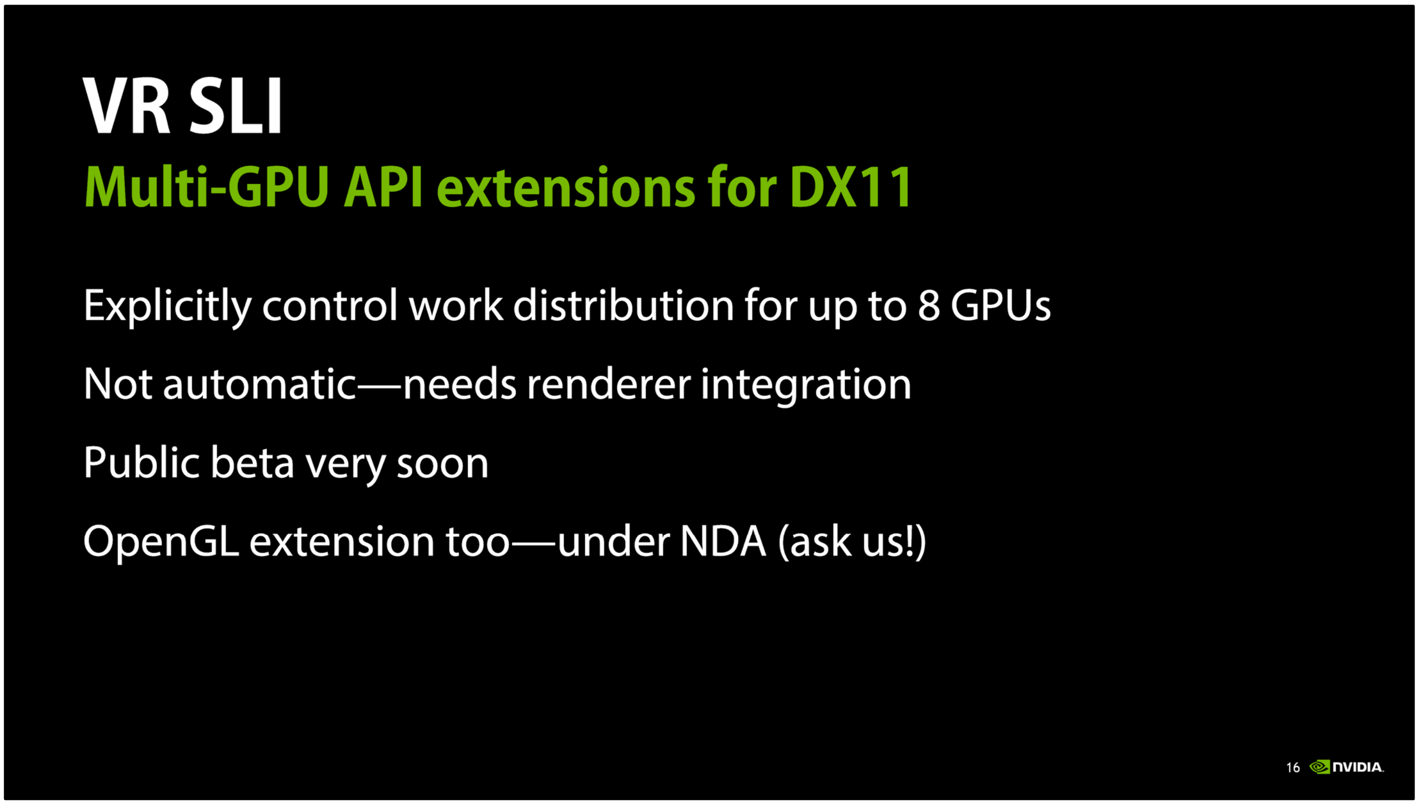
Cross-GPU data transfer via PCI Express



Of course, at times we need to be able to transfer data between GPUs. For instance, after we've finished rendering our two eye views, we have to get them back onto a single GPU to output to the display. So we have an API call that lets you copy a texture or a buffer between two specified GPUs, or to/from system memory, using the PCI Express bus.

One point worth noting here is that the PCI Express bus is actually kind of slow. PCIe2.0 x16 only gives you 8 GB/sec of bandwidth, which isn't that much, and it means that transferring an eye view will require about a millisecond. That's a significant fraction of your frame time at 90 Hz, so that's something to keep in mind.

To help work around that problem, our API supports asynchronous copies. The copy can be kicked off and done in the background while the GPU does some other rendering work, and the GPU can later wait for the copy to finish using fences. So at least you have the opportunity to hide the PCIe latency behind some other work.



VR SLI

Multi-GPU API extensions for DX11

Explicitly control work distribution for up to 8 GPUs

Not automatic—needs renderer integration

Public beta very soon

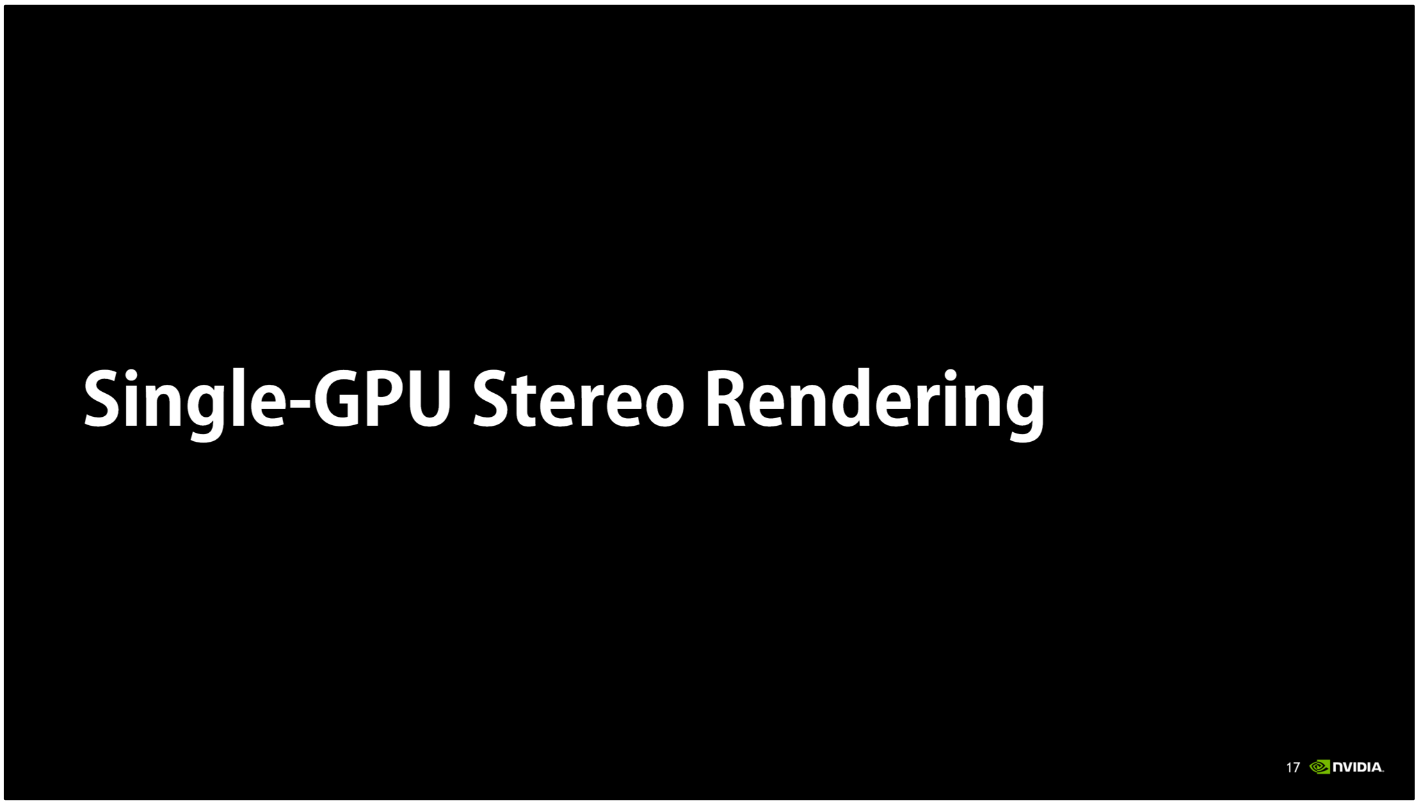
OpenGL extension too—under NDA (ask us!)

16

Stereo rendering with one eye per GPU is the main way we expect people to use VR SLI. But VR SLI is even more than that—it's really a DX11 extension API that gives application developers explicit control over how work is distributed across any number of GPUs. So if you want to support 4 GPUs or even 8 GPUs in a machine, you can do it. The power is in your hands to split up the work however you want, over however many GPUs you choose to support.

An important point here is that since VR SLI is all about giving the application explicit control, it does require active integration into a renderer—it's not possible to automatically enable VR SLI in apps or games that haven't integrated it, in contrast to other SLI modes like alternate-frame rendering.

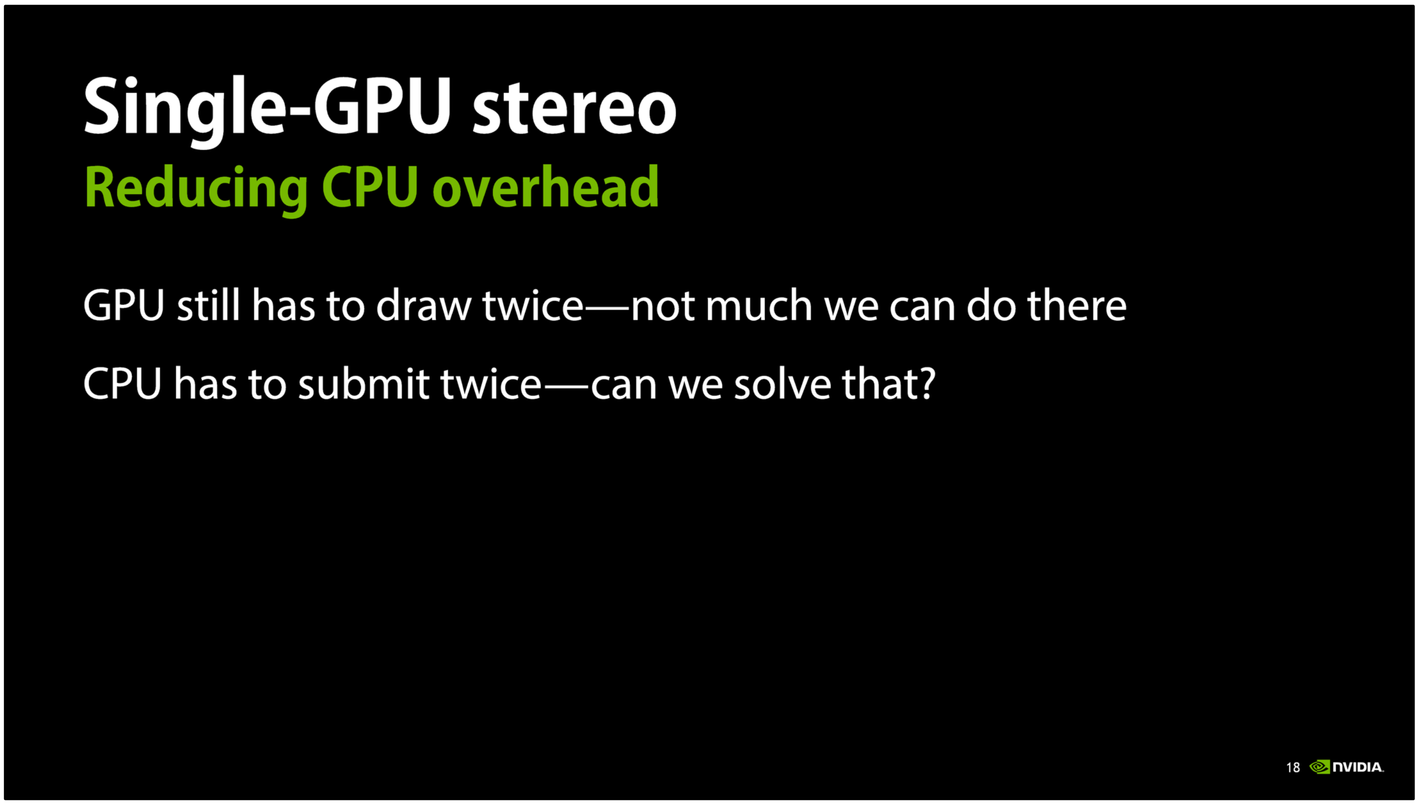
It exists as an extension API to DX11. The VR SLI API is currently under NDA, but will be released as a public beta very soon. And we haven't forgotten about OpenGL—we have a GL extension in progress that provides very similar multi-GPU features. It's still in development and therefore also under NDA for the time being, but if you'd like to test it and give feedback, get in touch with us!



Single-GPU Stereo Rendering

17

VR SLI is a great feature to get maximum performance out of a multi-GPU system. But how can we make stereo rendering more efficient even on a single GPU? That’s where my next topic comes in.



Single-GPU stereo

Reducing CPU overhead

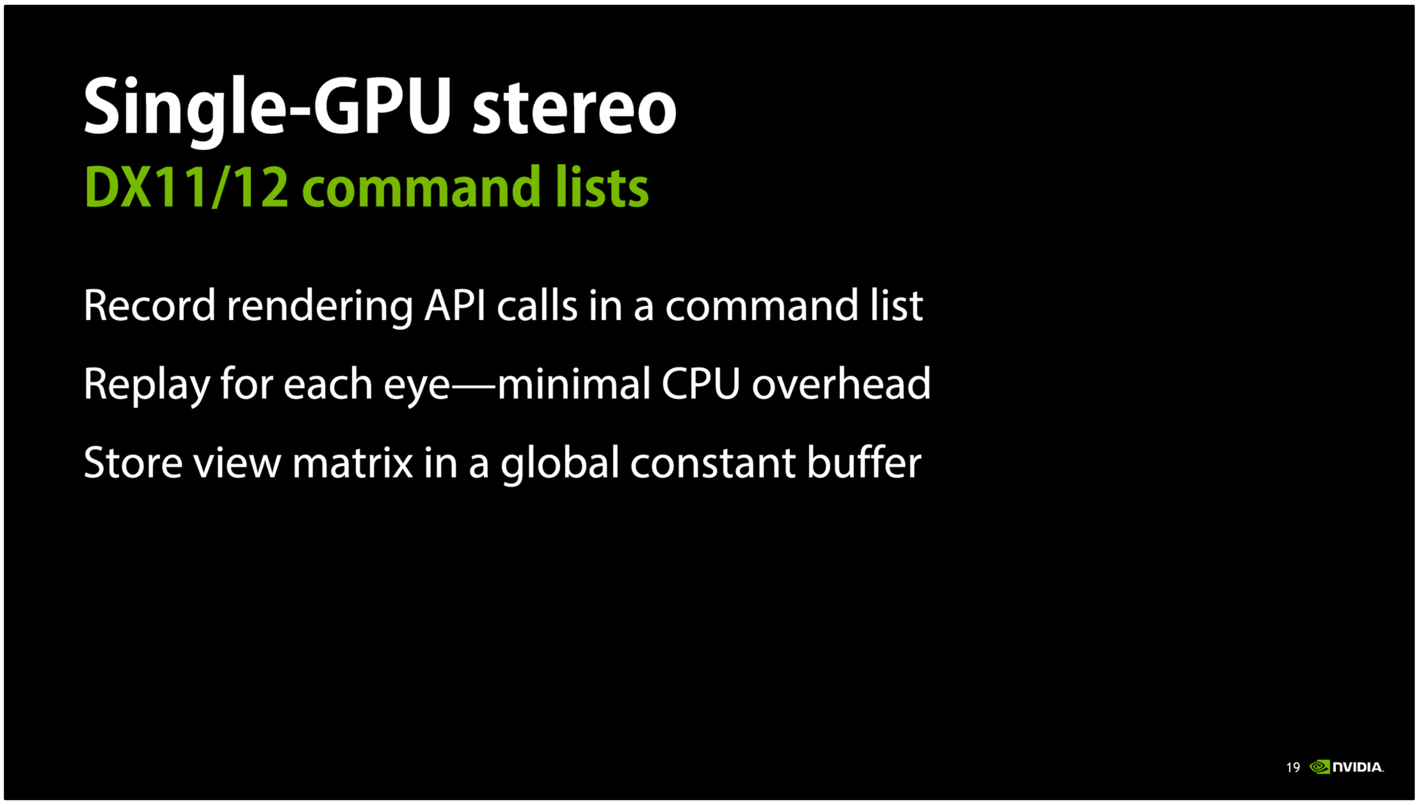
GPU still has to draw twice—not much we can do there

CPU has to submit twice—can we solve that?

18

With VR SLI, we were talking about splitting the stereo rendering work across multiple GPUs. If we're on a single-GPU system, then the GPU just has to draw both views; there's not much we can do about that for the time being.

On the other hand, consider the CPU overhead of resubmitting the draw calls for each eye. With VR SLI we had a broadcast mode where we could submit the draw calls once, cutting the CPU overhead. Can we do something like that with single-GPU rendering?



Single-GPU stereo

DX11/12 command lists

Record rendering API calls in a command list

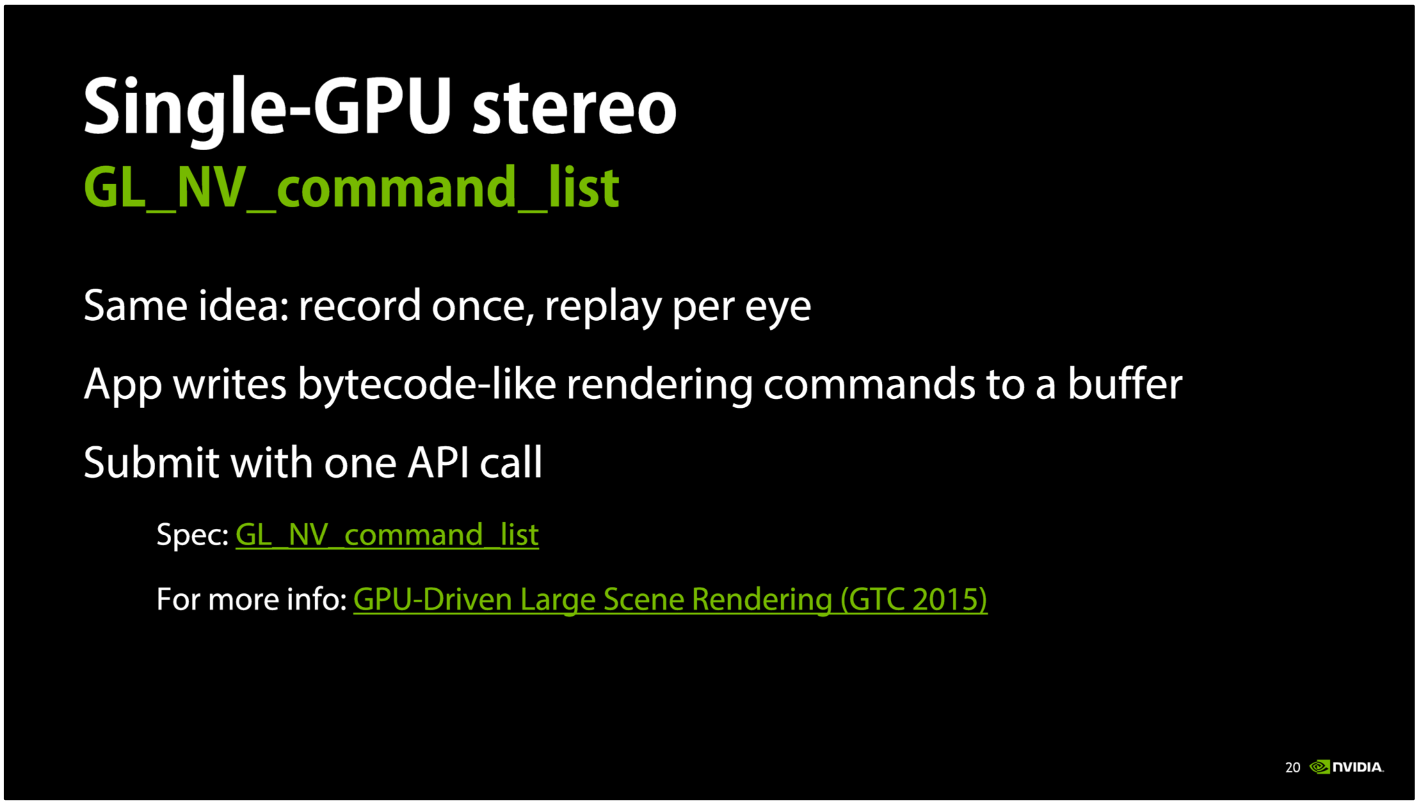
Replay for each eye—minimal CPU overhead

Store view matrix in a global constant buffer

19

In DirectX, both DX11 and DX12 offer the concept of a command list at the API level. You can record a sequence of API calls into a command list, and the driver will do as much of the internal processing as it can when the command list is created, making it relatively cheap to execute later.

So, you can record your scene rendering in a command list, then replay the command list for each eye with minimal CPU overhead. In order to get the correct view matrix for each eye, you can store it in a global constant buffer, and update it between eyes. That works because command lists only store references to resources like buffers and textures, not the contents of the resources.


Single-GPU stereo
GL_NV_command_list

Same idea: record once, replay per eye

App writes bytecode-like rendering commands to a buffer

Submit with one API call

Spec: GL_NV_command_list

For more info: GPU-Driven Large Scene Rendering (GTC 2015)

In OpenGL, there's no native concept of command lists. (There were display lists once upon a time, but they're deprecated now, and they don't quite do what you want anyway.) However, on NVIDIA GPUs we expose an extension called GL_NV_command_list that offers similar functionality.

I won't go into details here, but basically the extension defines a bytecode-like binary format in which you encode your rendering commands and store them all in a buffer; then you can submit that to the driver in one very efficient API call. Again, you can record your scene rendering in this buffer once, then submit it twice to render both eyes, while updating a uniform buffer in between to set the correct view matrix for each eye.

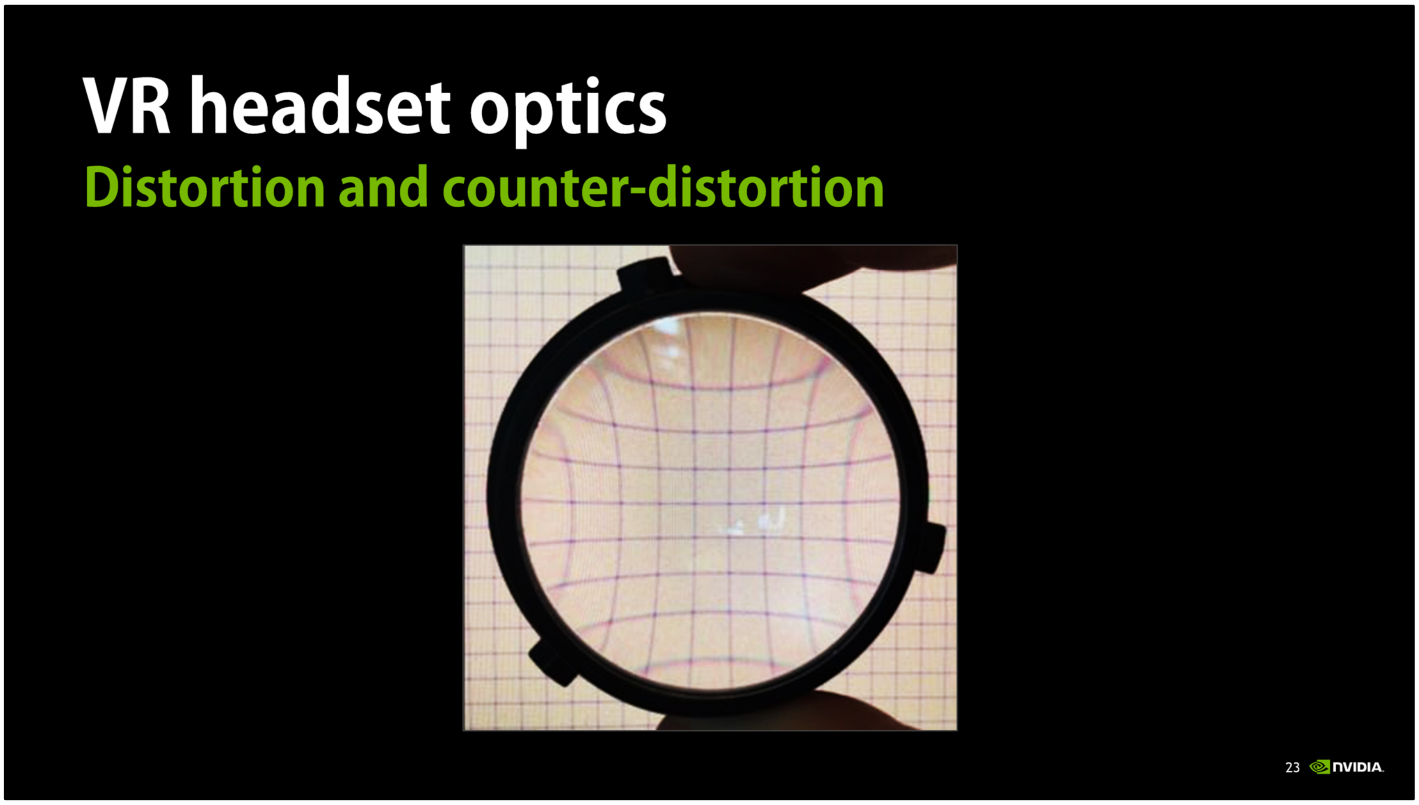
Spec: http://developer.download.nvidia.com/opengl/specs/GL_NV_command_list.txt

For more info: http://on-demand.gputechconf.com/gtc/2015/presentation/S5135-Christoph-Kubisch-Pierre-Boudier.pdf

Multi-Resolution Shading

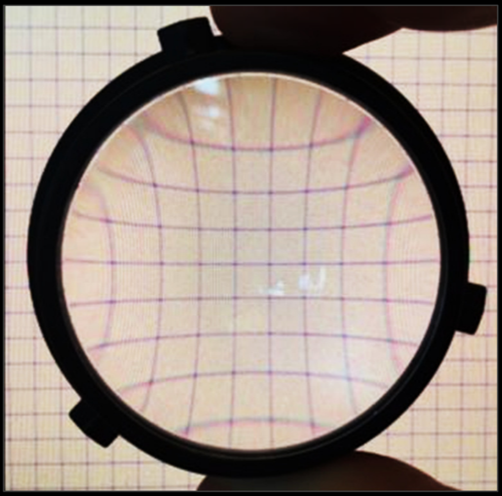
22

We’ve been talking a lot about how to do efficient stereo rendering for VR. Now I’m going to switch gears and talk about how we can take advantage of the optics in a VR headset to improve rendering performance.



VR headset optics

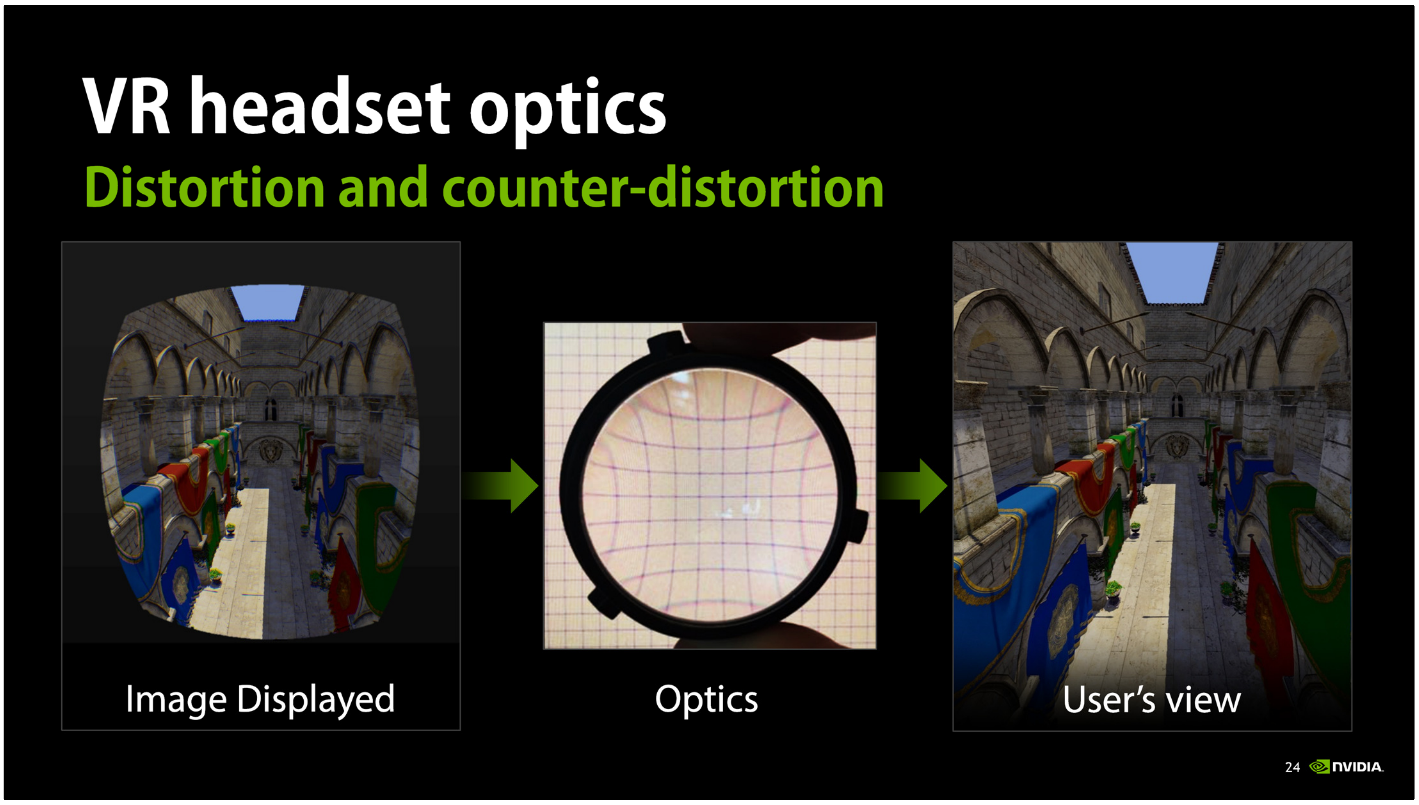
Distortion and counter-distortion



23

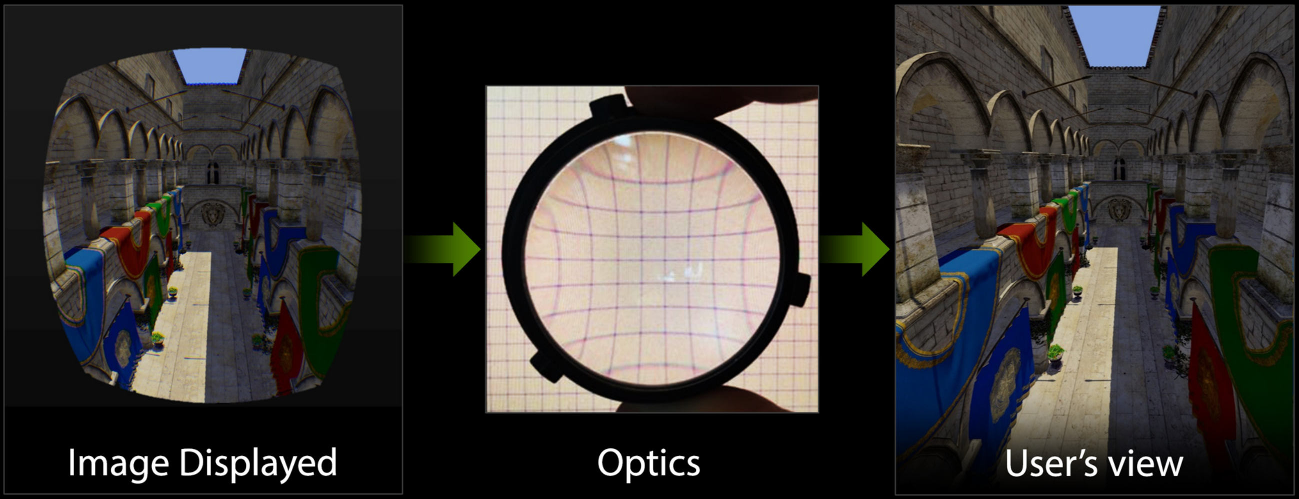
First, the basic facts about how the optics in a VR headset work.

VR headsets have lenses to expand their field of view and enable your eyes to focus on the screen. However, the lenses also introduce pincushion distortion in the image, as seen here. Note how the straight grid lines on the background are bowed inward when seen through the lens.



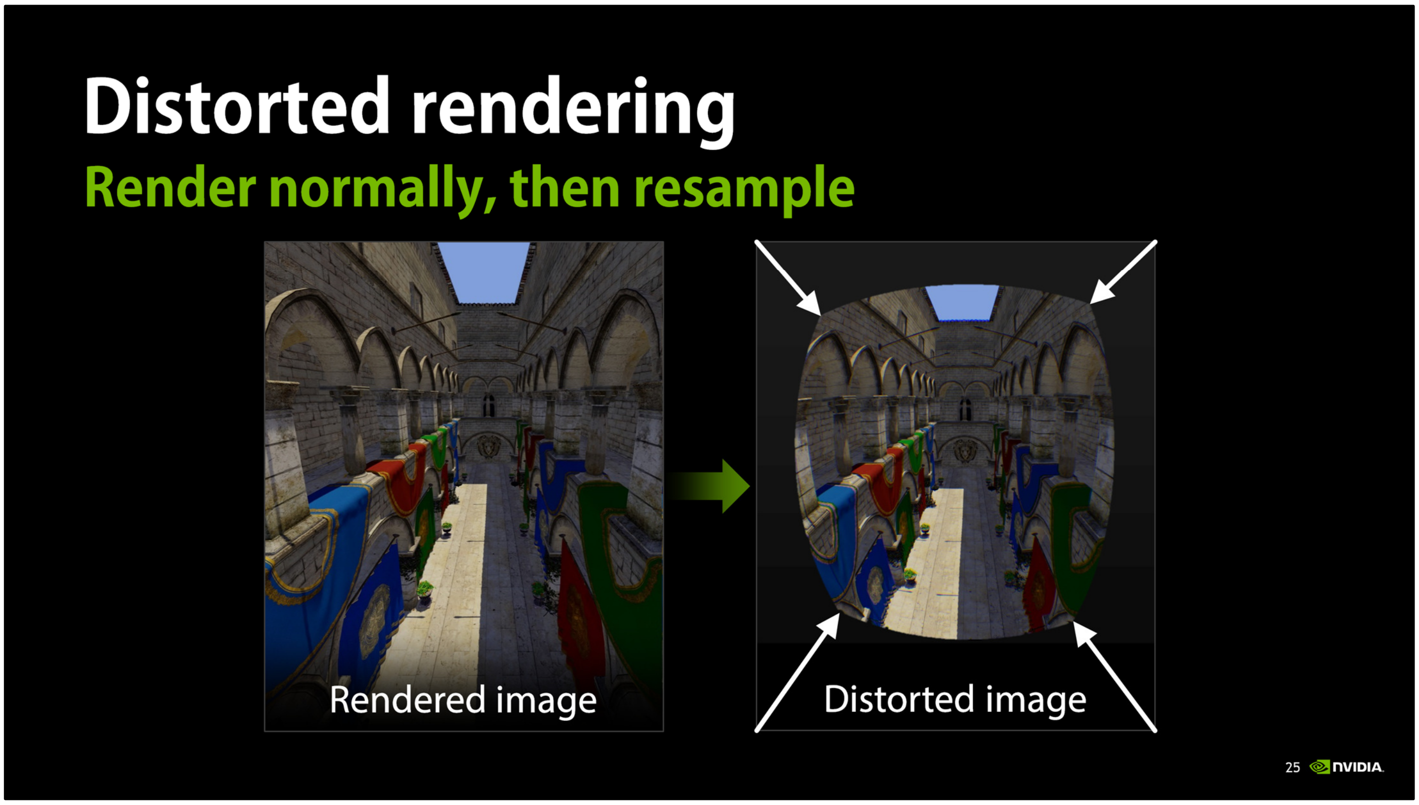
VR headset optics

Distortion and counter-distortion



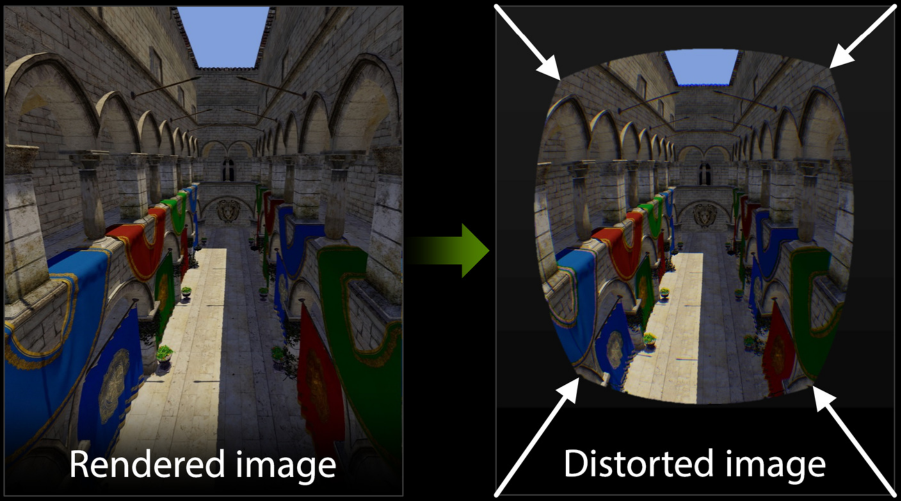
So we have to render an image that's distorted in the opposite way—barrel distortion, like what you see on the right—to cancel out the lens effects. When viewed through the lens, the user perceives a geometrically correct image again.

Chromatic aberration, or the separation of red, green, and blue colors, is another lens artifact that we have to counter in software to give the user a faithfully rendered view.



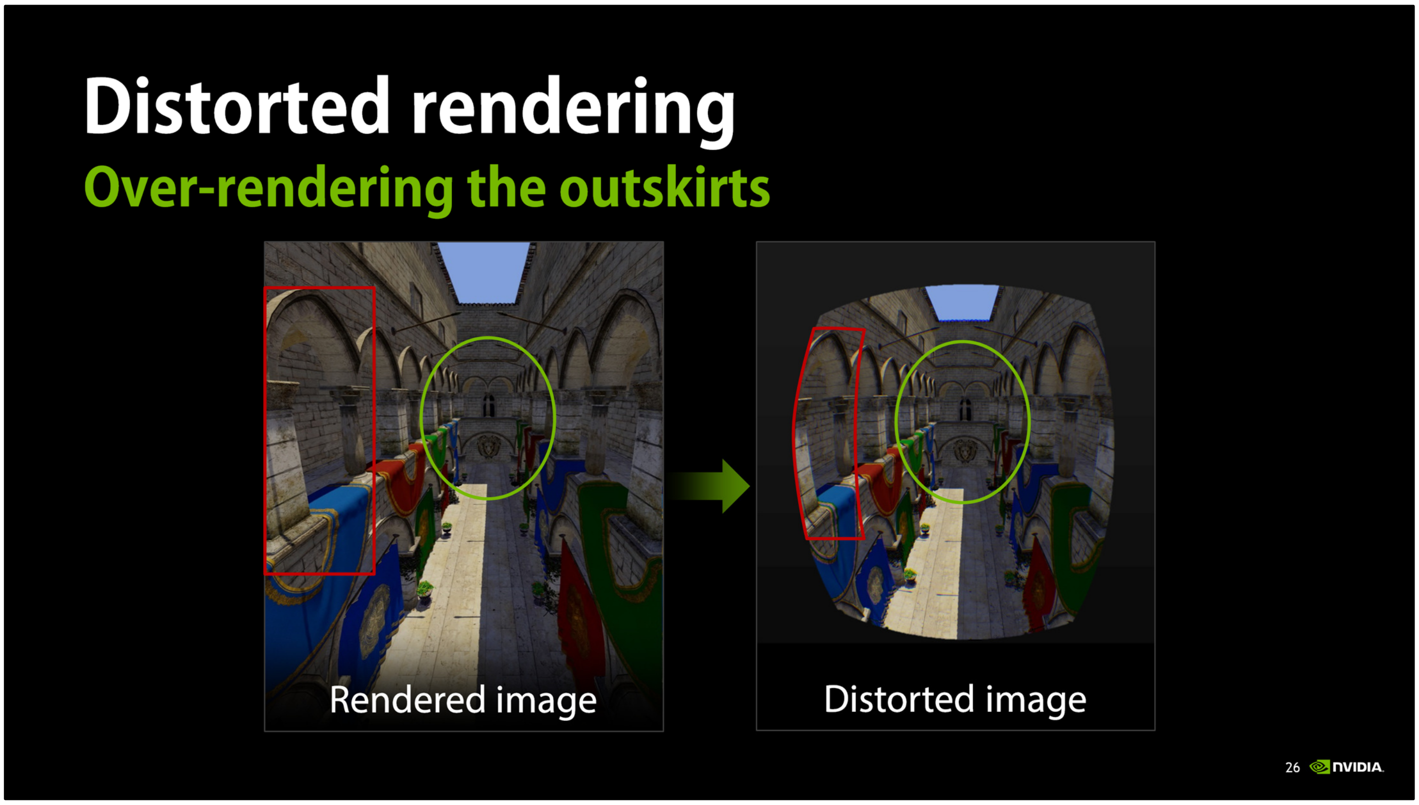
Distorted rendering

Render normally, then resample



The trouble is that GPUs can't natively render into a nonlinearly distorted view like this—their rasterization hardware is designed around the assumption of linear perspective projections. Current VR software solves this problem by first rendering a normal perspective projection (left), then resampling to the distorted view (right) as a postprocess.

You'll notice that the original rendered image is much larger than the distorted view. In fact, on the Oculus Rift and HTC Vive headsets, the recommended rendered image size is close to double the pixel count of the final distorted image.



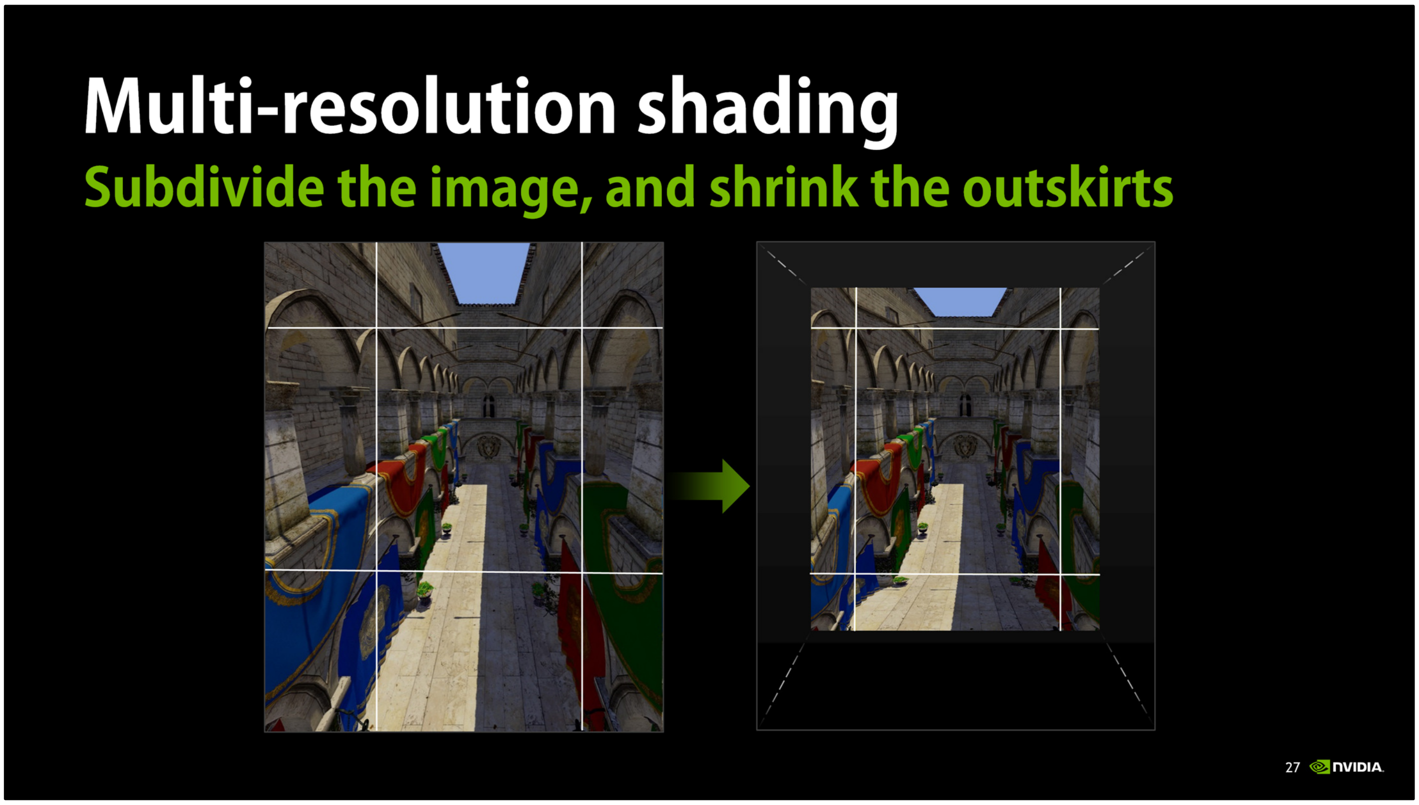
Distorted rendering

Over-rendering the outskirts

The reason for this is that if you look at what happens during the distortion pass, you find that while the center of the image stays the same, the outskirts are getting squashed quite a bit.

Look at the green circles—they're the same size, and they enclose the same region of the image in both the original and the distorted views. Then compare that to the red box. It gets mapped to a significantly smaller region in the distorted view.

This means we're over-shading the outskirts of the image. We're rendering and shading lots of pixels that are never making it out to the display—they're just getting thrown away during the distortion pass. It's a significant inefficiency, and it slows you down.



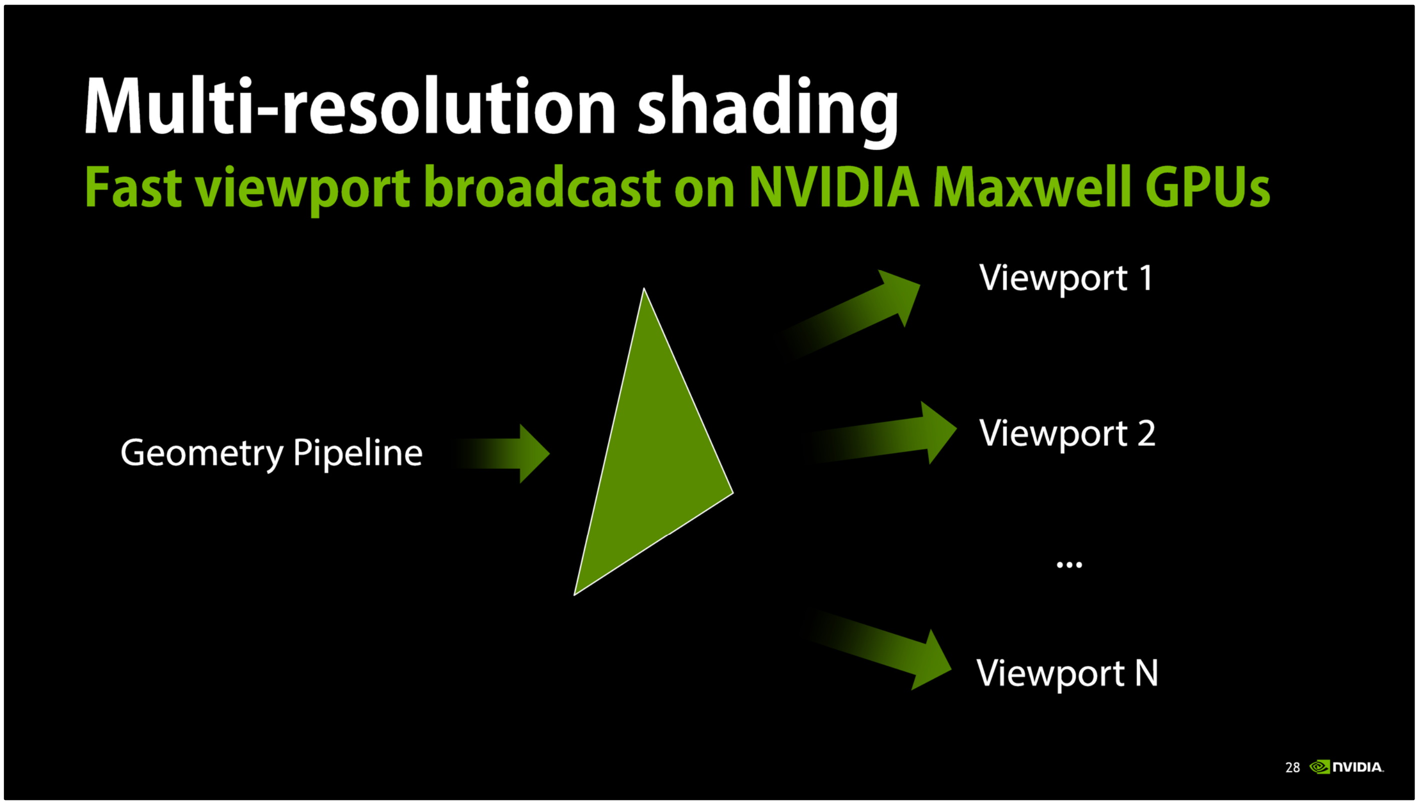
Multi-resolution shading

Subdivide the image, and shrink the outskirts

That brings us to multi-resolution shading. The idea is to subdivide the image into a set of adjoining viewports—here, a 3x3 grid of them. We keep the center viewport the same size, but scale down all the ones around the outside. All the left, right, top and bottom edges are scaled in, effectively reducing the resolution in the outskirts of the image, while maintaining full resolution at the center.

Now, because everything is still just a standard, rectilinear perspective projection, the GPU can render natively into this collection of viewports. But now we're better approximating the pixel density of the distorted image that we eventually want to generate. Since we're closer to the final pixel density, we're not over-rendering and wasting so many pixels, and we can get a substantial performance boost for no perceptible reduction in image quality.

Depending on how aggressive you want to be with scaling down the outer regions, you can save anywhere from 20% to 50% of the pixels.



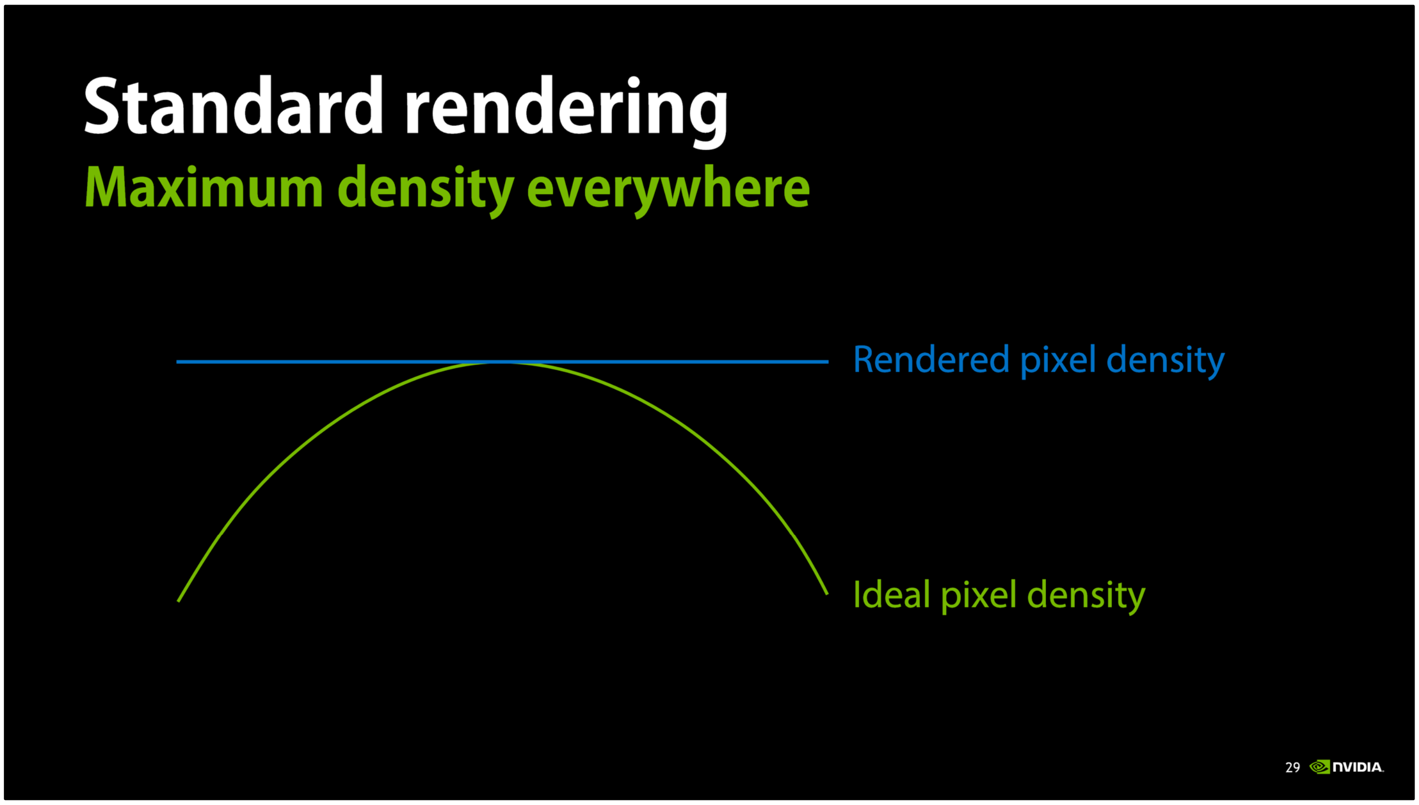
Multi-resolution shading

Fast viewport broadcast on NVIDIA Maxwell GPUs

The key thing that makes this technique a performance win is a hardware feature we have on NVIDIA's Maxwell architecture—in other words, the GeForce GTX 900 series, Titan X, and Quadro M6000.

Ordinarily, replicating all scene geometry to several viewports would be prohibitively expensive. There are various ways you can do it, such as resubmitting draw calls, instancing, and geometry shader expansion—but all of those will add enough overhead to eat up any gains you got from reducing the pixel count.

With Maxwell, we have the ability to very efficiently broadcast the geometry to many viewports, of arbitrary shapes and sizes, in hardware, while only submitting the draw calls once and running the GPU geometry pipeline once. That lets us render into this multi-resolution render target in a single pass, just as efficiently as an ordinary render target.

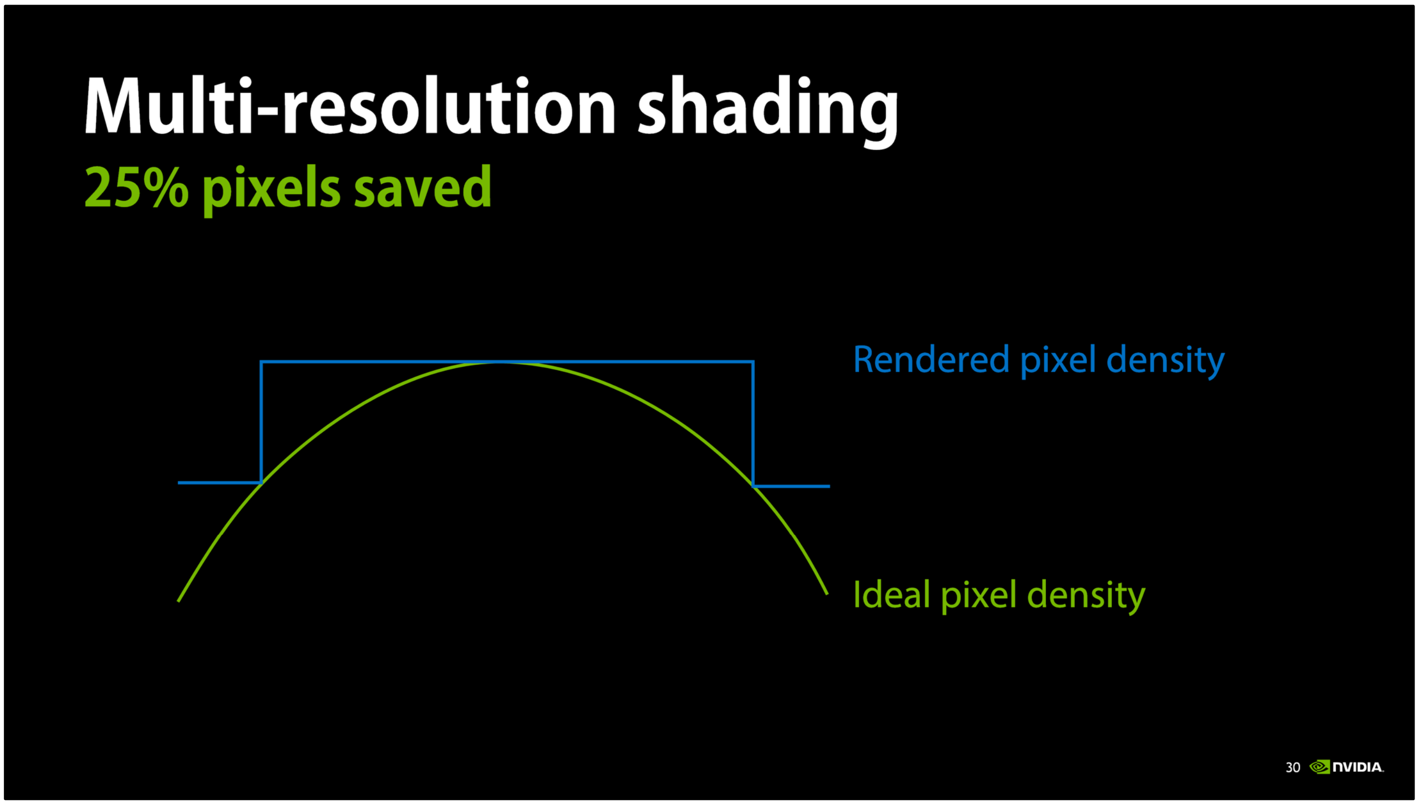


Standard rendering

Maximum density everywhere

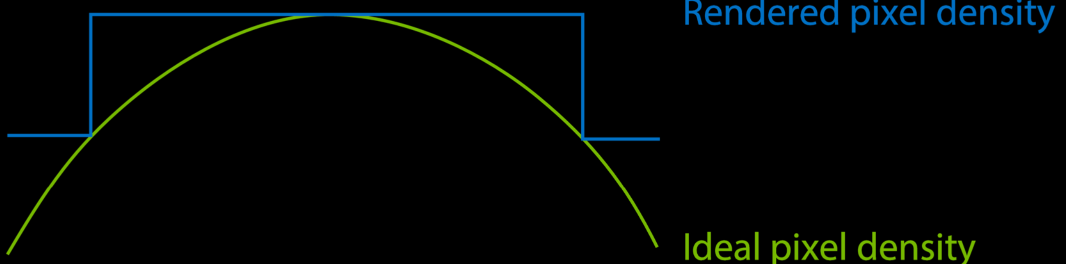
Another way to understand what multi-resolution shading is doing is to look at a graph of pixel density aross the image.

The green line represents the ideal pixel density needed for the final image; it's high in the center, but falls off toward the edges where the barrel distortion squashes the image. With standard rendering, we're taking the maximum density—which occurs at the center—and rendering the entire image at that high density. The space between these curves represents wasted over-rendering.



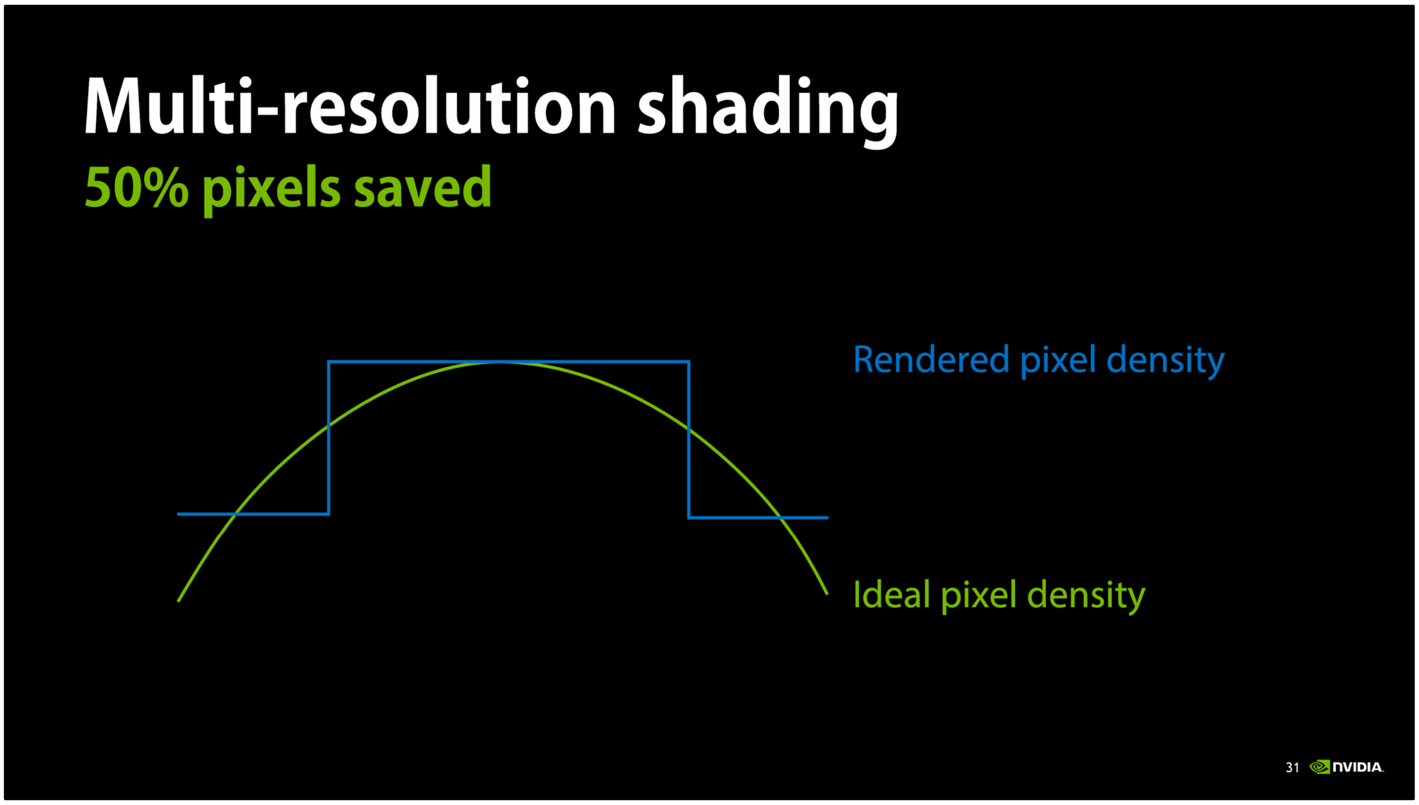
Multi-resolution shading

25% pixels saved



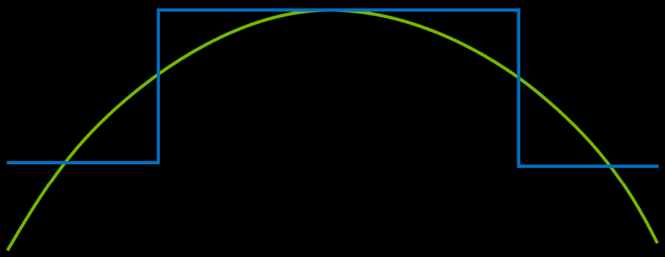
With multi-resolution shading, we can reduce the pixel density at the edges, to more closely approximate the ideal density.

In this graph, I'm showing a "conservative" multi-res setting in which the blue line never drops below the green line. In other words, we never have less than one rendered pixel per display pixel, anywhere in the image. This setting lets us save about 25% of the pixels. In the ideal case where you're perfectly pixel-bound, that works out to a 1.3x performance improvement.



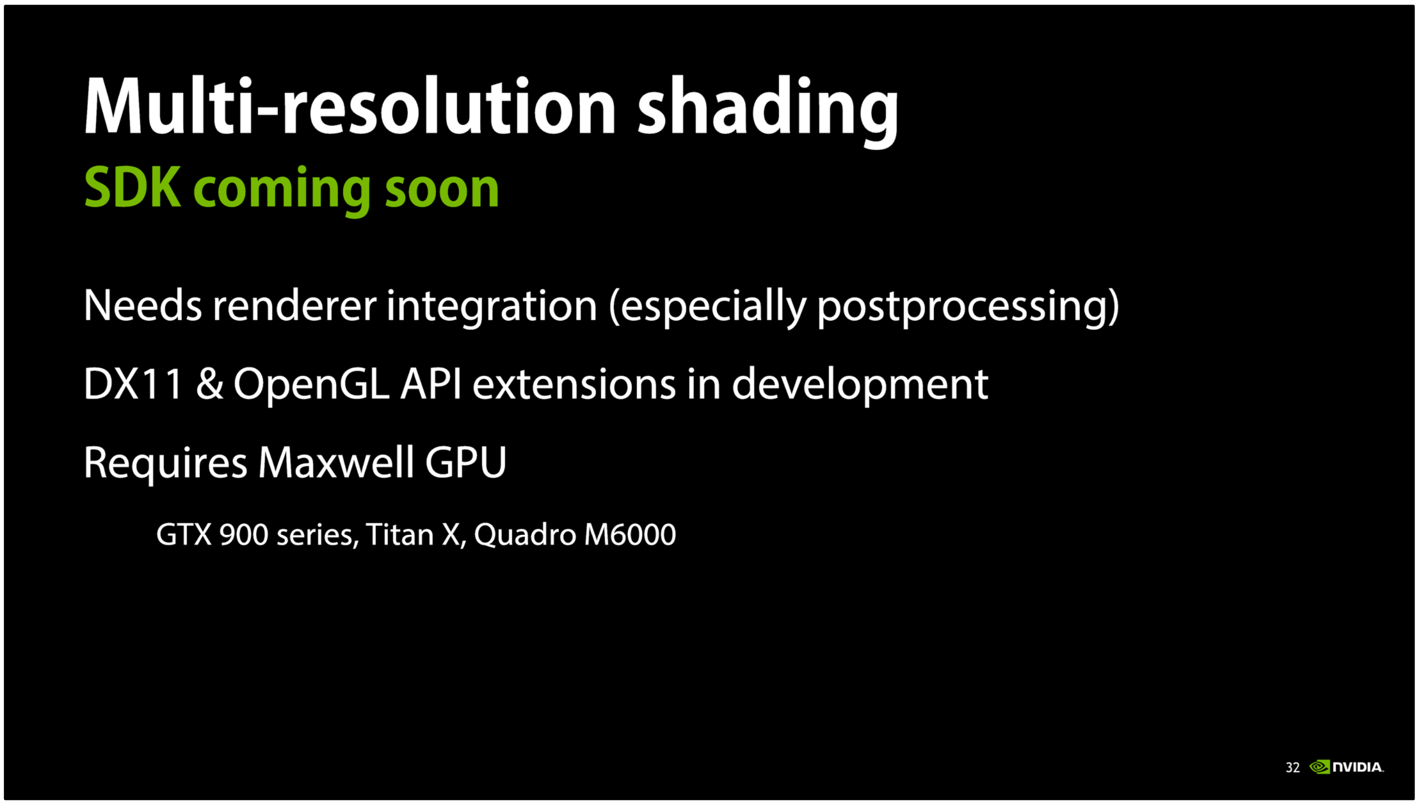
Multi-resolution shading

50% pixels saved



31

It's also possible to go to a more aggressive setting, where we reduce the pixel density at the edges of the image even further. This requires some care, as it could visibly affect image quality, depending on your scene. But in many scenes, even an aggressive setting like this may be almost unnoticeable—and it lets you save 50% of the pixels, which translates into a 2x speedup if perfectly pixel-bound.



Multi-resolution shading

SDK coming soon

Needs renderer integration (especially postprocessing)

DX11 & OpenGL API extensions in development

Requires Maxwell GPU

- GTX 900 series, Titan X, Quadro M6000

32

Like VR SLI, multi-res shading will require developers to integrate the technique in their applications—it's not something that can be turned on automatically by a driver. Beyond simply enabling multi-res in the main pass, developers will also have to modify most screen-space postprocessing passes that operate on the multi-res render target, such as bloom, SSAO, or motion blur. Deferred shading renderers will also need modifications to their lighting passes.

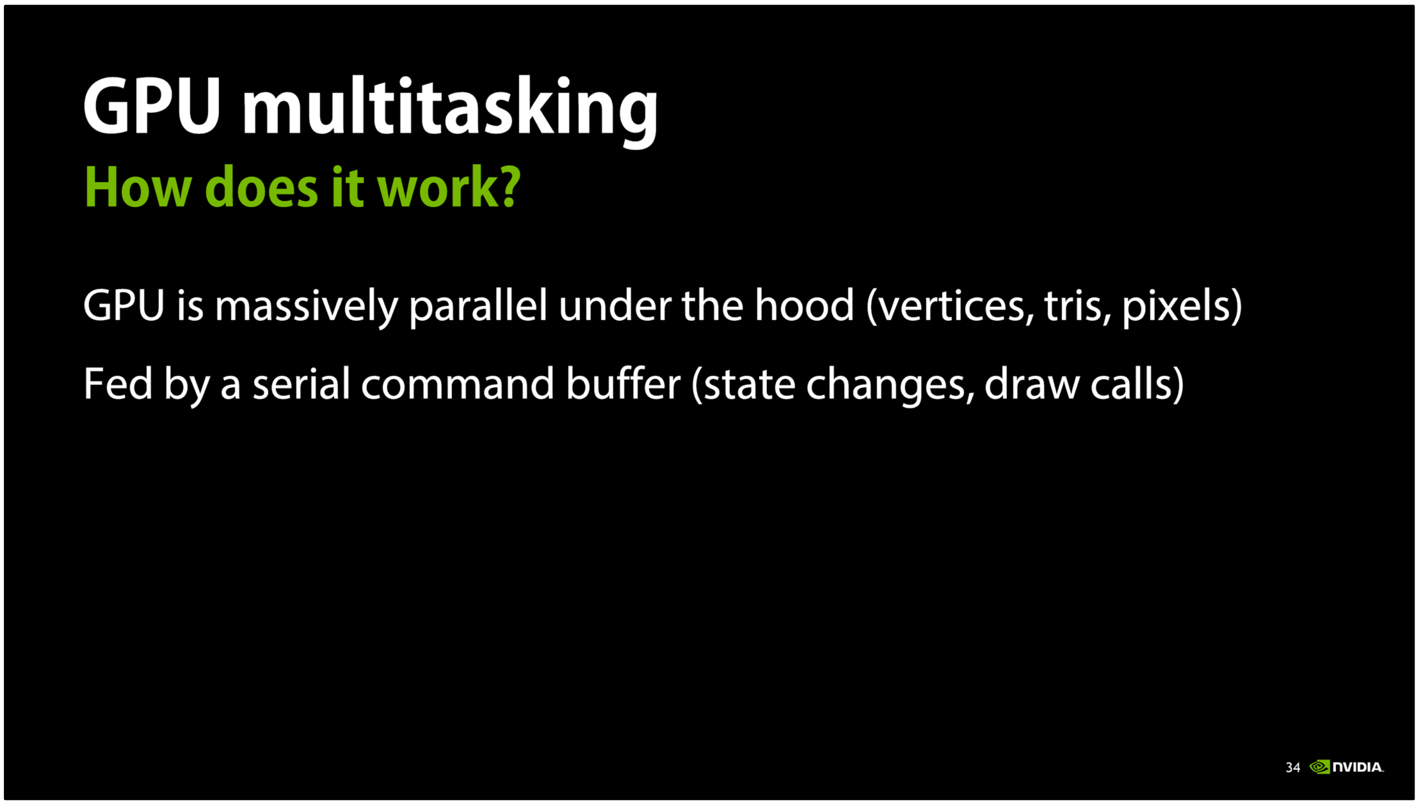
Our SDK for multi-res shading is still in development, but stay tuned for updates over the next couple of months. Like VR SLI, multi-res shading will also take the form of DX11 and OpenGL extensions. And because it relies on the Maxwell fast viewport broadcast feature, it's only going to be available for Maxwell GPUs—namely, the GTX 900 series, Titan X, and Quadro M6000.

GPU Multitasking and VR

33

We’ve been talking a lot about methods of improving rendering throughput for VR: distributing work across GPUs, getting more draw calls out faster, and cutting down the number of pixels you need to render. All of those techniques are meant for application, engine, and game developers to use to improve their VR rendering performance.

In the remainder of this session, I’ll talk about some lower-level features that are targeted more at people making VR headsets themselves, and the software stacks that enable apps and games to use them. To start with, I’m going to explain a little about how GPU multitasking works.



GPU multitasking

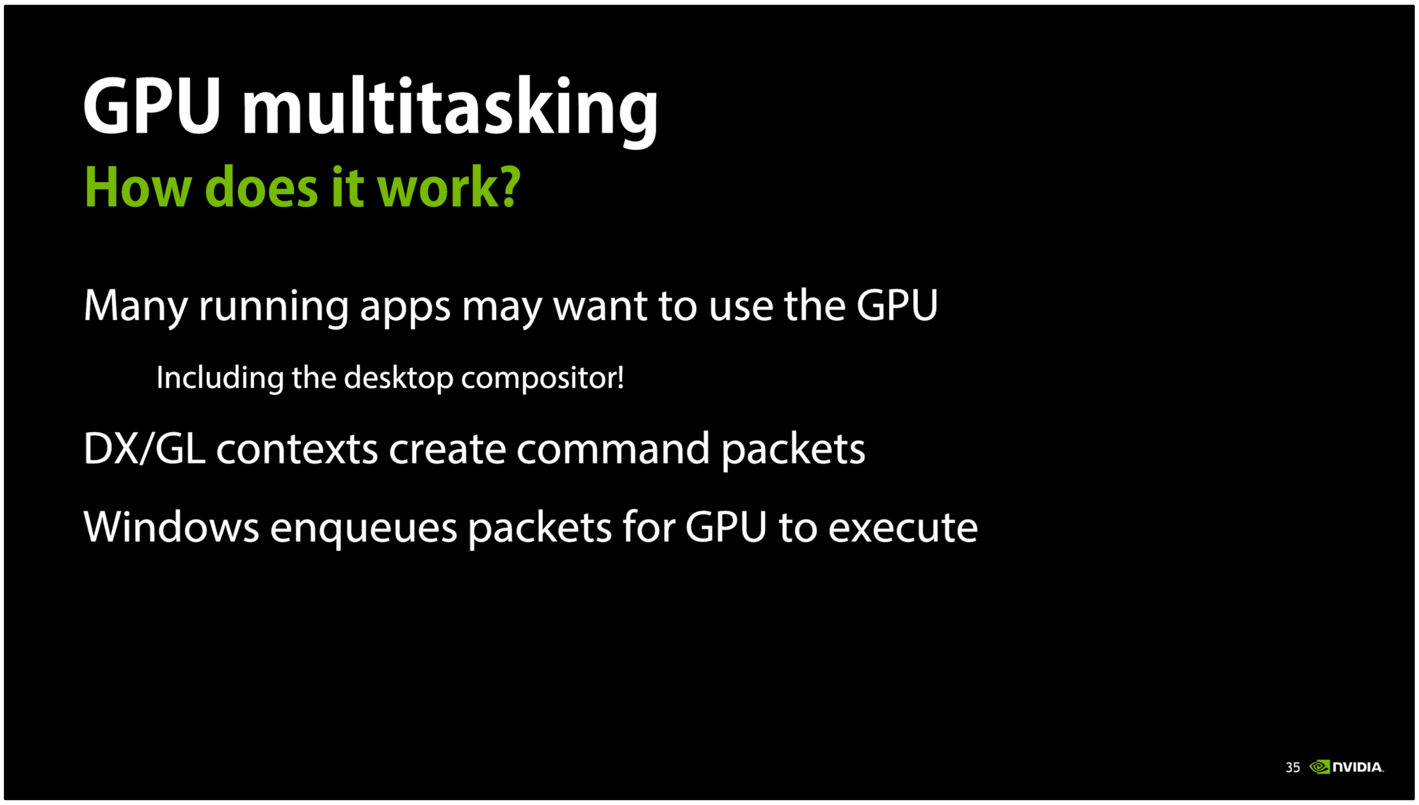
How does it work?

GPU is massively parallel under the hood (vertices, tris, pixels)

Fed by a serial command buffer (state changes, draw calls)

34

It's often said that the GPU is a massively parallel processor, and it is—it can process many vertices, triangles, pixels and so on in parallel. However, the front end of the GPU is still mostly serial. The front end is the part that accepts buffers of rendering commands, such as state changes and draw calls, from applications. It consumes them in order and launches work into the rest of the GPU based on their instructions.


GPU multitasking
How does it work?

Many running apps may want to use the GPU

- Including the desktop compositor!

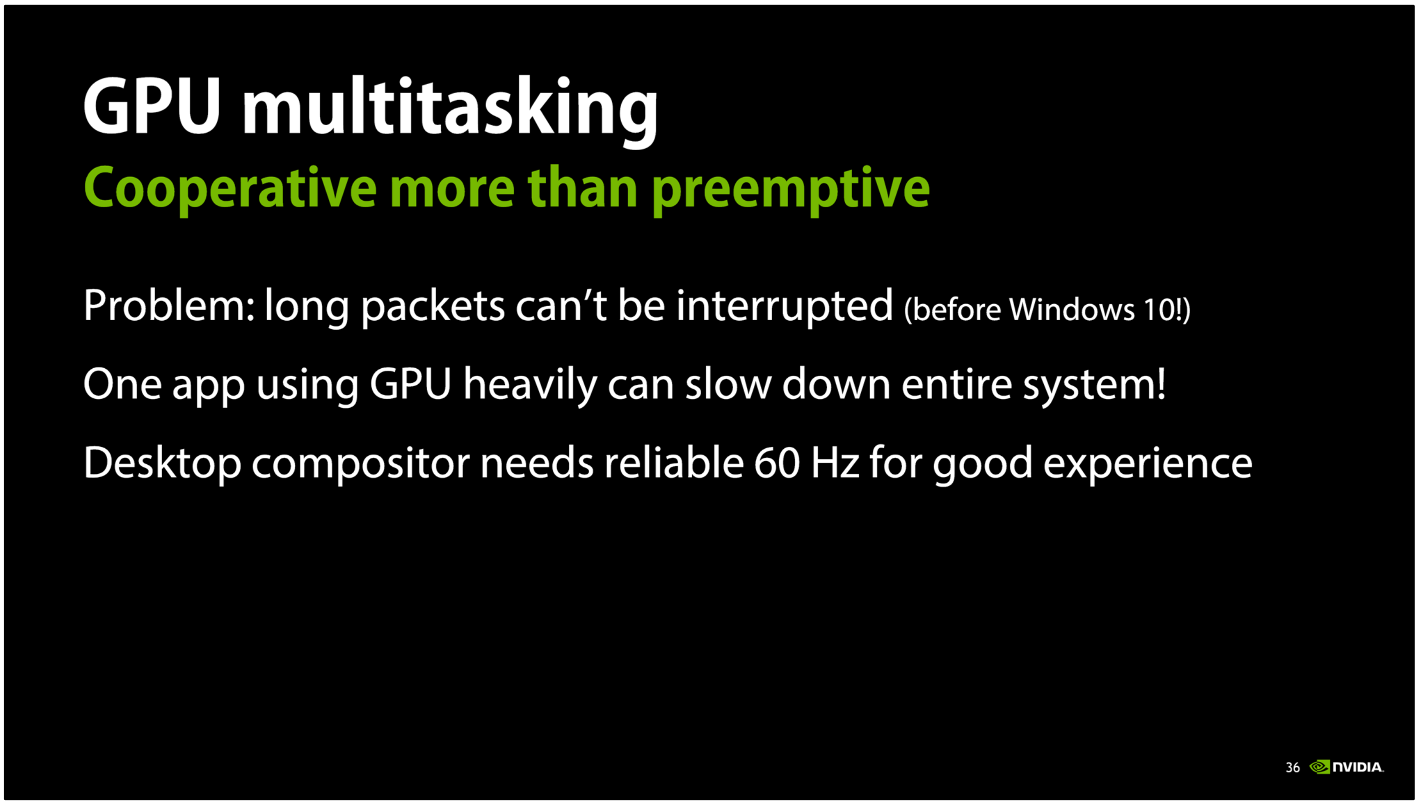
DX/GL contexts create command packets

Windows enqueues packets for GPU to execute

35

However, on a PC we may have many apps running that want to use the GPU—not just 3D apps and games, but web browsers, video players and other innocuous applications. And, importantly, on Windows the desktop compositor also uses the GPU to render your desktop and all your application windows and in some cases their UI. We'll come back to that.

Much like the operating system schedules threads from different applications to run on a CPU core, it also schedules DX/GL rendering contexts from different applications to run on the GPU. As apps issue DX/GL API calls, the driver batches them together into packets of hardware-level GPU commands, and eventually submits them to the operating system. Usually there's a small number of packets per frame, per application. The Windows GPU scheduler takes the packets from all the apps, orders them and queues them up for the GPU to execute, ensuring that every application gets a chance to use the GPU.



GPU multitasking

Cooperative more than preemptive

Problem: long packets can't be interrupted (before Windows 10!)

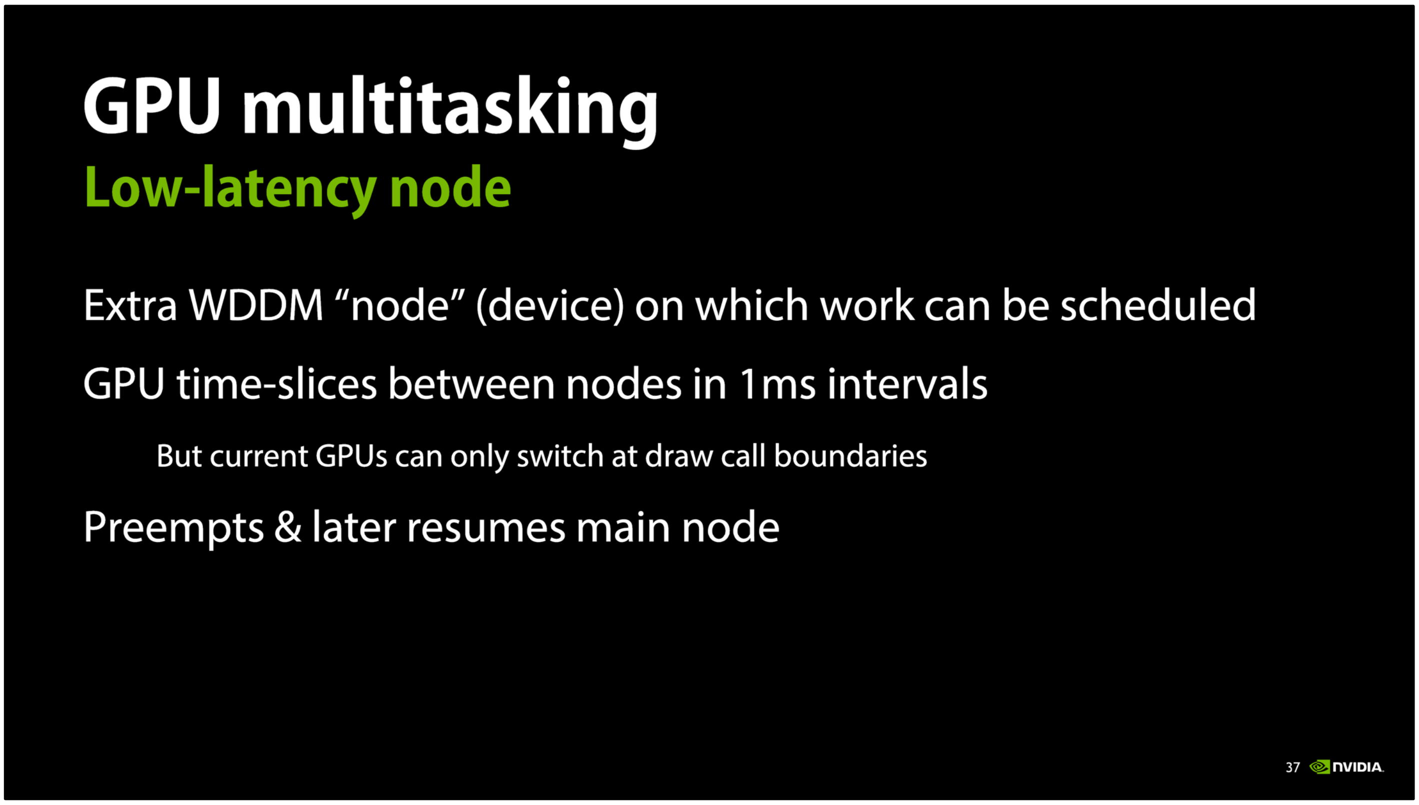
One app using GPU heavily can slow down entire system!

Desktop compositor needs reliable 60 Hz for good experience

36

One problem with this is that prior to Windows 10, a command packet once submitted to the GPU can't be interrupted. This means that an app that's using the GPU heavily, producing many long-running command packets, could starve other apps of GPU access—even if those apps are very lightweight.

This particularly affects the desktop compositor. Desktop composition is not an expensive operation, but it needs to happen reliably at 60 Hz (or whatever your monitor's refresh rate is) to ensure a good experience when manipulating windows, switching between apps and so forth. If you have a heavy rendering app running at 5fps, we don't want everything else on your system to slow down to 5fps as well.



GPU multitasking

Low-latency node

Extra WDDM “node” (device) on which work can be scheduled

GPU time-slices between nodes in 1ms intervals

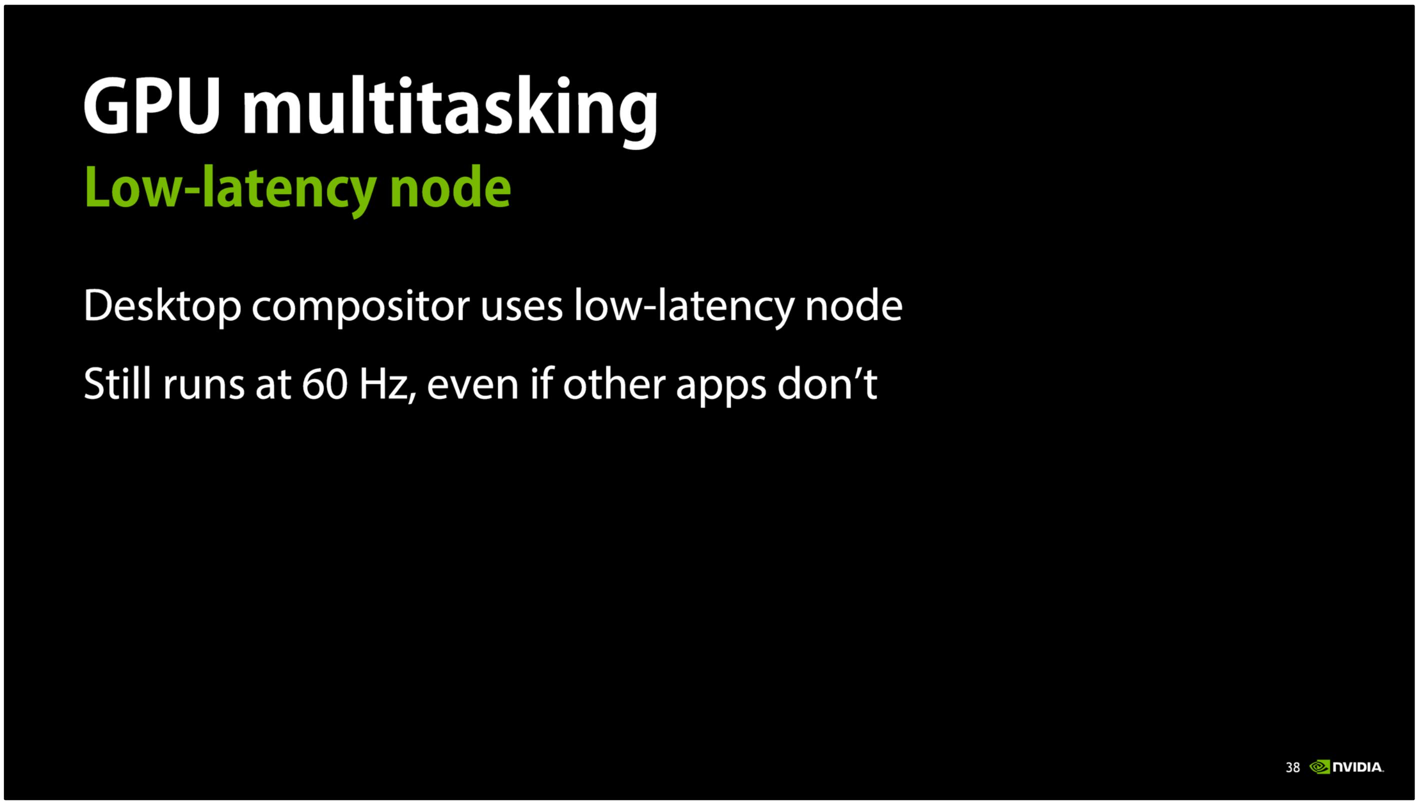
- But current GPUs can only switch at draw call boundaries

Preempts & later resumes main node

37

The way this is solved prior to Windows 10 is that we have an extra WDDM “node”, basically a thing that presents itself to Windows as a device on which graphics command packets can be scheduled. This secondary, “low-latency” node maps to the same GPU, but it’s logically separate from the main GPU graphics node.

At the hardware level, the GPU time-slices between the two nodes at (by default) 1ms intervals. There’s a slight caveat there, which is that currently they can only switch at draw call boundaries, so it’s really 1ms rounded up by the length of the last draw call. But the point is that we can submit work on the low-latency node, and within about 1ms, the GPU will switch to it, effectively preempting whatever command packet is running on the main node. Once the work on the low-latency node is finished, we’ll resume the main node.



GPU multitasking

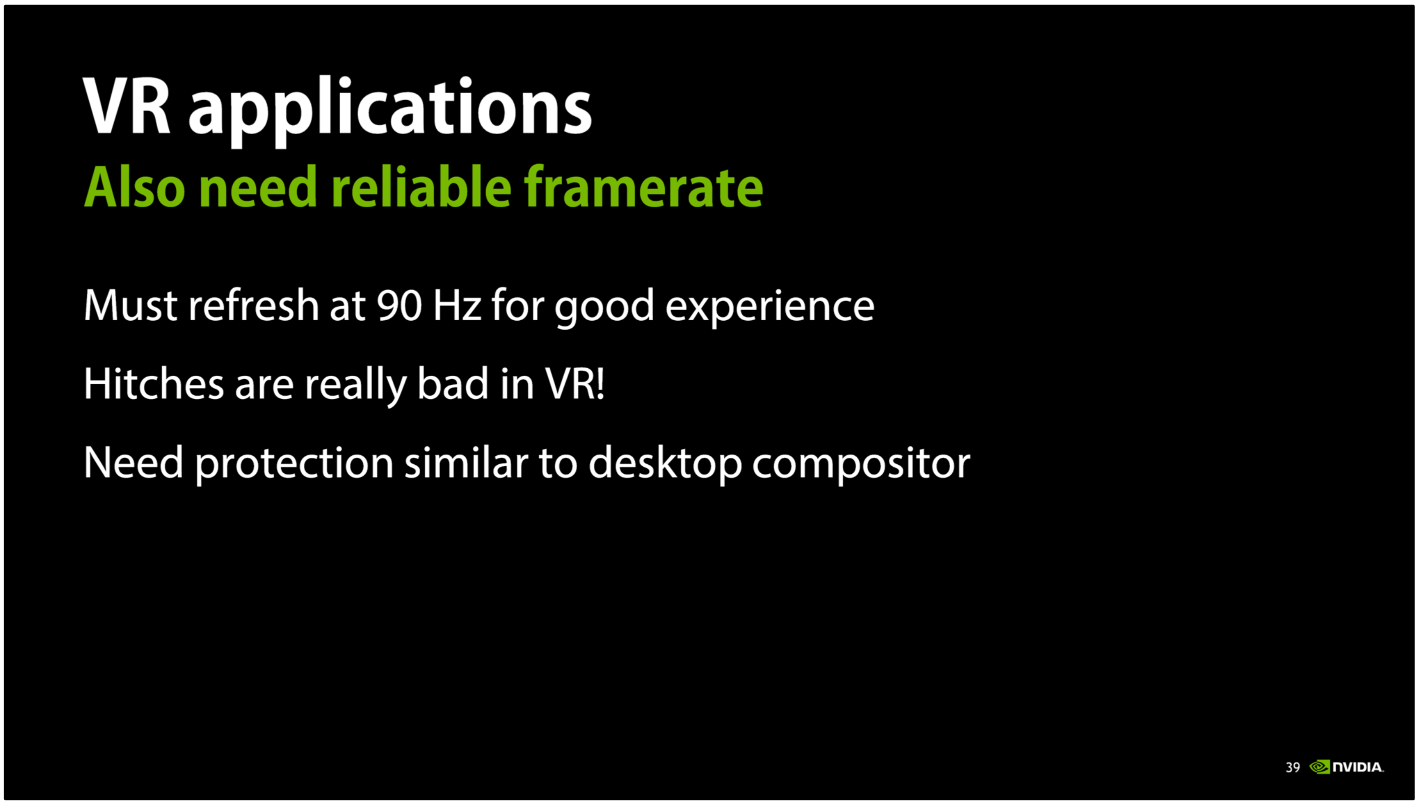
Low-latency node

Desktop compositor uses low-latency node

Still runs at 60 Hz, even if other apps don't

38

The upshot of all this is that the desktop compositor submits its work to the low-latency node, to help ensure that it gets a chance to run regularly even if other apps are using the GPU heavily.



VR applications

Also need reliable framerate

Must refresh at 90 Hz for good experience

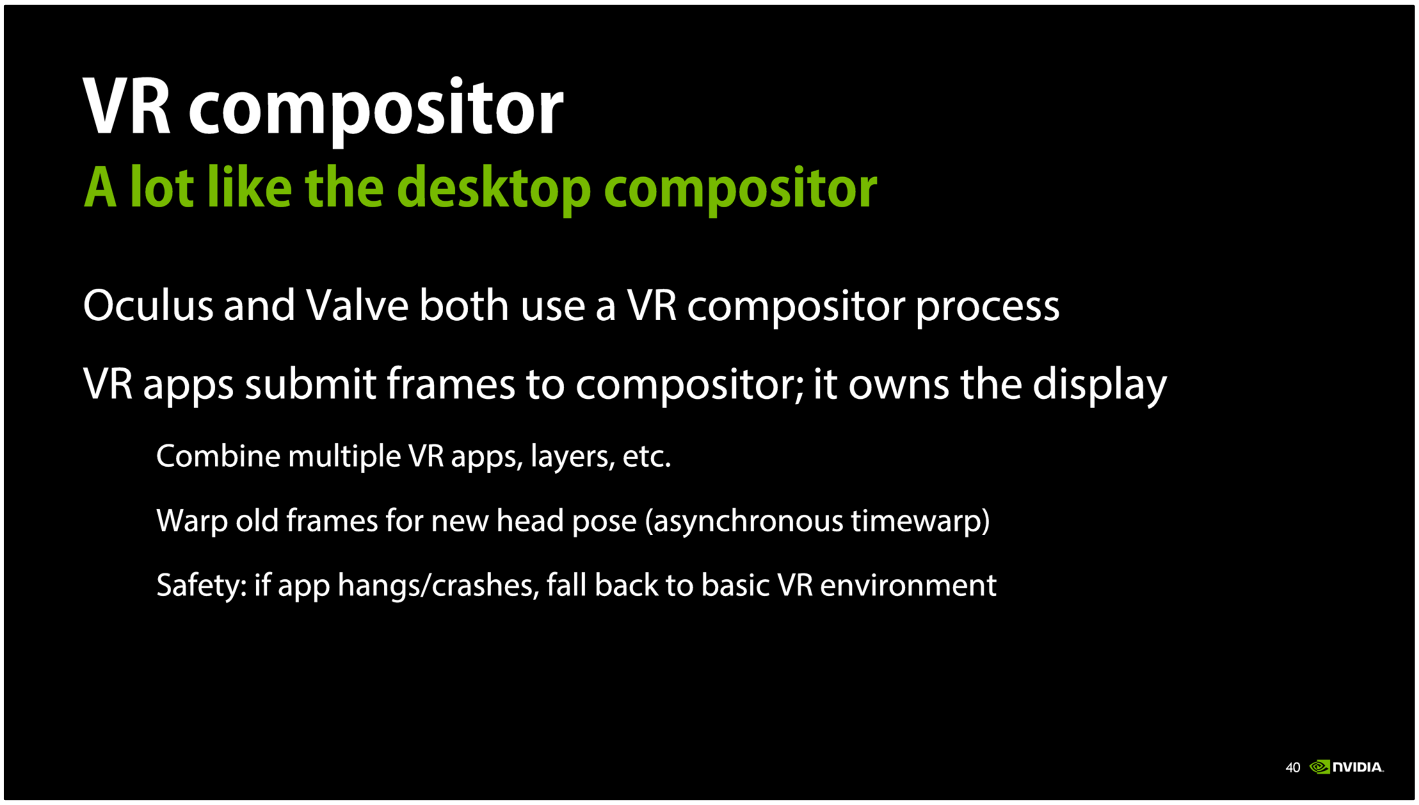
Hitches are really bad in VR!

Need protection similar to desktop compositor

39

Okay, so what does all of this have to do with VR? Well, similarly to the desktop compositor, we want VR applications to reliably render at 90 Hz, or whatever the native framerate of the headset is, to ensure a good experience. In fact, this is even more important in VR because a rendering hitch means that you lose head-tracking and have a frame stuck to your face. It instantly causes disorientation and, if hitches happen too often or for too long, it can lead to physical discomfort.

So, VR apps need a level of protection from other apps on the system, similar to the desktop compositor.


VR compositor
A lot like the desktop compositor

Oculus and Valve both use a VR compositor process

VR apps submit frames to compositor; it owns the display

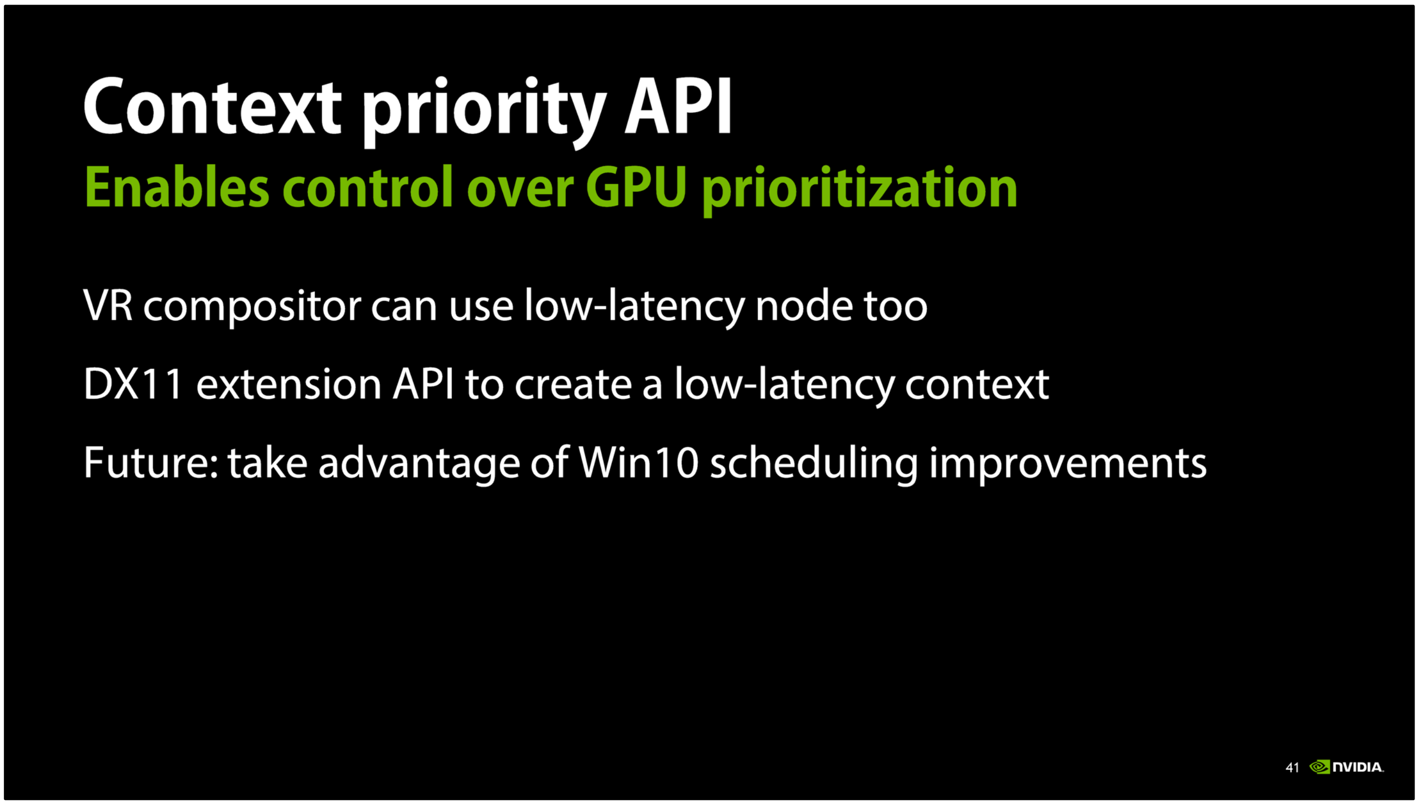
- Combine multiple VR apps, layers, etc.
- Warp old frames for new head pose (asynchronous timewarp)
- Safety: if app hangs/crashes, fall back to basic VR environment

40

In fact, the analogy with the desktop compositor goes further. Oculus and Valve both have, in their software stack that you use to talk to their headsets, a VR compositor process. In their systems, VR apps don't present to the headset directly; instead, they submit rendered frames to the compositor, which owns and manages the actual display. It's very much parallel to the way windowed apps submit frames to the desktop compositor.

There are a variety of reasons for this. For instance, it allows compositing multiple VR apps, or multiple "layers" of different framerate and resolution. You can imagine playing a VR game, while also having a VR chat window open to talk to your friends. Or a virtual office in which various apps are rendering objects into your workspace. Things like that.

The VR compositor also acts as a safety net; if an app misses a frame, the compositor can re-present the old frame and warp it for an updated head pose, so that you still have head-tracking during short hitches. That's called asynchronous timewarp. And in an extreme case, if an app crashes or hangs, the VR compositor can detect that and soft-fail back to some basic VR environment, like a plain room, so that you aren't disoriented.



Context priority API

Enables control over GPU prioritization

VR compositor can use low-latency node too

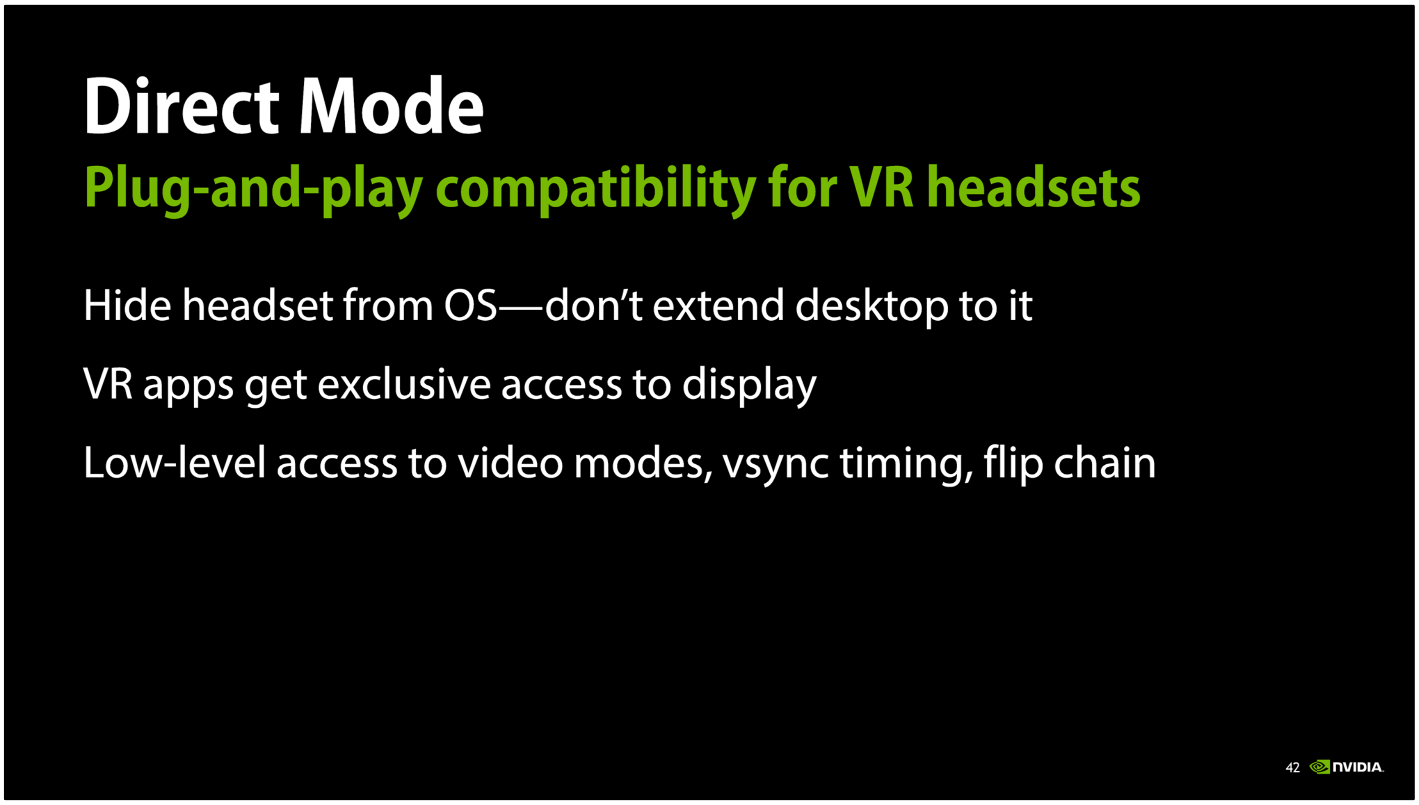
DX11 extension API to create a low-latency context

Future: take advantage of Win10 scheduling improvements

41

So, to make a very long story short, as part of NVIDIA's VR toolkits we expose an API that enables apps to opt-in to running on the low-latency node. So headset developers who have a VR compositor can give it the same prioritization that the normal desktop compositor gets. This is in the form of a DX11 extension that enables you to create a special low-latency DX context, which will submit to the low-latency node.

In the future, we'll also extend this API to take advantage of the GPU scheduling and preemption improvements in Windows 10.



Direct Mode

Plug-and-play compatibility for VR headsets

Hide headset from OS—don’t extend desktop to it

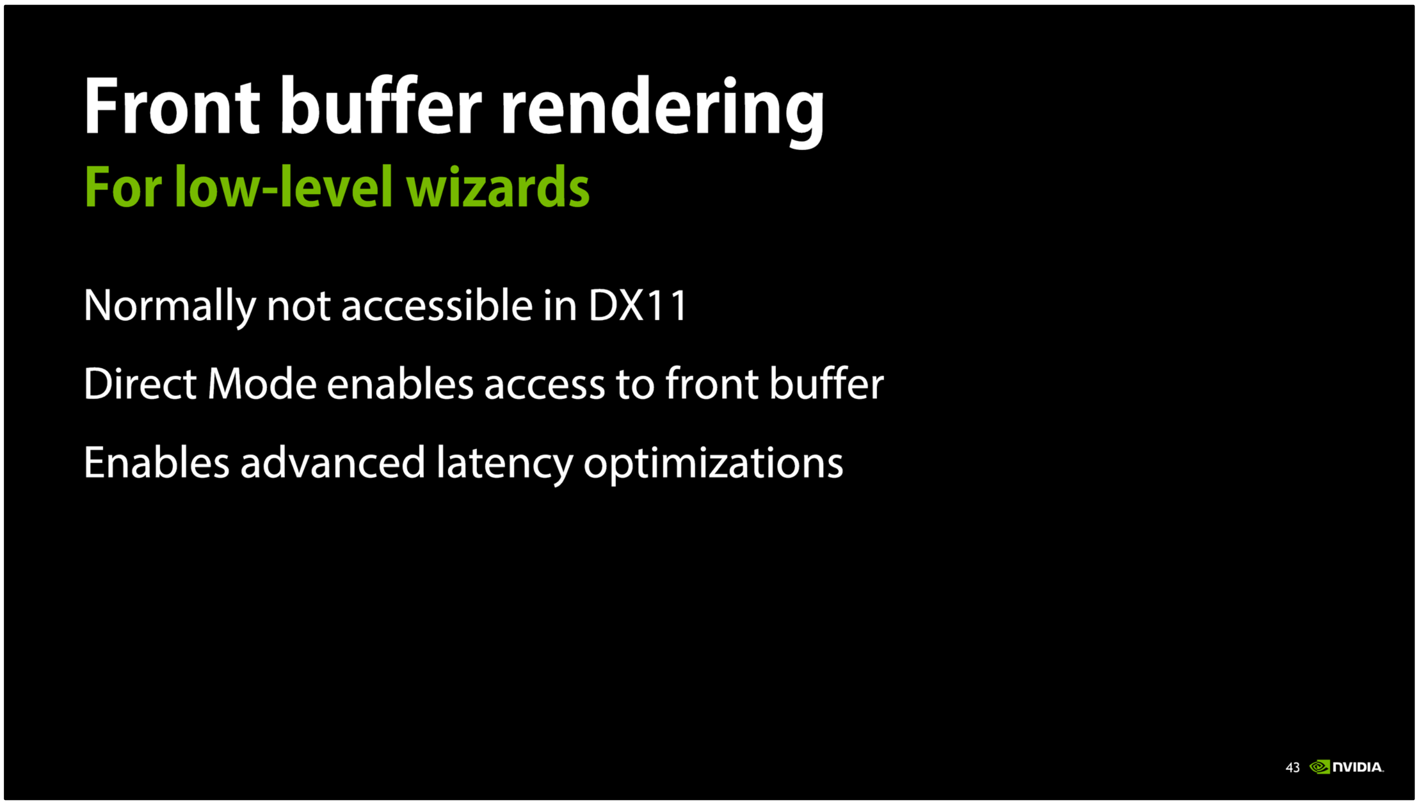
VR apps get exclusive access to display

Low-level access to video modes, vsync timing, flip chain

42

Direct Mode is another feature targeted at VR headset vendors running on Windows. It enables our driver to recognize when a display that’s plugged in is a VR headset, as opposed to a monitor, and hide it from the OS so that the OS doesn’t try to extend the desktop onto the headset or move your windows onto it. Then VR apps can take over the display and render to it directly, getting completely exclusive access to the display with no OS compositor or anything interfering.

Our Windows Direct Mode API also provides low-level access to information about video modes, vsync timing, and control over the flip chain. VR software can manage those details itself to ensure the lowest possible latency between rendering and scan-out to the display. All in all, this just provides a better user experience in general for working with VR.



Front buffer rendering

For low-level wizards

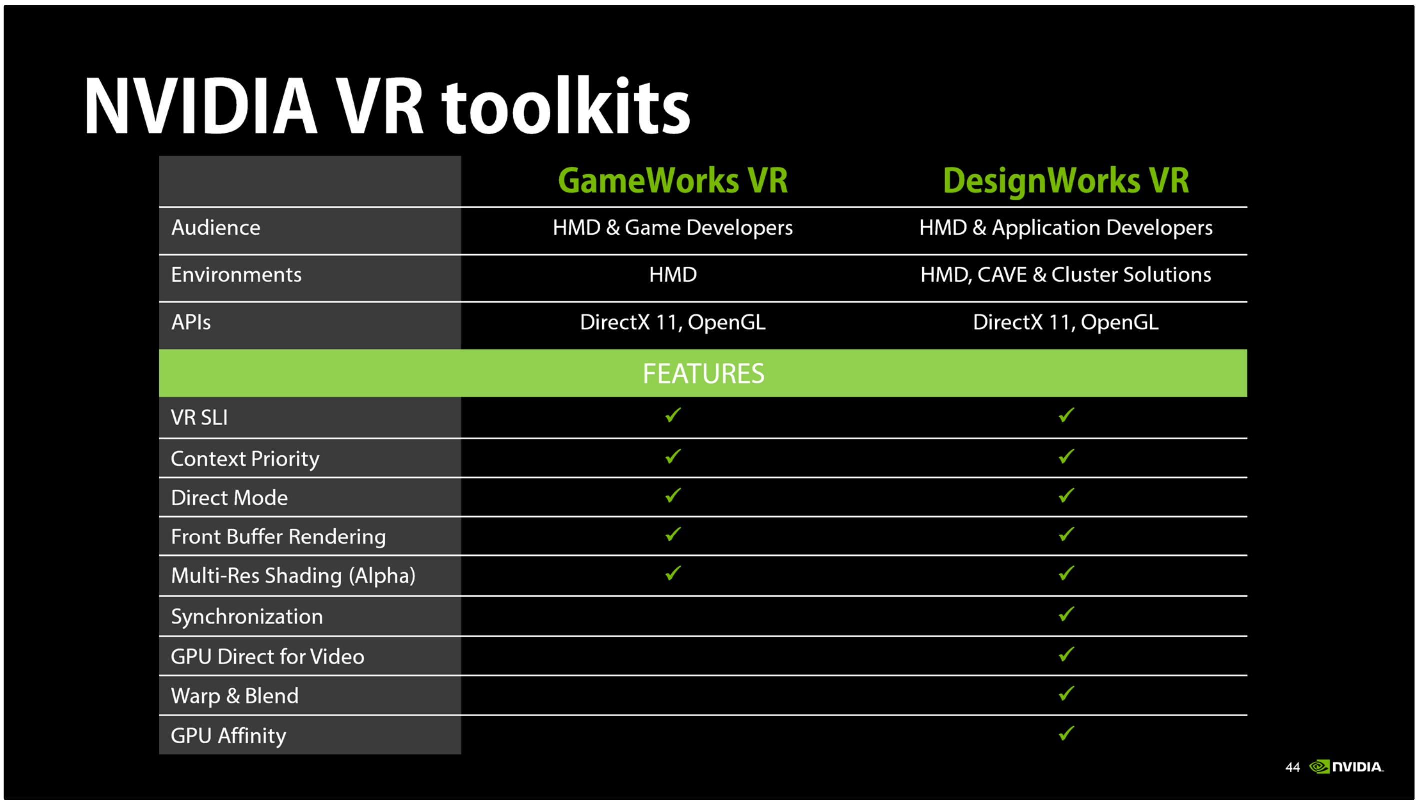
Normally not accessible in DX11

Direct Mode enables access to front buffer

Enables advanced latency optimizations

43

Another benefit of Direct Mode is that we can expose the ability to render directly to the front buffer—i.e. the buffer currently being scanned out to the display. Although it takes advanced low-level know-how to make use of this feature, it can help reduce latency still further, using tricks like rendering during vblank or racing the beam.



NVIDIA VR toolkits

	GameWorks VR	DesignWorks VR
Audience	HMD & Game Developers	HMD & Application Developers
Environments	HMD	HMD, CAVE & Cluster Solutions
APIs	DirectX 11, OpenGL	DirectX 11, OpenGL
FEATURES		
VR SLI	✓	✓
Context Priority	✓	✓
Direct Mode	✓	✓
Front Buffer Rendering	✓	✓
Multi-Res Shading (Alpha)	✓	✓
Synchronization		✓
GPU Direct for Video		✓
Warp & Blend		✓
GPU Affinity		✓

44

`pDevice->Flush();`

Questions?

nreed@nvidia.com

Slides will be posted: http://developer.nvidia.com