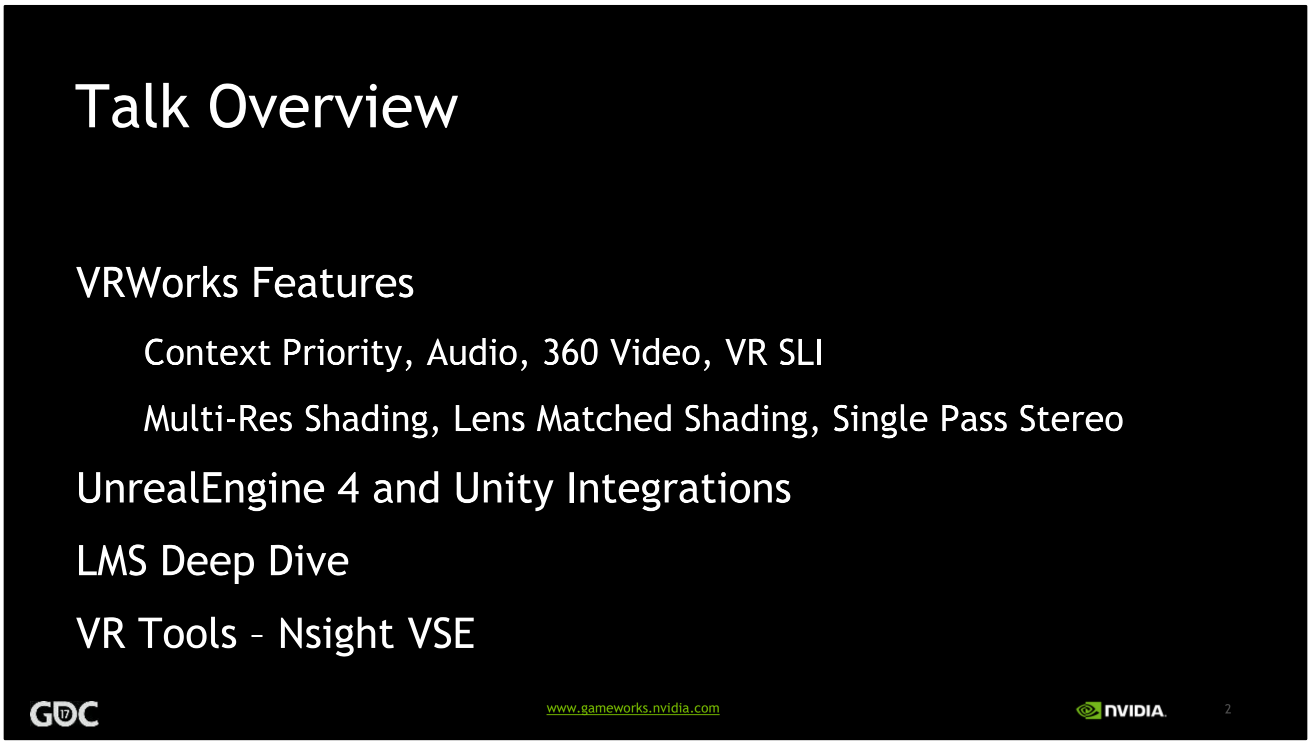
# Accelerating Your VR Games with VRWorks

Cem Cebenoyan
Edward Liu
Daniel Price



## Talk Overview

VRWorks Features

- Context Priority, Audio, 360 Video, VR SLI
- Multi-Res Shading, Lens Matched Shading, Single Pass Stereo

UnrealEngine 4 and Unity Integrations

LMS Deep Dive

VR Tools – Nsight VSE

www.gameworks.nvidia.com

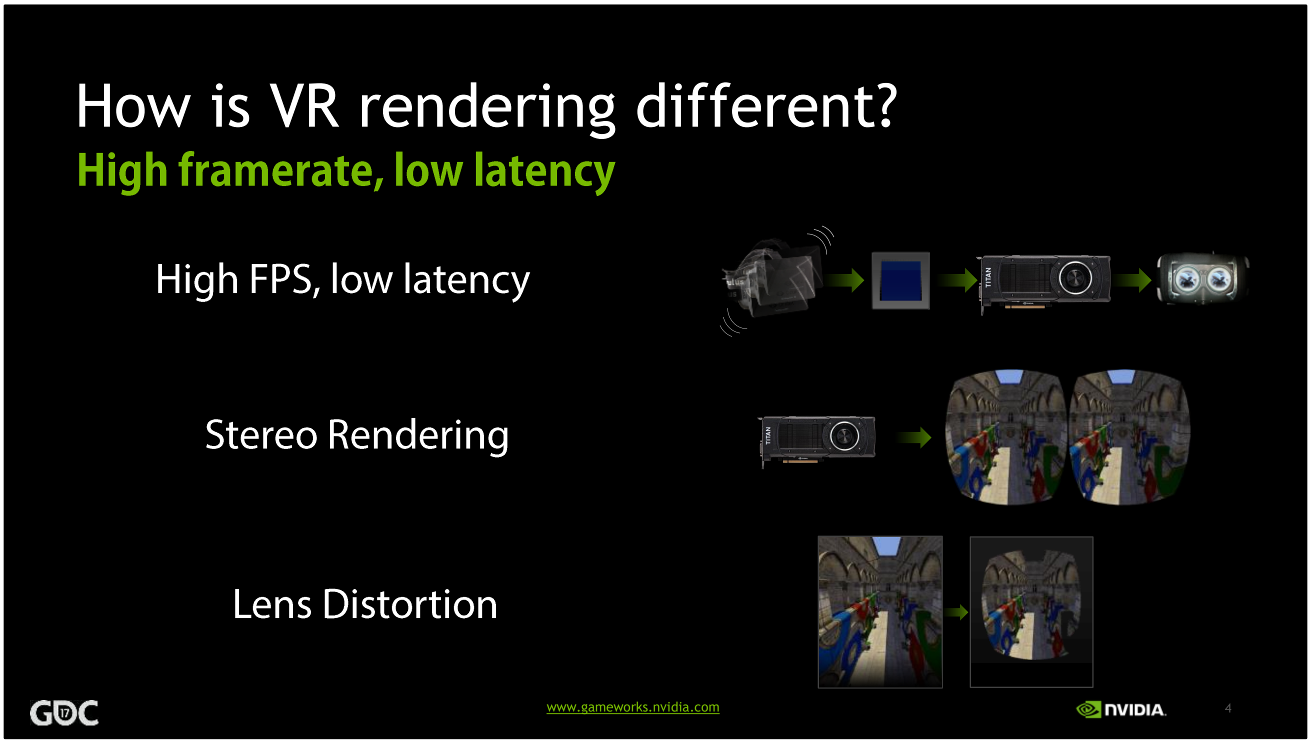
2

## How is VR rendering different?

www.gameworks.nvidia.com

3

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**

High FPS, low latency

Stereo Rendering

Lens Distortion

GDC

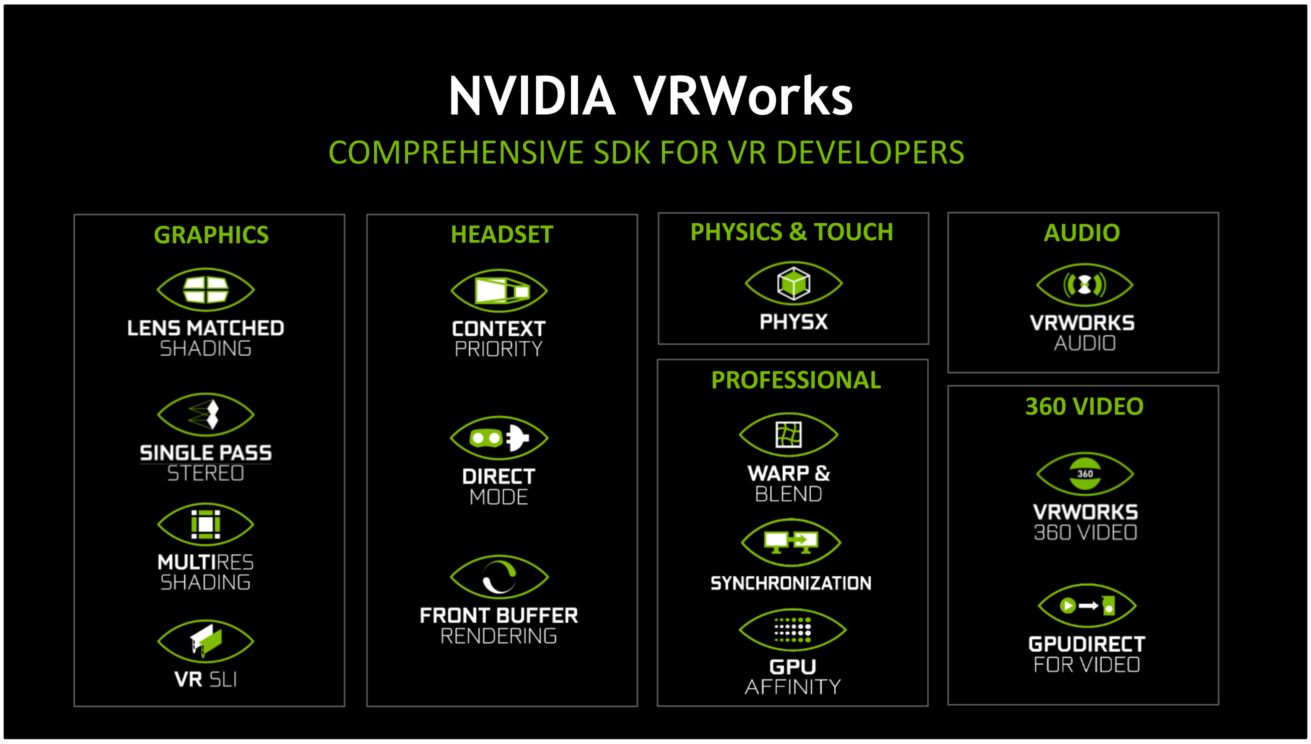
www.gameworks.nvidia.com

NVIDIA 4

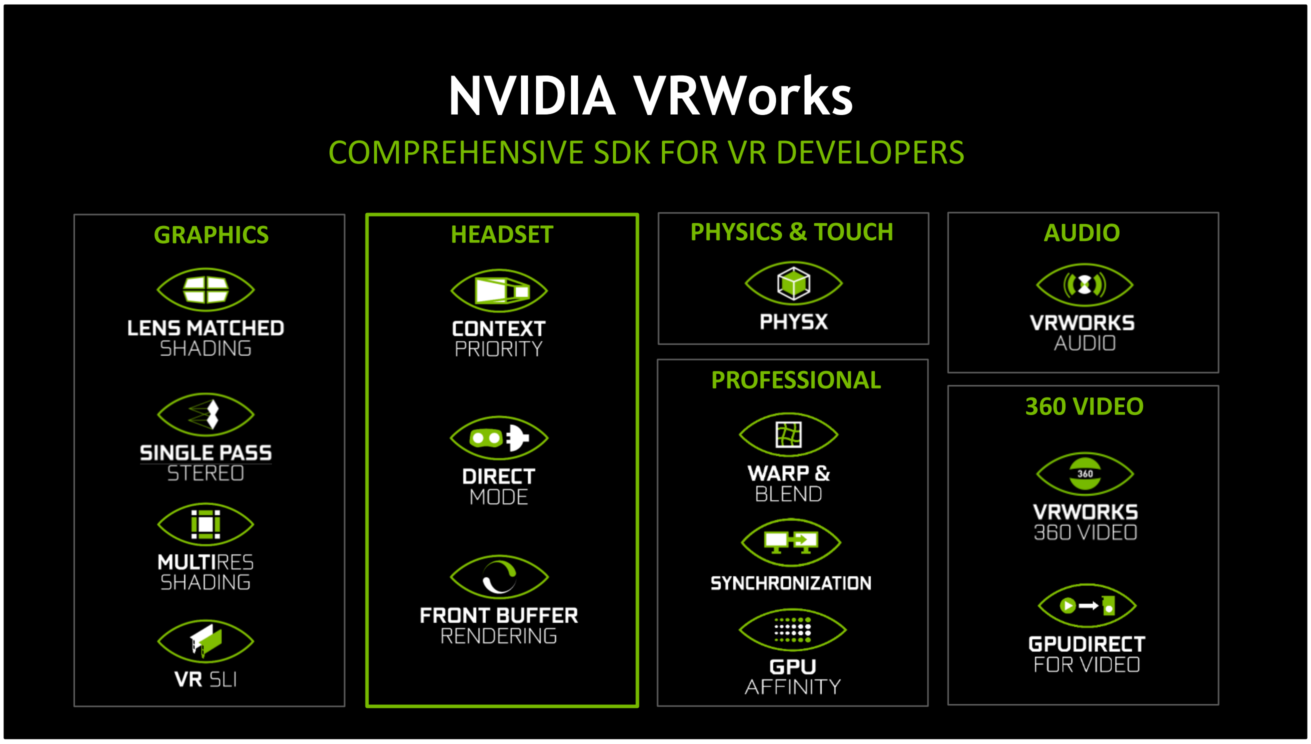
First, virtual reality is extremely demanding with respect to rendering performance. Both the Oculus Rift and HTC Vive headsets require 90 frames per second, which is much higher than the 60 fps that's usually considered the gold standard for real-time rendering.

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.



The new NVIDIA VRWorks adds graphics, audio, video, and physics simulation capabilities to the existing suite of HMD, graphics, and professional features.



## NVIDIA VRWorks

COMPREHENSIVE SDK FOR VR DEVELOPERS

GRAPHICS
- LENS MATCHED SHADING
- SINGLE PASS STEREO
- MULTIRES SHADING
- VR SLI

HEADSET
- CONTEXT PRIORITY
- DIRECT MODE
- FRONT BUFFER RENDERING

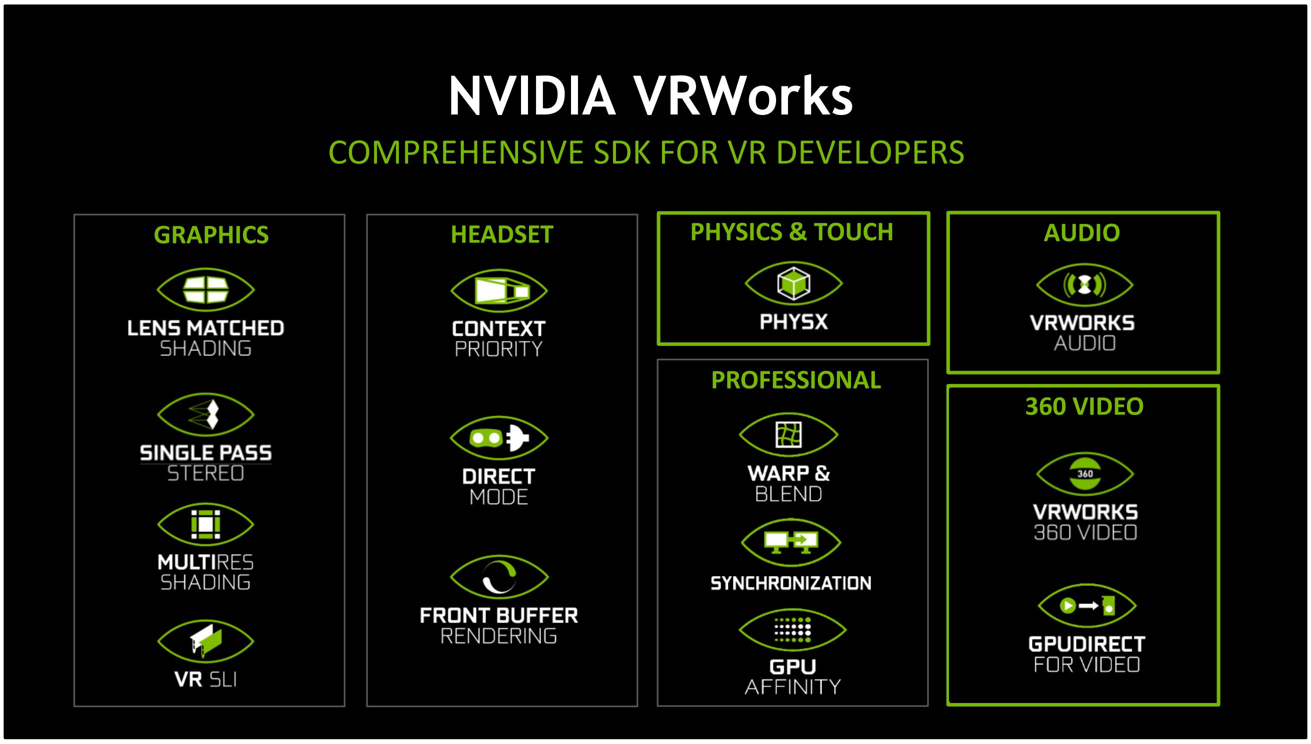
PHYSICS & TOUCH
- PHYSX

PROFESSIONAL
- WARP & BLEND
- SYNCHRONIZATION
- GPU AFFINITY

AUDIO
- VRWORKS AUDIO

360 VIDEO
- VRWORKS 360 VIDEO
- GPUDIRECT FOR VIDEO

The new NVIDIA VRWorks adds graphics, audio, video, and physics simulation capabilities to the existing suite of HMD, graphics, and professional features.


NVIDIA VRWorks

COMPREHENSIVE SDK FOR VR DEVELOPERS

GRAPHICS
LENS MATCHED SHADING
SINGLE PASS STEREO
MULTIRES SHADING
VR SLI

HEADSET
CONTEXT PRIORITY
DIRECT MODE
FRONT BUFFER RENDERING

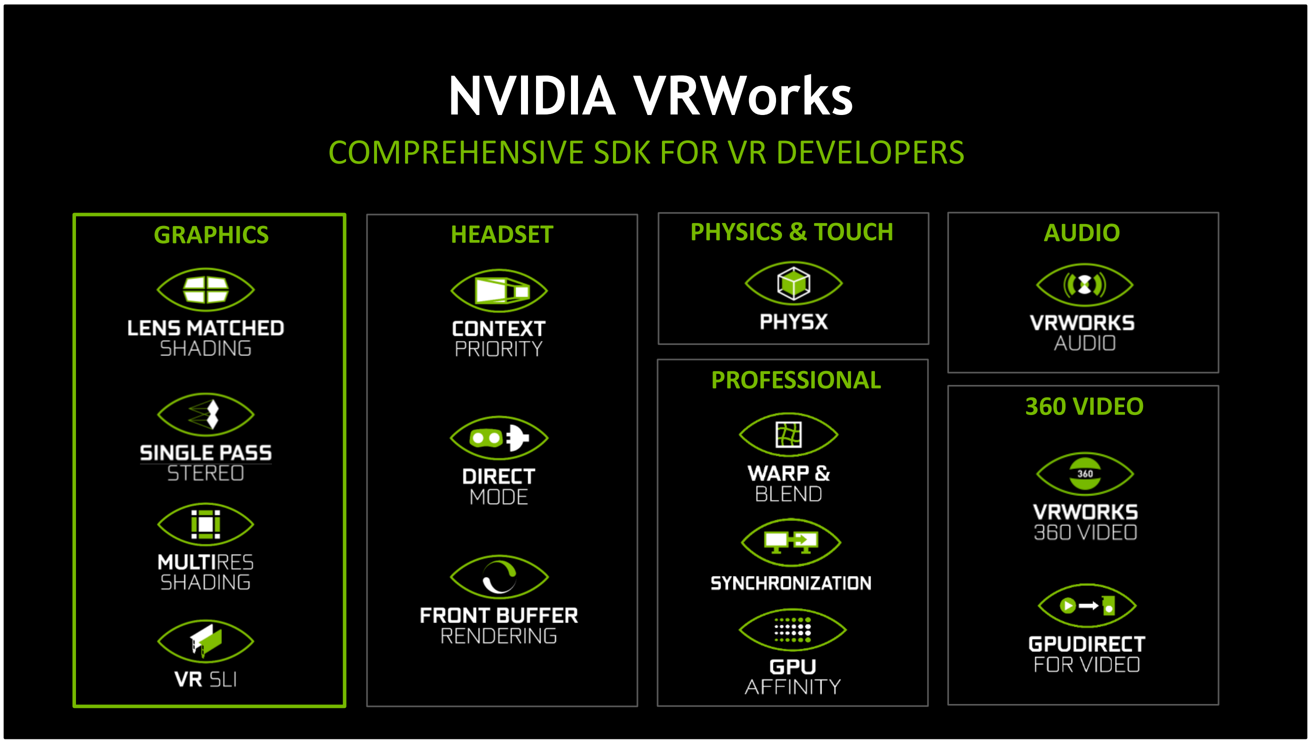
PHYSICS & TOUCH
PHYSX

PROFESSIONAL
WARP & BLEND
SYNCHRONIZATION
GPU AFFINITY

AUDIO
VRWORKS AUDIO

360 VIDEO
VRWORKS 360 VIDEO
GPUDIRECT FOR VIDEO

The new NVIDIA VRWorks adds graphics, audio, video, and physics simulation capabilities to the existing suite of HMD, graphics, and professional features.



NVIDIA VRWorks

COMPREHENSIVE SDK FOR VR DEVELOPERS

GRAPHICS

LENS MATCHED SHADING

SINGLE PASS STEREO

MULTIRES SHADING

VR SLI

HEADSET

CONTEXT PRIORITY

DIRECT MODE

FRONT BUFFER RENDERING

PHYSICS & TOUCH

PHYSX

PROFESSIONAL

WARP & BLEND

SYNCHRONIZATION

GPU AFFINITY

AUDIO

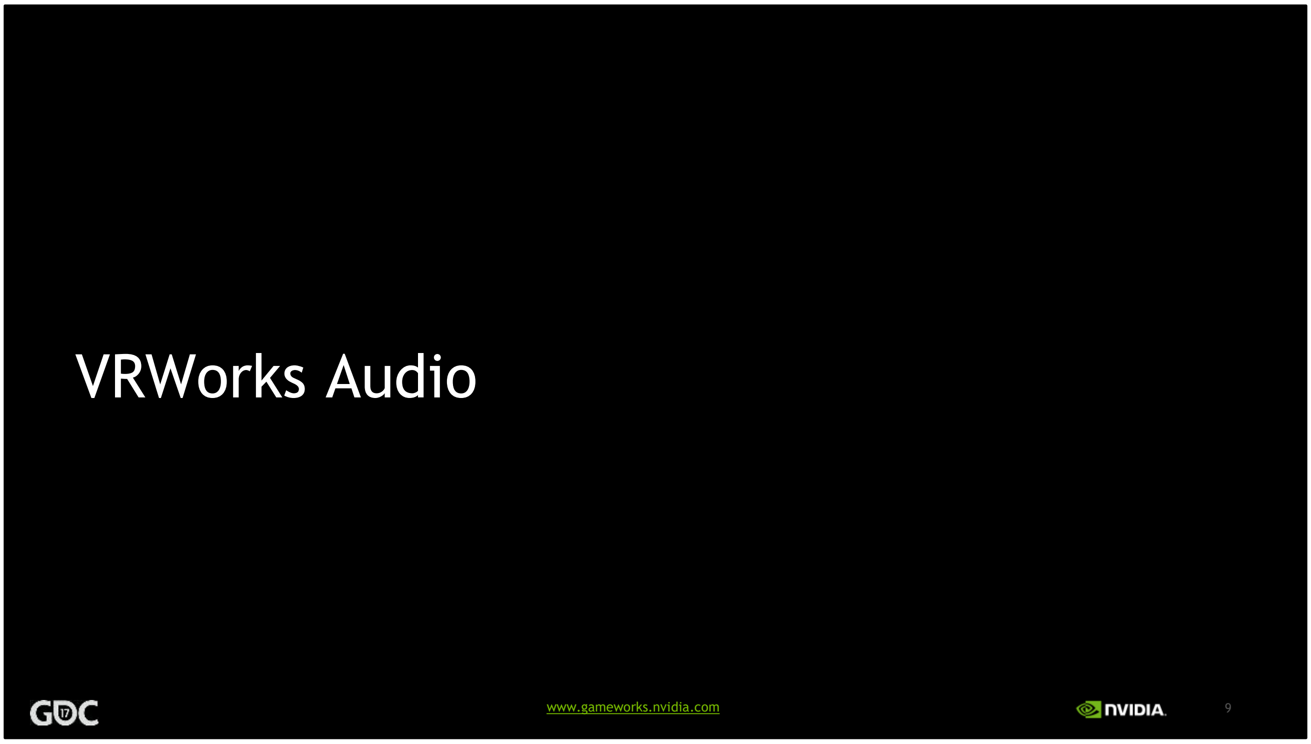
VRWORKS AUDIO

360 VIDEO

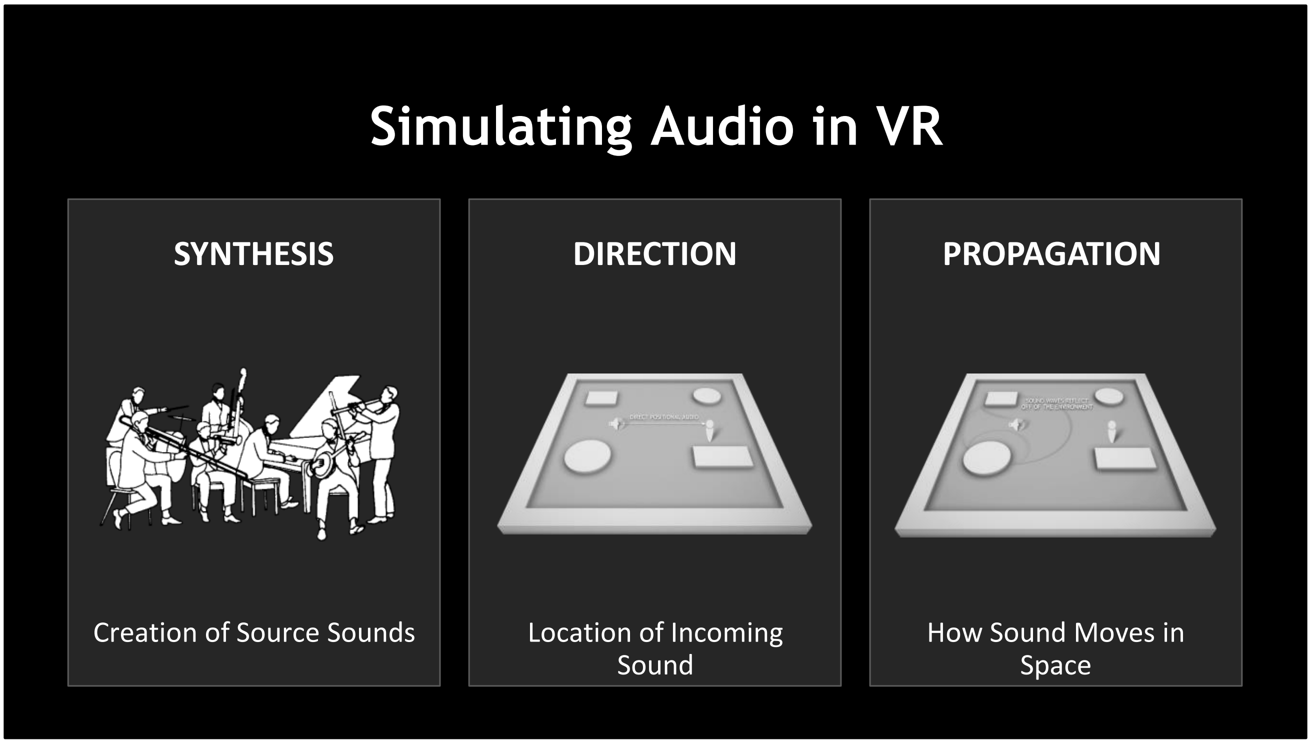
VRWORKS 360 VIDEO

GPUDIRECT FOR VIDEO

The new NVIDIA VRWorks adds graphics, audio, video, and physics simulation capabilities to the existing suite of HMD, graphics, and professional features.



## VRWorks Audio



## Simulating Audio in VR

**SYNTHESIS**

Creation of Source Sounds

**DIRECTION**

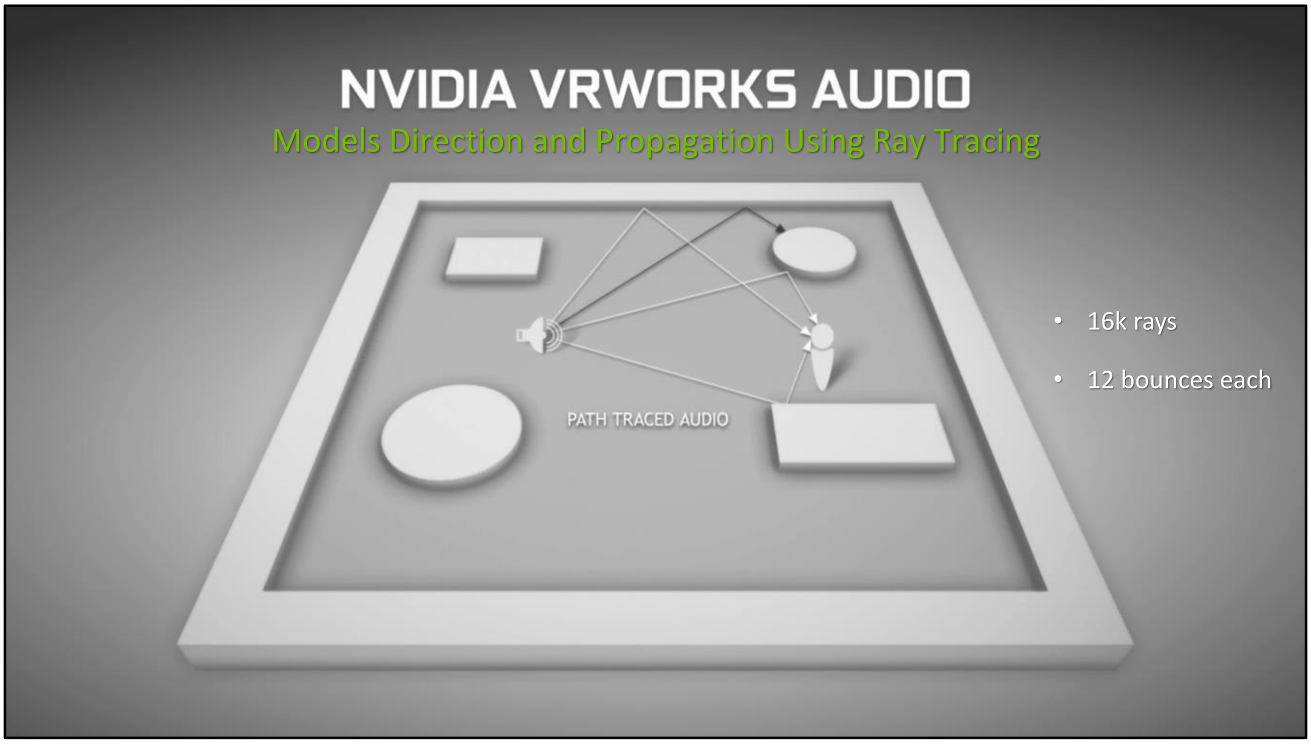
Location of Incoming Sound

**PROPAGATION**

How Sound Moves in Space

Great VR graphics demands great VR audio.

Today, state of the art VR audio provides accurate 3D direction of sound in a virtual environment – you can hear where the sound is coming from.  However, there is more to sound than just location – there is also how audio propagates and reflects off the surrounding virtual environment – today's VR audio doesn't account for this.



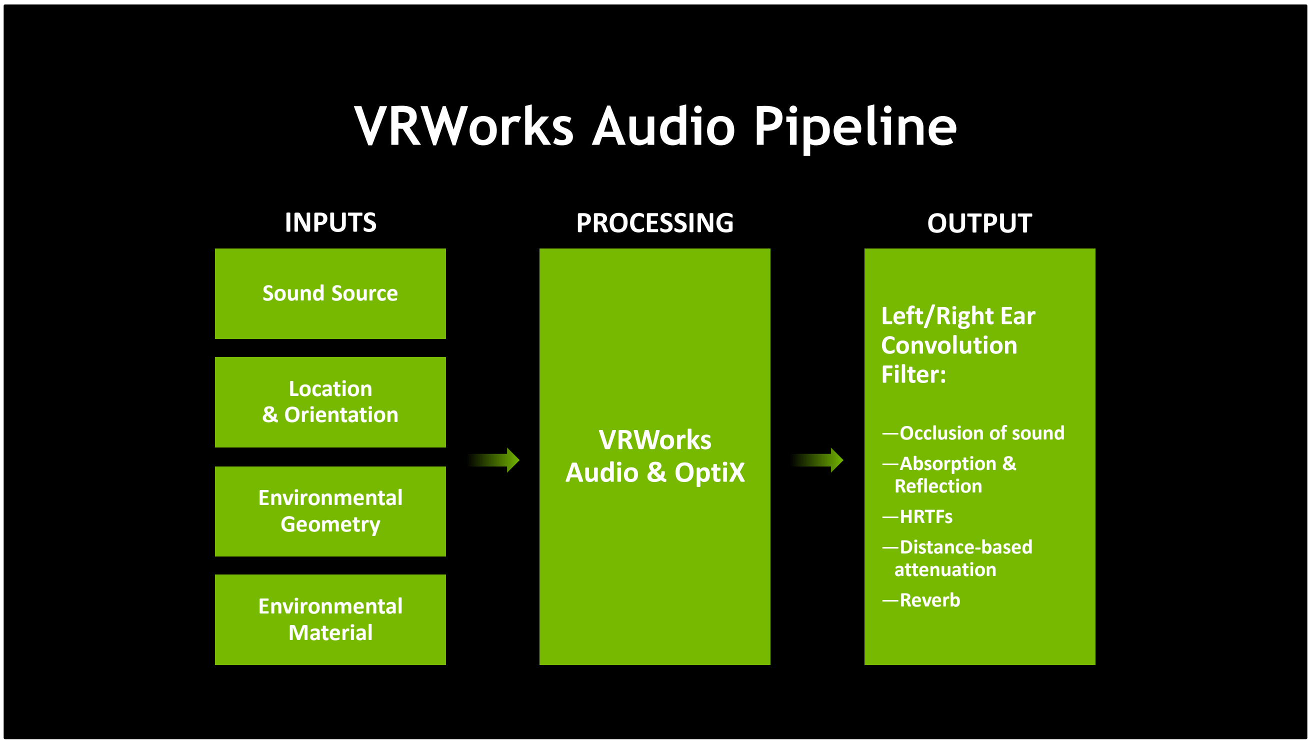
## NVIDIA VRWORKS AUDIO

Models Direction and Propagation Using Ray Tracing

PATH TRACED AUDIO

- 16k rays
- 12 bounces each

VRWorks Audio works by following the path of audio as it moves around an environment using a technique called ray tracing.  Ray tracing models the wave propagation as it bounces and reflects off walls until it finally reaches the listener. Thanks to NVIDIA's OptiX ray tracing engine and the power of Pascal GPUs, NVIDIA can calculate 16,000 individual rays at 12 bounces each to get amazingly realistic audio in VR.



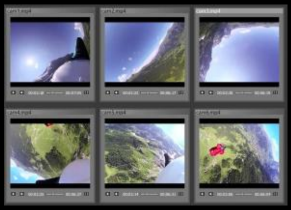
Here is how VRWorks Audio works in detail:

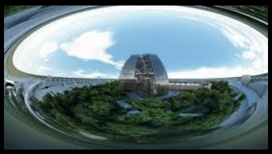
- The engine takes as inputs the audio source, the location and orientation of the sound, the geometry of the environment (walls, etc) and objects in the scene, and the material properties (wood vs. cement wall, etc) of the environment.
- The VRWorks audio SDK and the OptiX ray tracing engine calculates the path of audio as it propagates throughout the environment
- It then outputs a convolution filter that represents the sound propagation and modifies the source audio stream with reverb, HRTFs, etc

## VRWorks 360 Video

www.gameworks.nvidia.com

## Significant computation required to deliver 360 video





**Capture**
4k cameras

**Stitch**
Decode→Calibrate→Equalize→Stitch→Encode

**Display**
Single 360 video

## Introducing VRWORKS 360 VIDEO

CAPTURE, STITCH, & STREAM
360º VIDEOS IN REAL-TIME

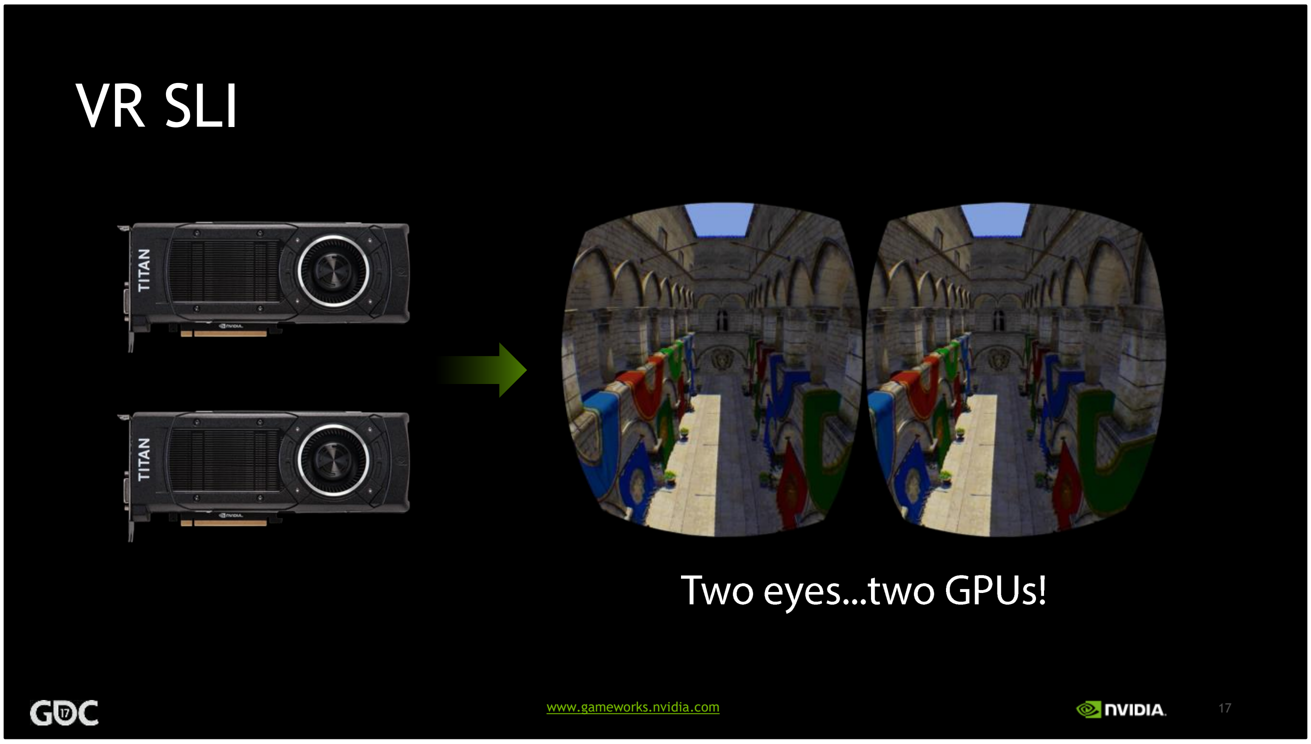
- Real-time and offline stitching from 4k camera rigs
- GPU-accelerated video decode, calibration, equalization, stitching, and encode
- 360 projection onto cube-map and equi-rectangular panorama
- Works with GPUDirect for Video for low latency video ingest

*"Capturing and stitching 360 video is time consuming and computationally demanding. NVIDIA's VRWorks 360 Video SDK will help accelerate STRIVR's workflows, delivering real-time, high quality 360 video."*

*— Masaki Miyanohara, CTO, STRIVR*

## VR SLI

www.gameworks.nvidia.com



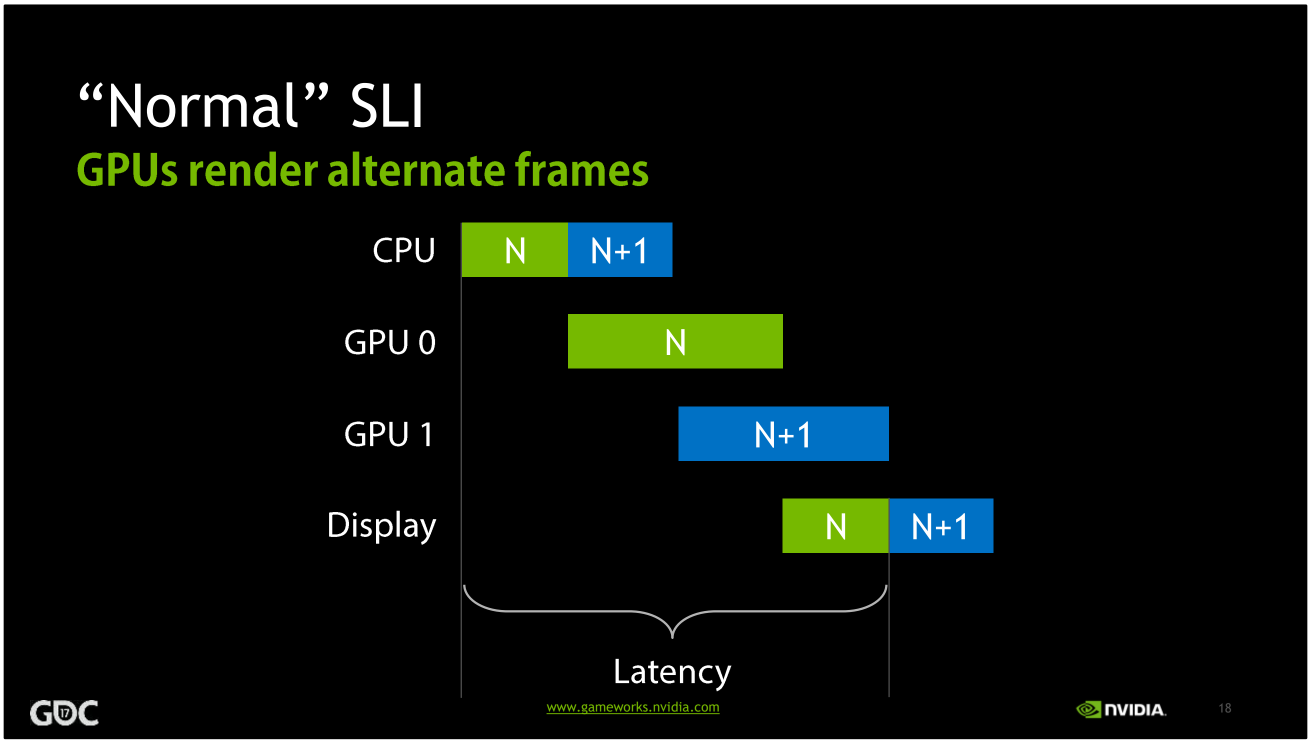
## VR SLI

Two eyes…two GPUs!

GDC www.gameworks.nvidia.com NVIDIA 17

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.



# "Normal" SLI

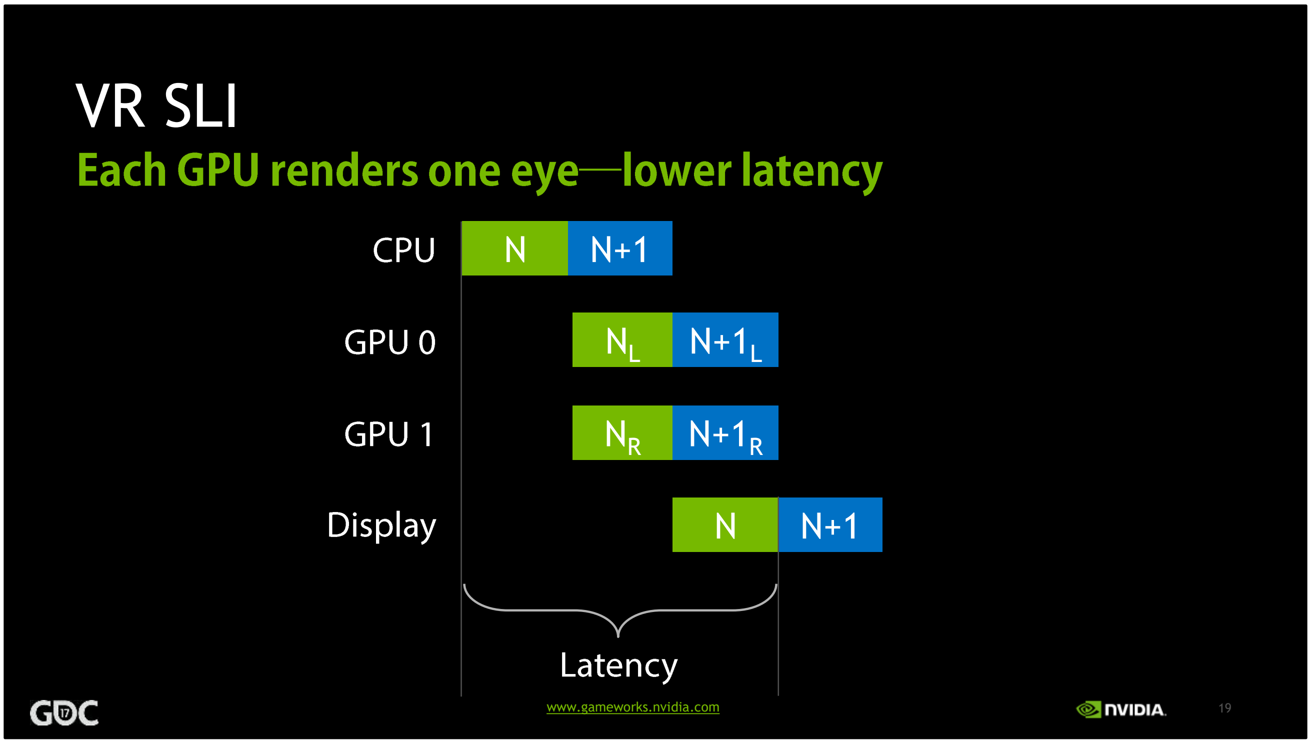
## GPUs render alternate frames

www.gameworks.nvidia.com

18

Before we dig into VR SLI, as a quick interlude, let me first explain how "normal", non-VR SLI 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.

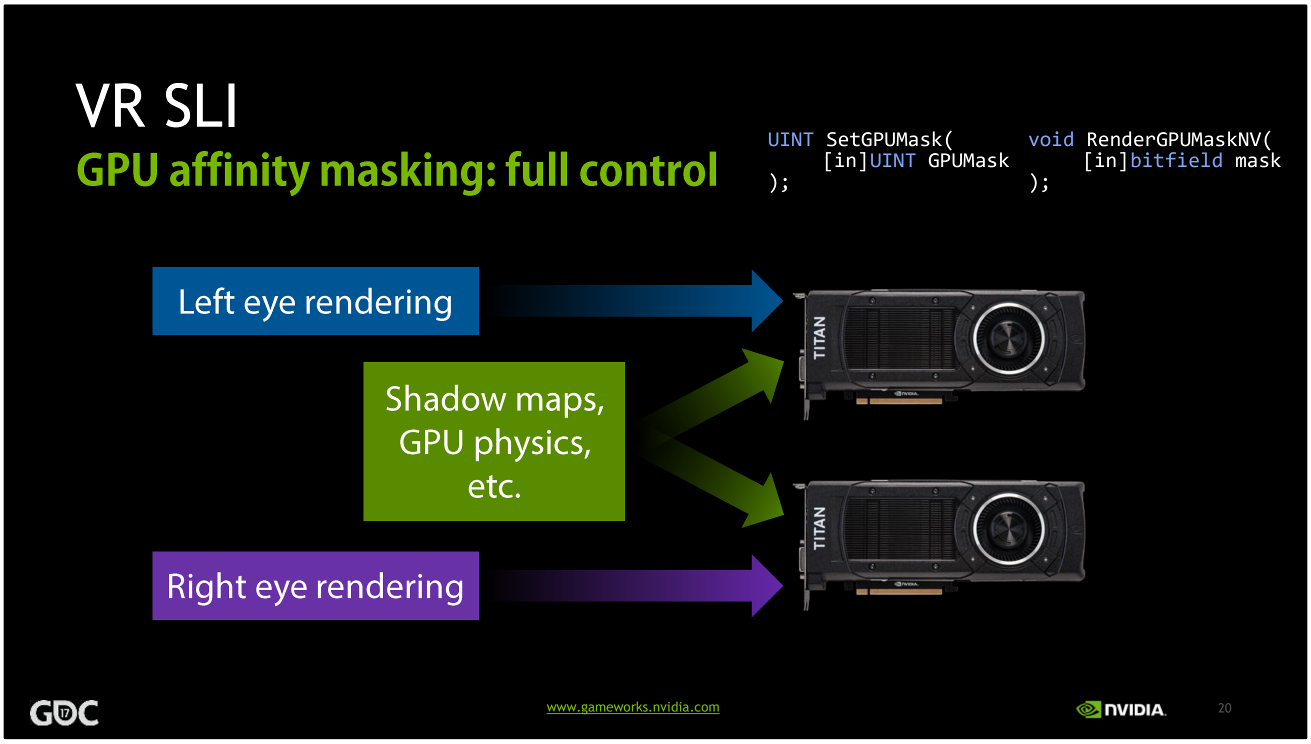
This works well to increase framerate relative to a single-GPU system, but it doesn't really help with latency. So this isn't the best model for VR.



## 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

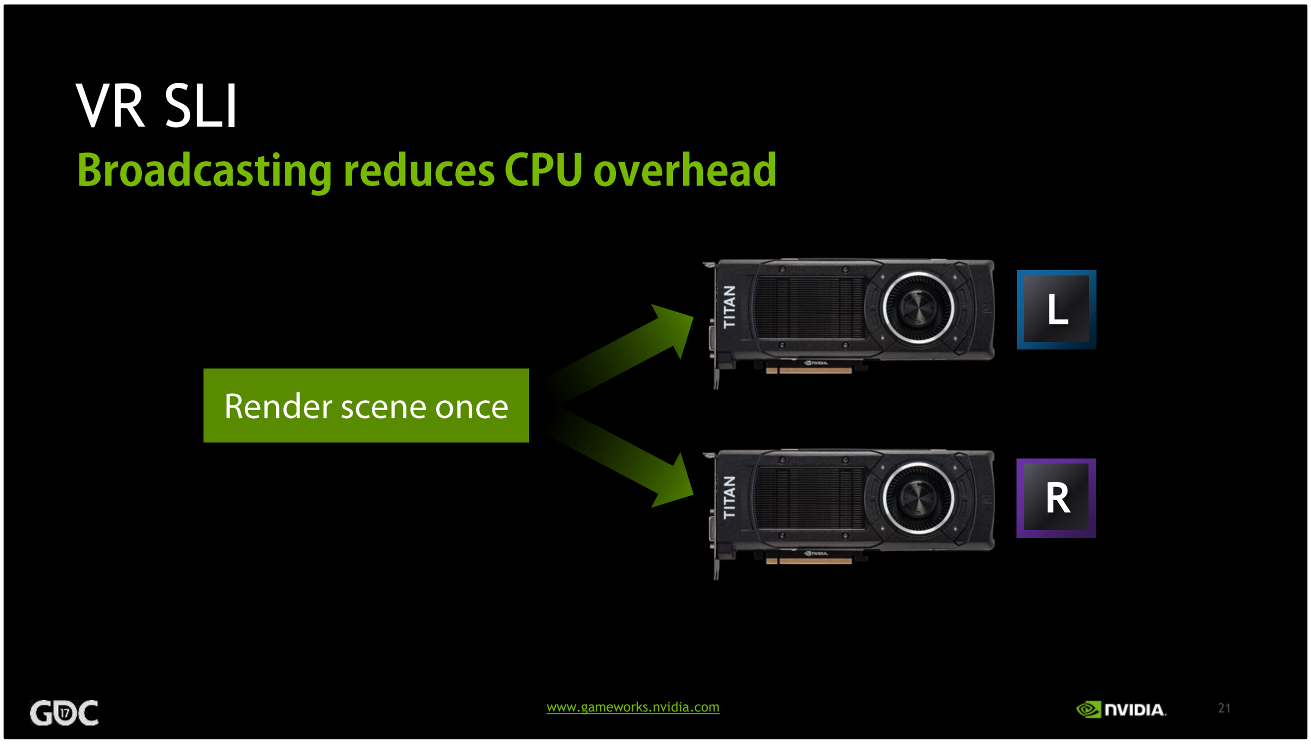
```
UINT SetGPUMask(
        [in]UINT GPUMask
);
```

```
void RenderGPUMaskNV(
        [in]bitfield mask
);
```

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

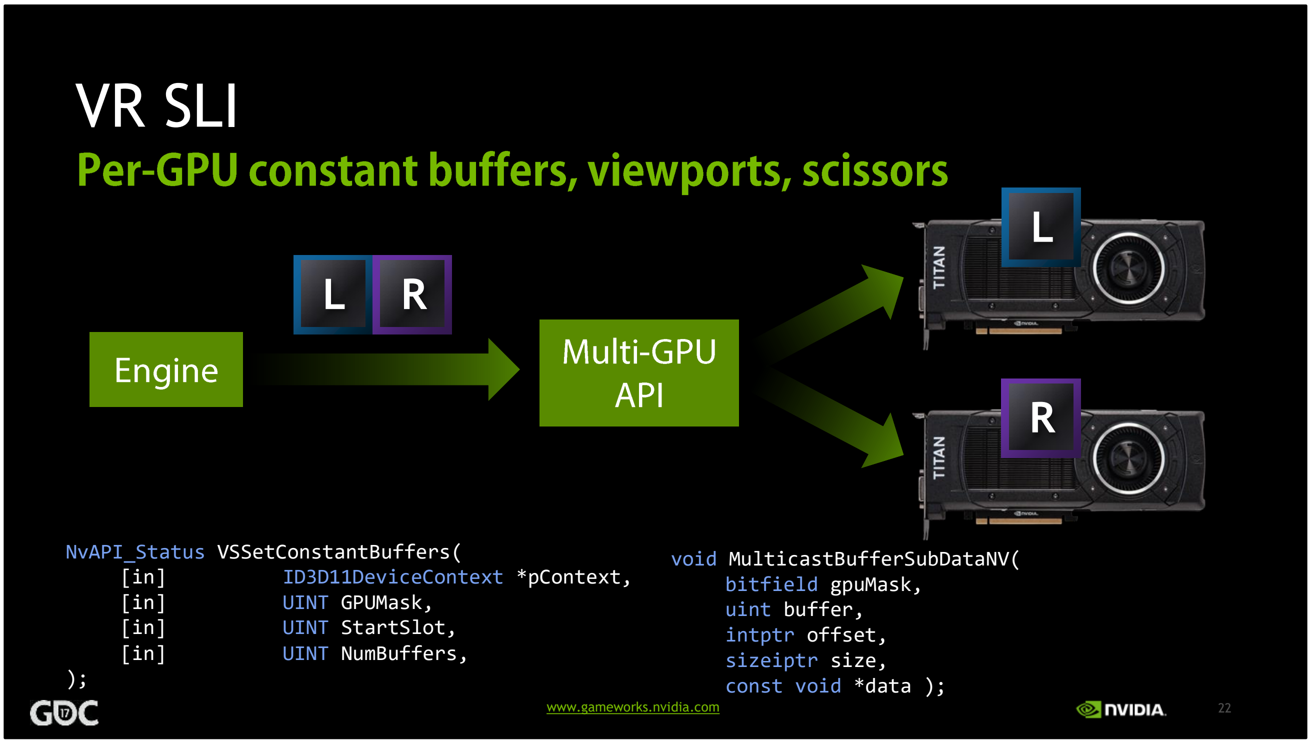
### Broadcasting reduces CPU overhead

GDC www.gameworks.nvidia.com NVIDIA 21

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

```
NvAPI_Status VSSetConstantBuffers(
     [in]          ID3D11DeviceContext *pContext,
     [in]          UINT GPUMask,
     [in]          UINT StartSlot,
     [in]          UINT NumBuffers,
);
```

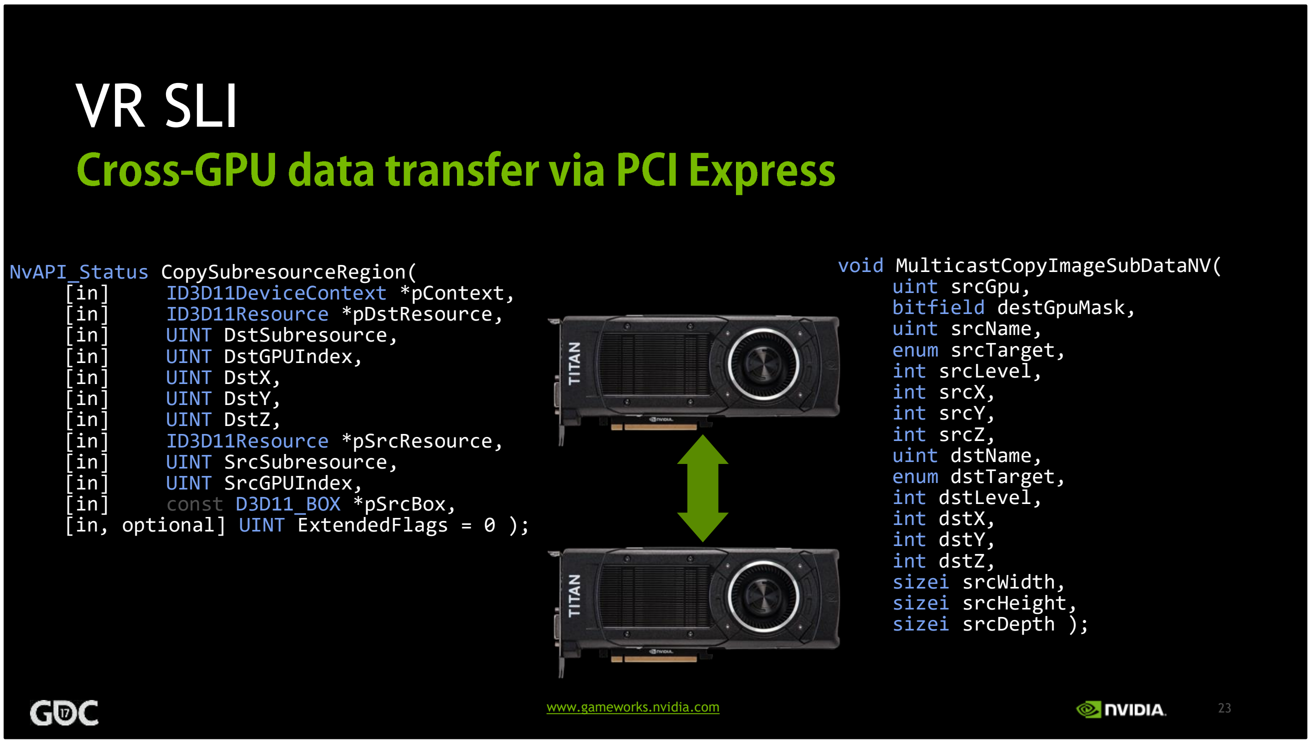
```
void MulticastBufferSubDataNV(
     bitfield gpuMask,
     uint buffer,
     intptr offset,
     sizeiptr size,
     const void *data );
```

www.gameworks.nvidia.com 22

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

```
NvAPI_Status CopySubresourceRegion(
     [in]     ID3D11DeviceContext *pContext,
     [in]     ID3D11Resource *pDstResource,
     [in]     UINT DstSubresource,
     [in]     UINT DstGPUIndex,
     [in]     UINT DstX,
     [in]     UINT DstY,
     [in]     UINT DstZ,
     [in]     ID3D11Resource *pSrcResource,
     [in]     UINT SrcSubresource,
     [in]     UINT SrcGPUIndex,
     [in]     const D3D11_BOX *pSrcBox,
     [in, optional] UINT ExtendedFlags = 0 );
```

```
void MulticastCopyImageSubDataNV(
     uint srcGpu,
     bitfield destGpuMask,
     uint srcName,
     enum srcTarget,
     int srcLevel,
     int srcX,
     int srcY,
     int srcZ,
     uint dstName,
     enum dstTarget,
     int dstLevel,
     int dstX,
     int dstY,
     int dstZ,
     sizei srcWidth,
     sizei srcHeight,
     sizei srcDepth );
```

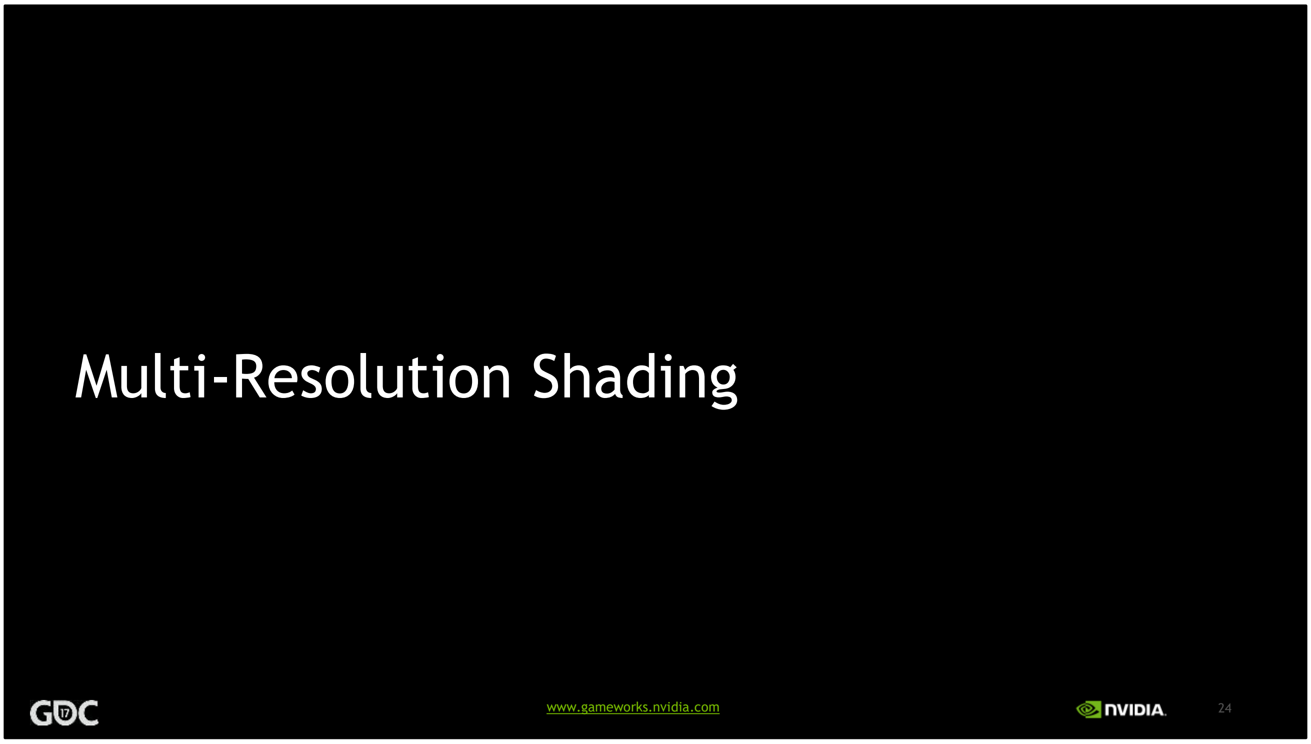
www.gameworks.nvidia.com

23

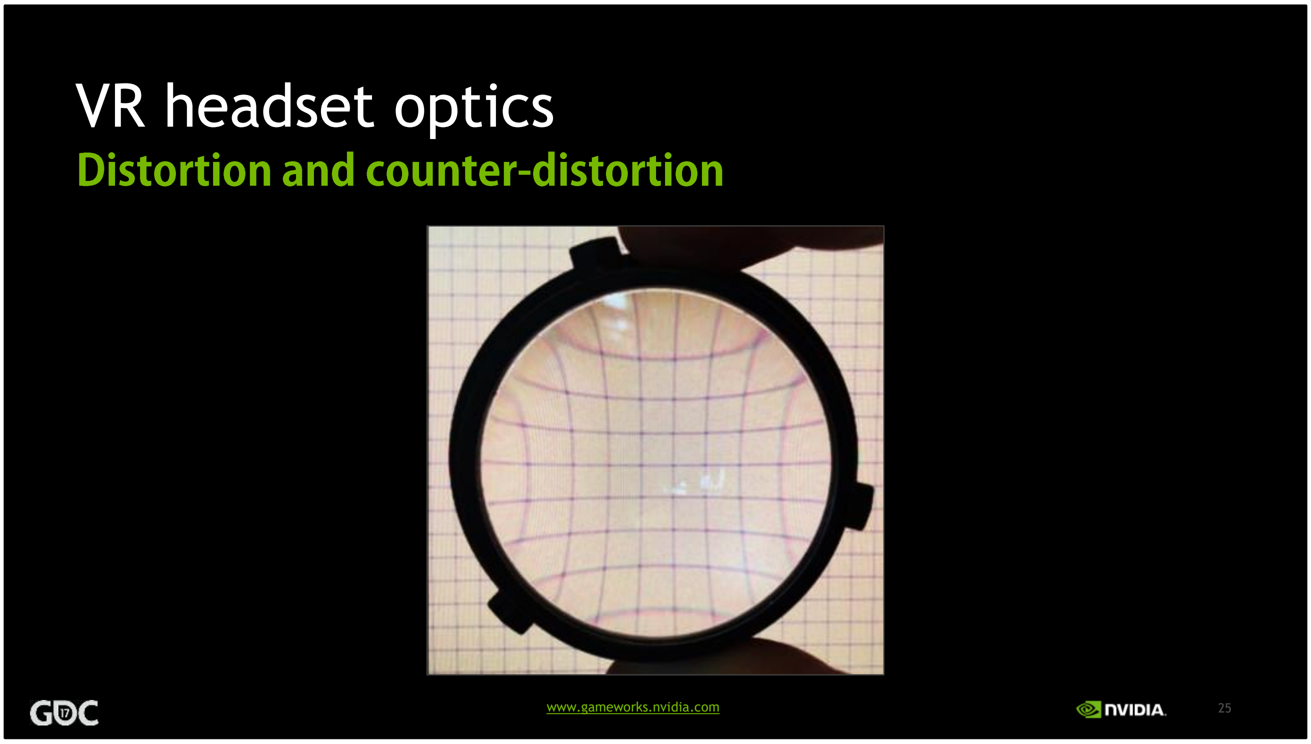
Of course, at times we need to be able to transfer data between GPUs. For instance, after we've finished rendering our two eye views, we have to get them back onto a single GPU to output to the display. So we have an API call that lets you copy a texture or a buffer between two specified GPUs, or to/from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

One point worth noting here is PCI Express bus bandwidth. PCIe2.0 x16 gives you 8 GB/sec of bandwidth, which isn't a huge amount, 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 you have the opportunity to hide the PCIe latency behind some other work.

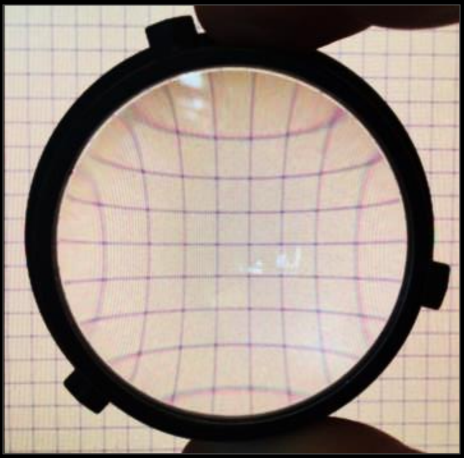


## Multi-Resolution Shading



## VR headset optics

### Distortion and counter-distortion

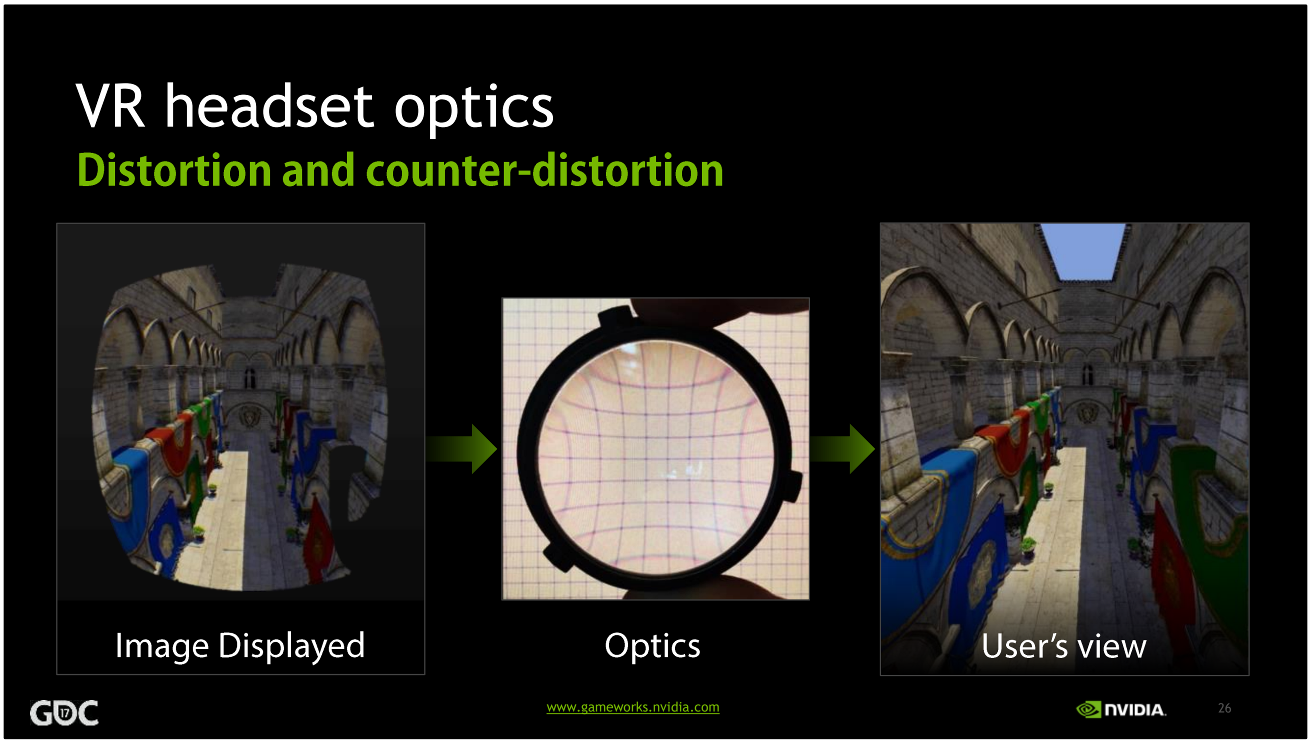


www.gameworks.nvidia.com

25

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.



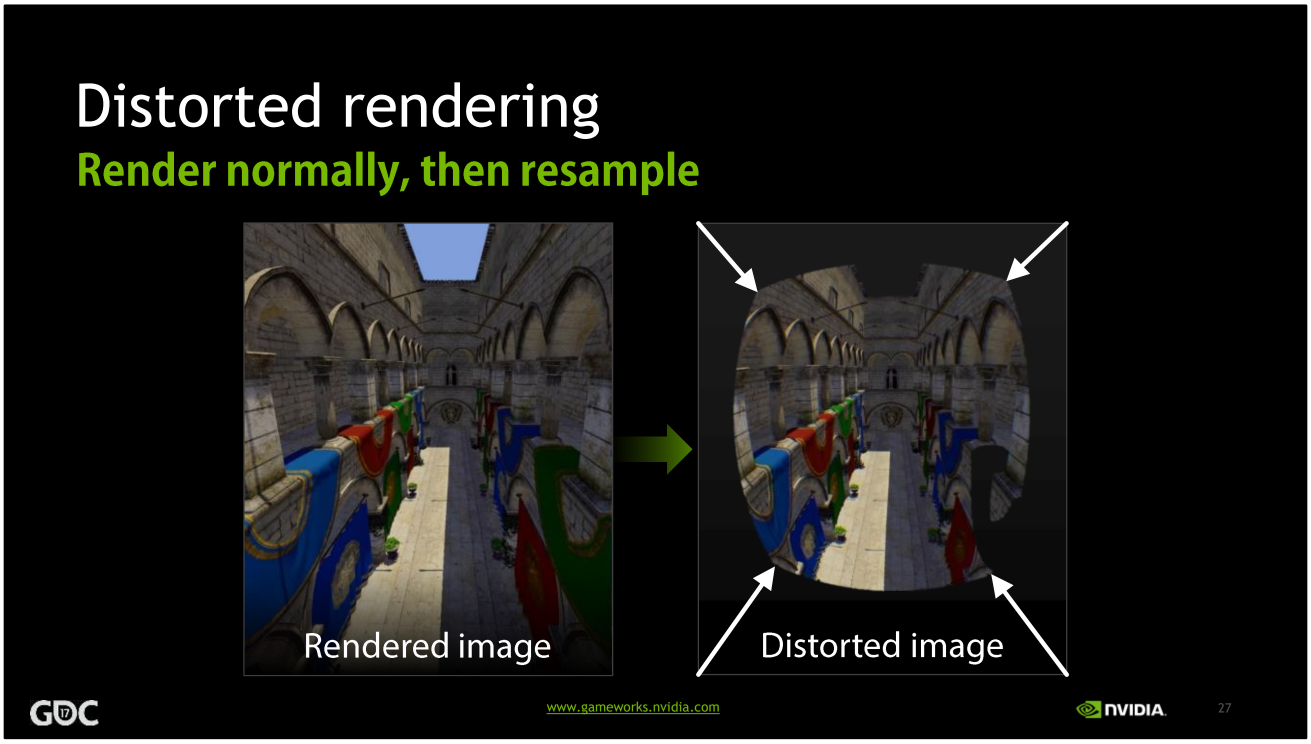
## VR headset optics

### Distortion and counter-distortion

Image Displayed
Optics
User's view
GDC
www.gameworks.nvidia.com
NVIDIA
26

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.



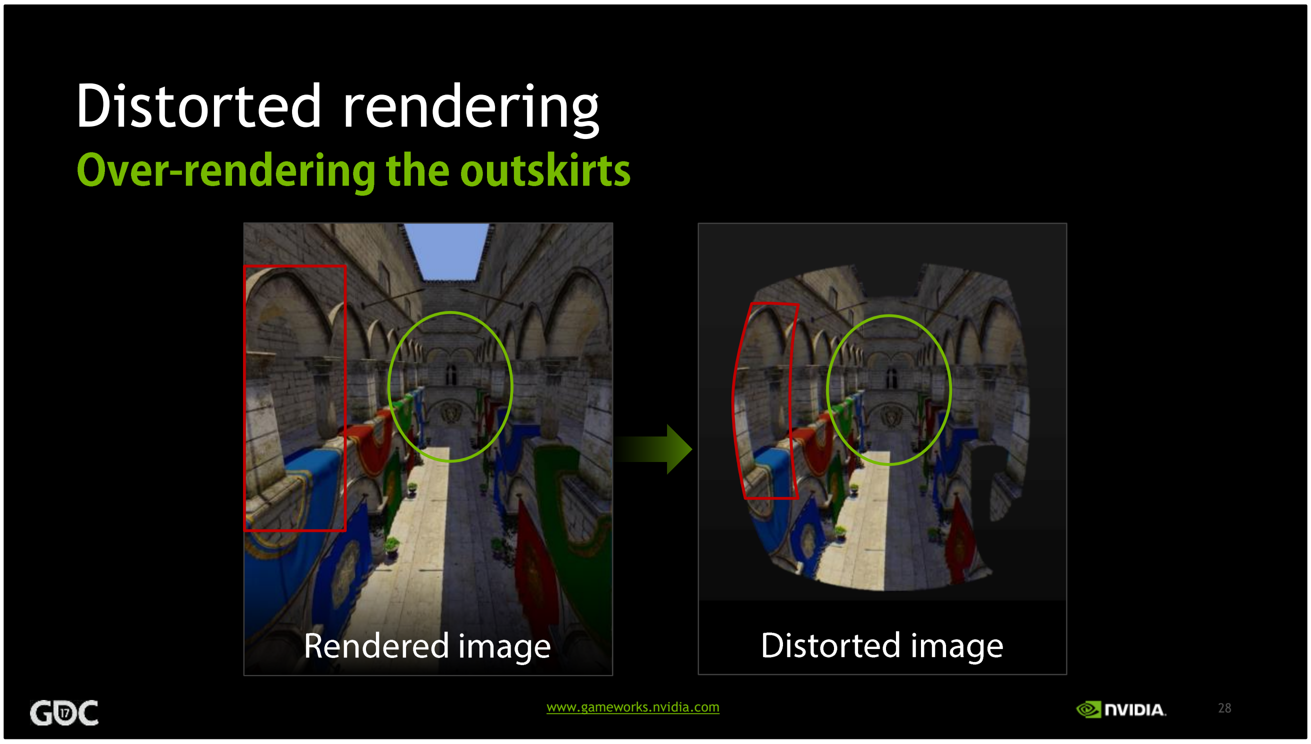
## Distorted rendering

**Render normally, then resample**

Rendered image
Distorted image
GDC
www.gameworks.nvidia.com
27

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.



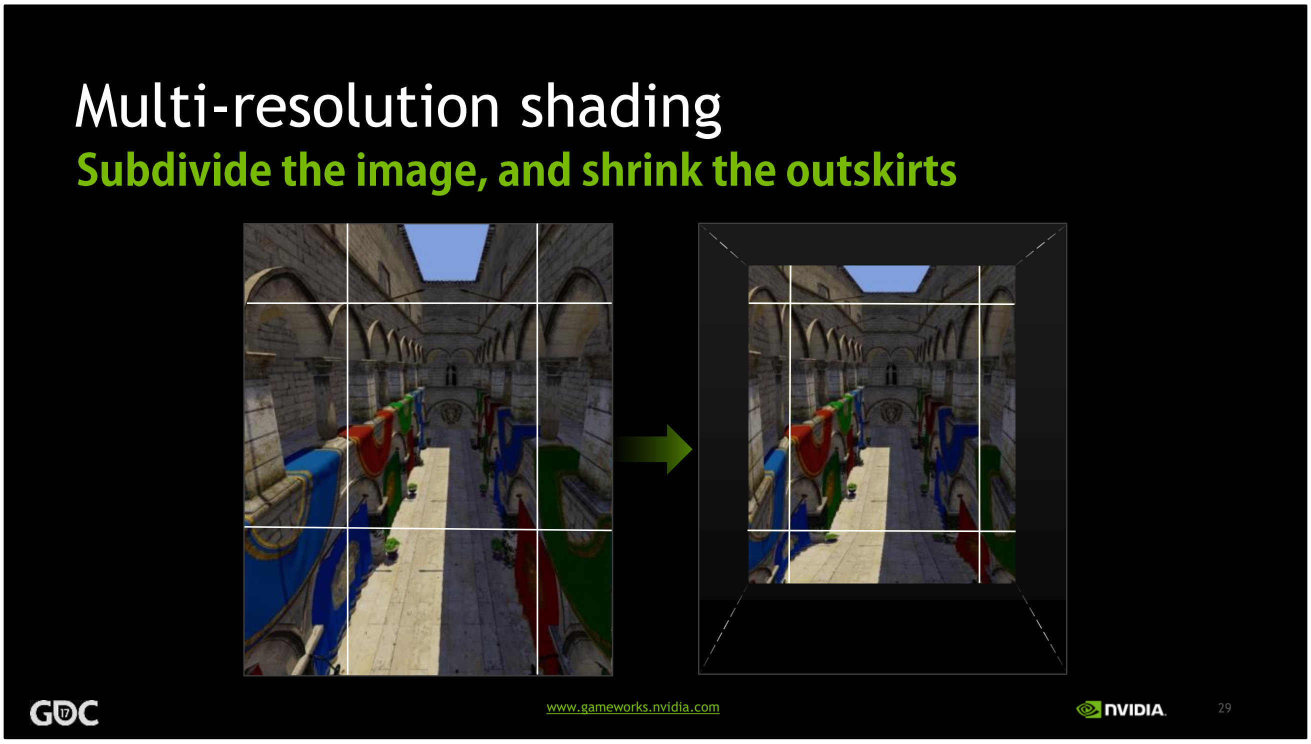
## Distorted rendering

### 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.

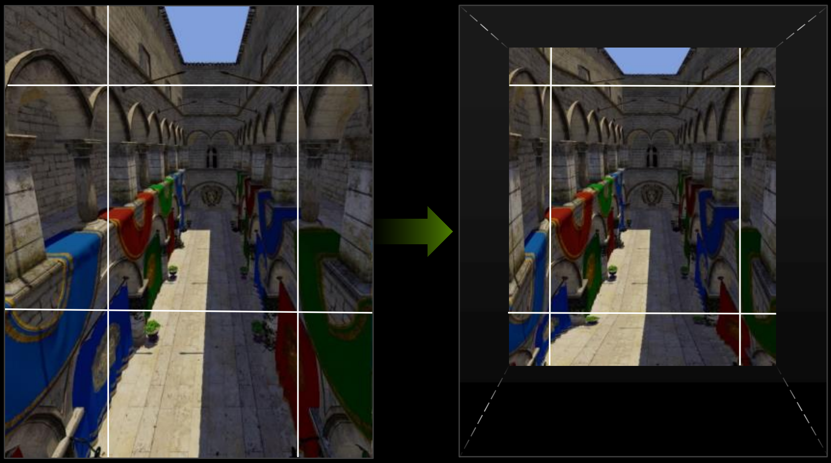
Look at the green circles—they're the same size, and they enclose the same region of the image in both the original and the distorted views. Then compare that to the red box. It gets mapped to a significantly smaller region in the distorted view.

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**



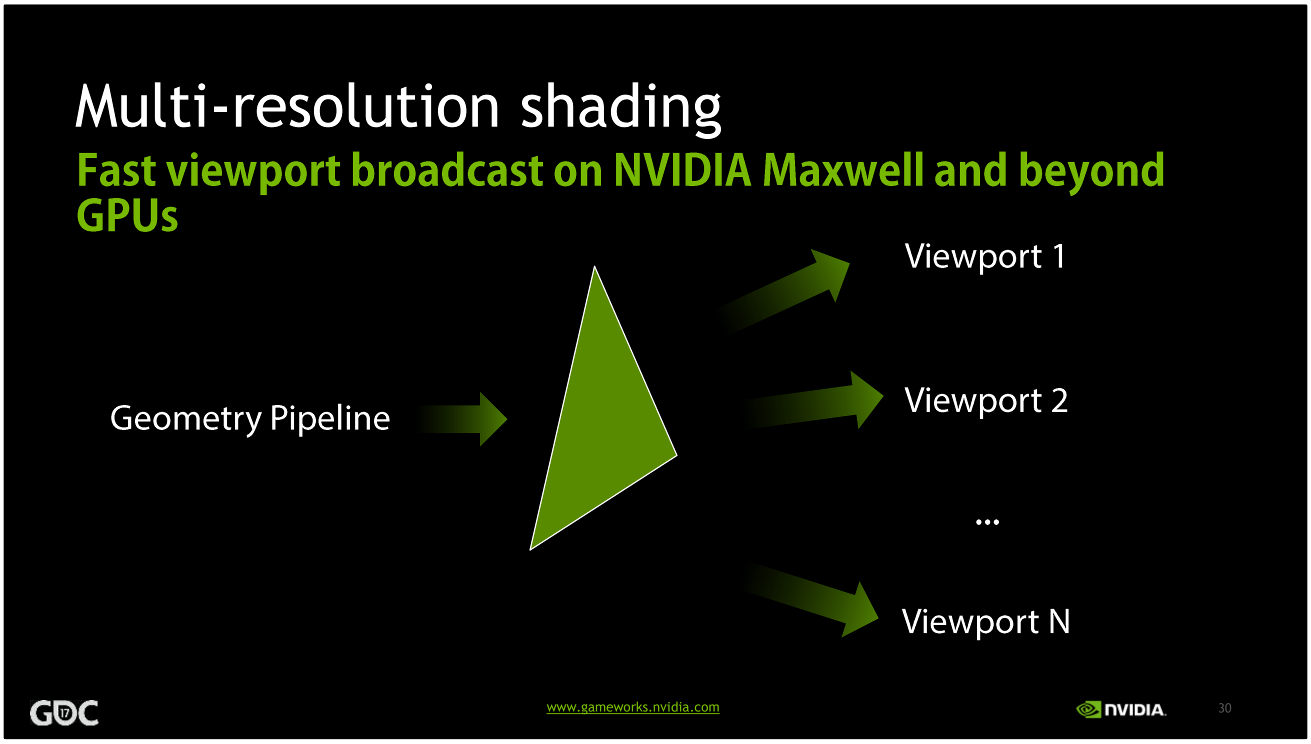
www.gameworks.nvidia.com

29

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

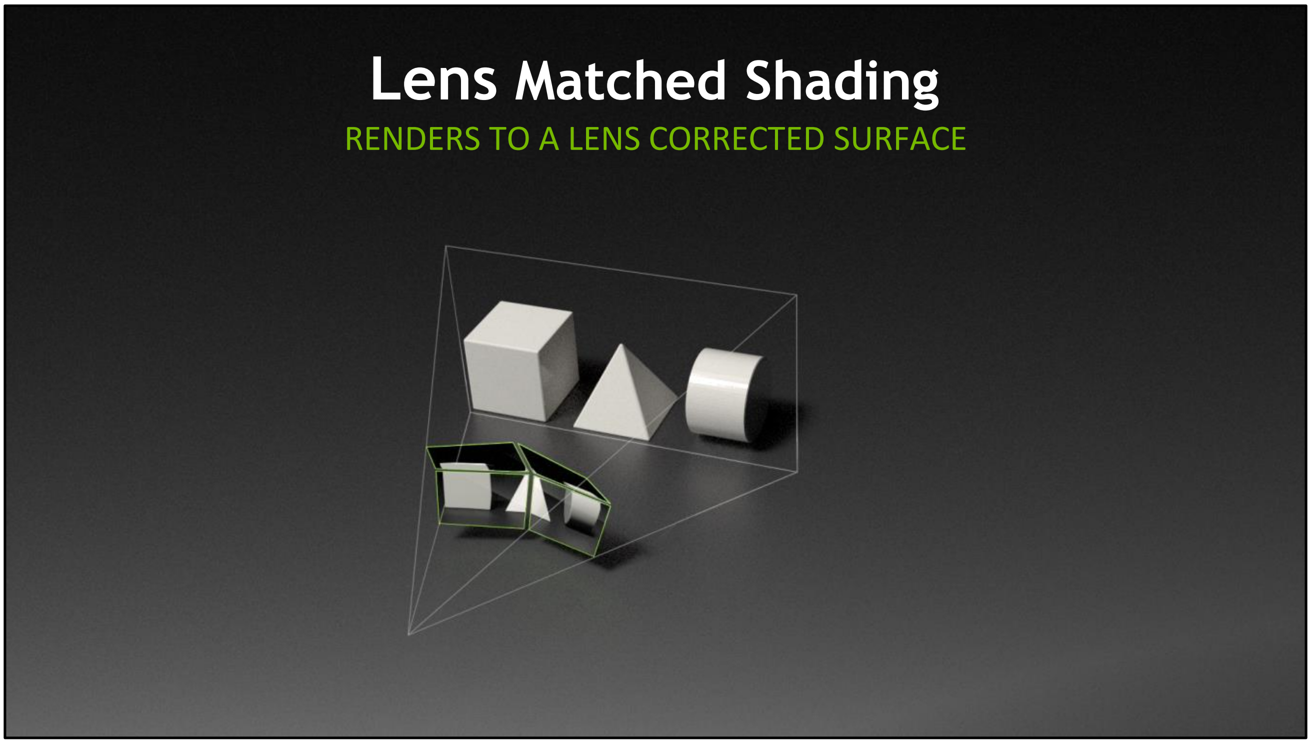
**Fast viewport broadcast on NVIDIA Maxwell and beyond GPUs**

www.gameworks.nvidia.com 30

The key thing that makes this technique a performance win is a hardware feature we have on NVIDIA's Maxwell and later architectures.

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

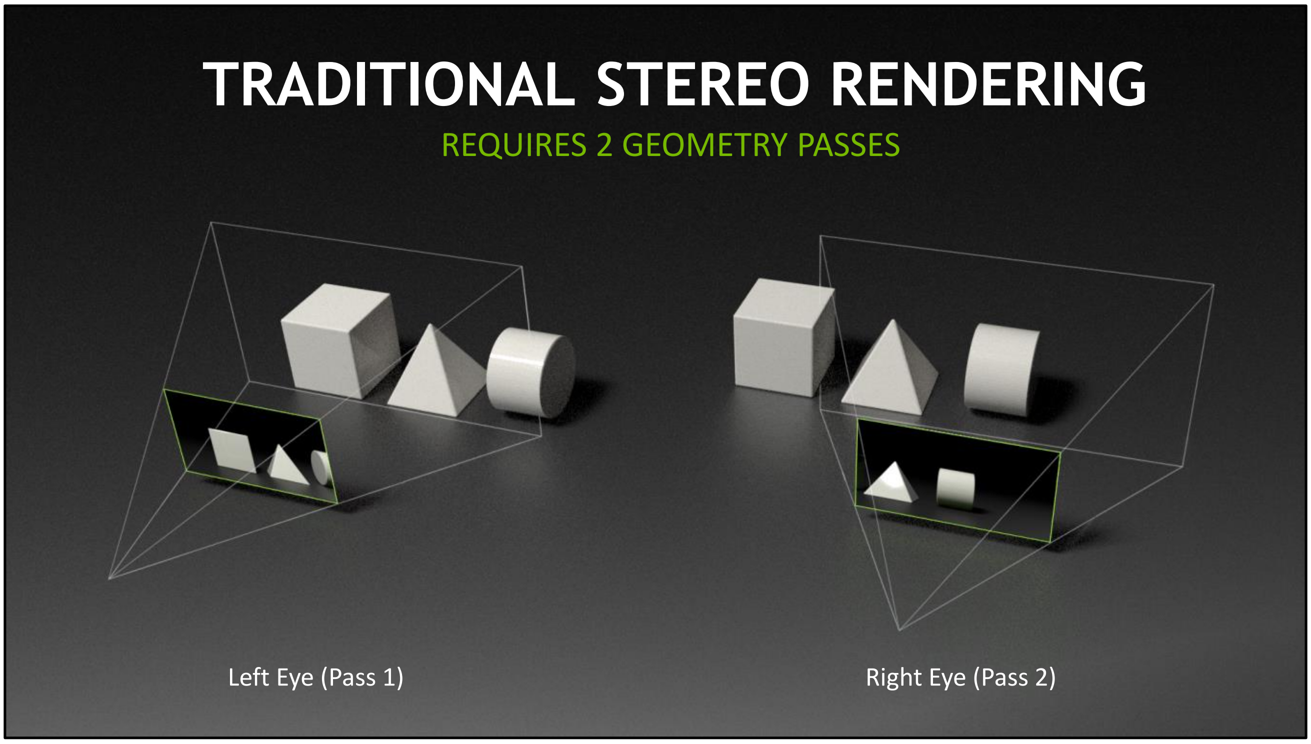
With Maxwell and beyond, we have the ability to very efficiently broadcast the geometry to many viewports, of arbitrary shapes and sizes, in hardware, while only submitting the draw calls once and running the GPU geometry pipeline once. That lets us render into this multi-resolution render target in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



## Lens Matched Shading

RENDERS TO A LENS CORRECTED SURFACE

Lens Matched Shading is a new feature of VRWorks that uses Pascal's Simultaneous Multi-projection capability to render directly to a surface that more closely approximates the final lens corrected display output. By doing this, GTX 1080 can improve performance by dramatically reducing the extra pixels that get drawn and then later discarded in the final lens warp.

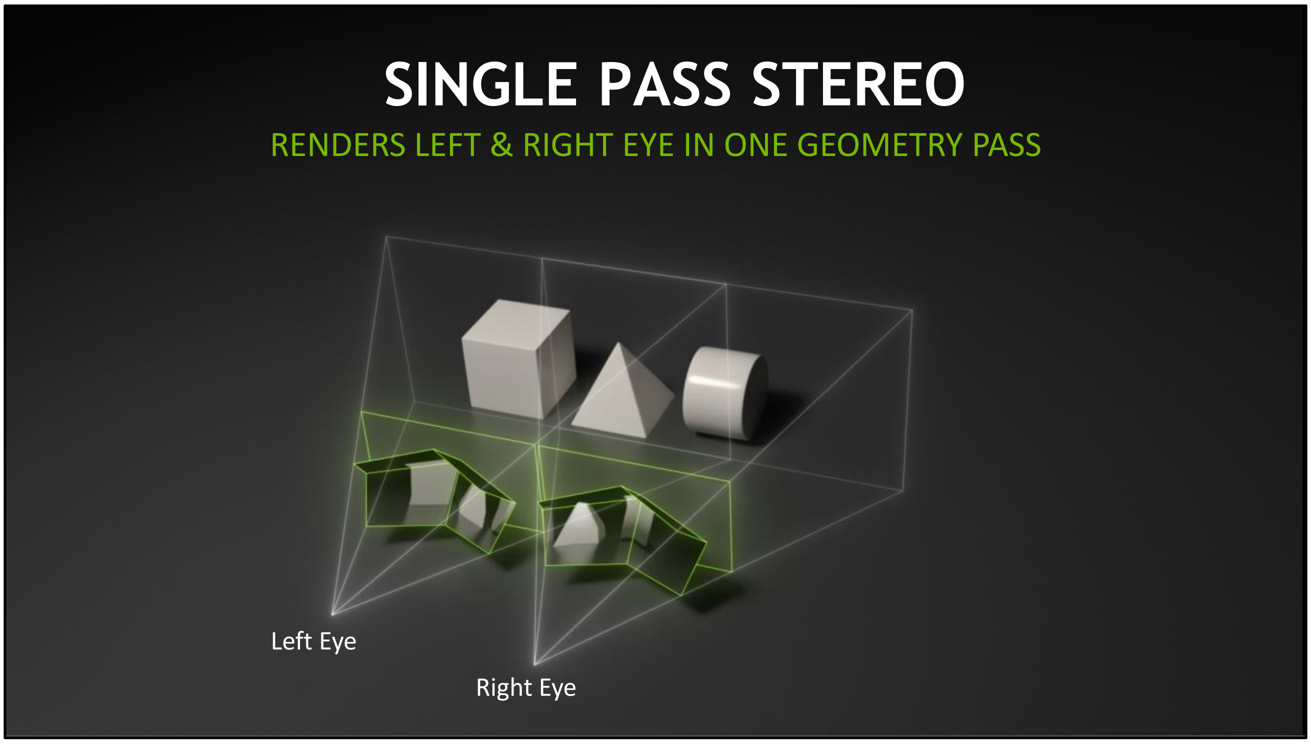


## TRADITIONAL STEREO RENDERING

REQUIRES 2 GEOMETRY PASSES

While Lens Matched Shading provides substantial performance improvements for pixel shading, let’s look at how Simultaneous Multi-projection can also save geometry performance.

As mentioned previously, virtual reality requires drawing an image for each eye. Traditional VR rendering does this in two separate geometry passes – one for the left eye, and one for the right eye.



## SINGLE PASS STEREO

RENDERS LEFT & RIGHT EYE IN ONE GEOMETRY PASS

Single Pass Stereo is a new feature of VRWorks that uses Pascal's Simultaneous Multi-projection capability to render left and right eyes in a single geometry pass. This saves an entire geometry pass, effectively doubling the amount of geometry the GPU can process for VR!

Lens Matched Shading improves pixel shading performance, and Single Pass Stereo improves geometry performance. When combined, these features deliver major improvements in VR rendering performance.

Note: some game engines use a feature called Instanced Stereo that saves CPU overhead by issuing a single draw call to the GPU for both left and right eyes. Unlike SPS, the GPU still has to render both eyes with Instanced Stereo. SPS offers both CPU and major GPU performance savings.

## VRWorks API availability

Direct3D 11, 12

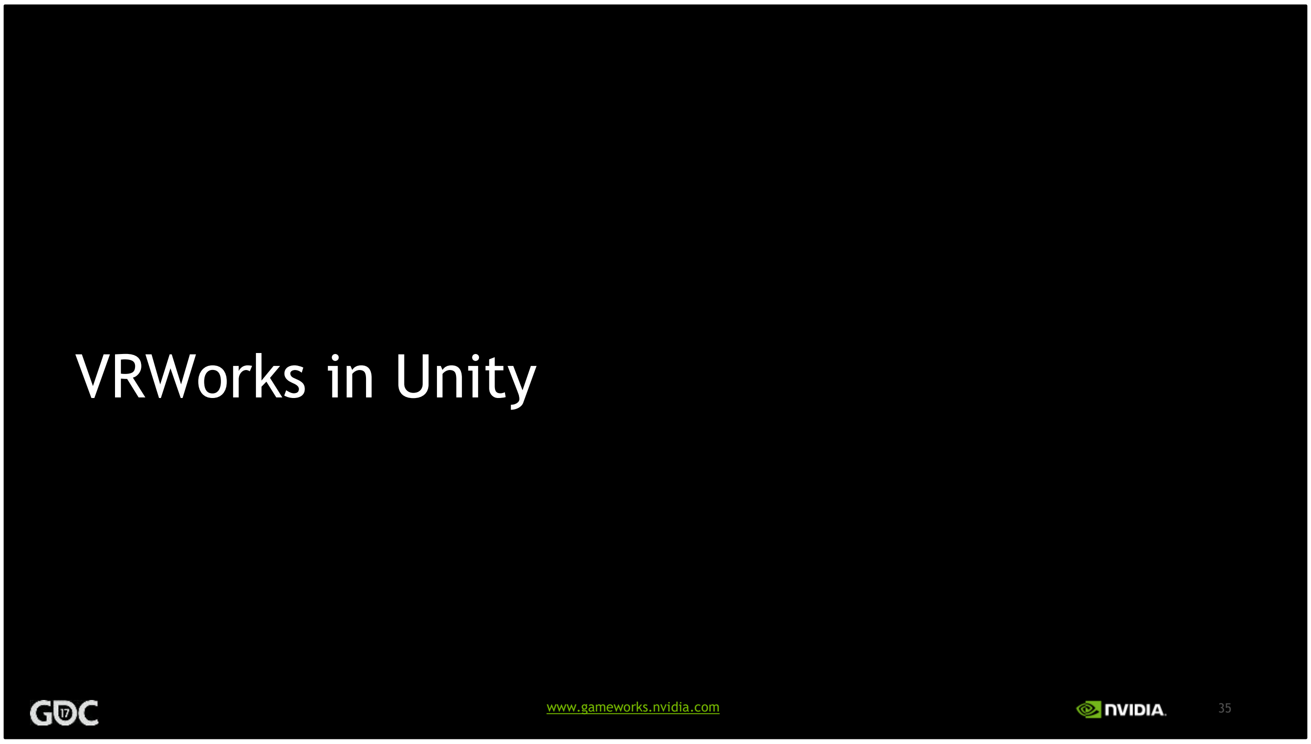
- Available now

OpenGL

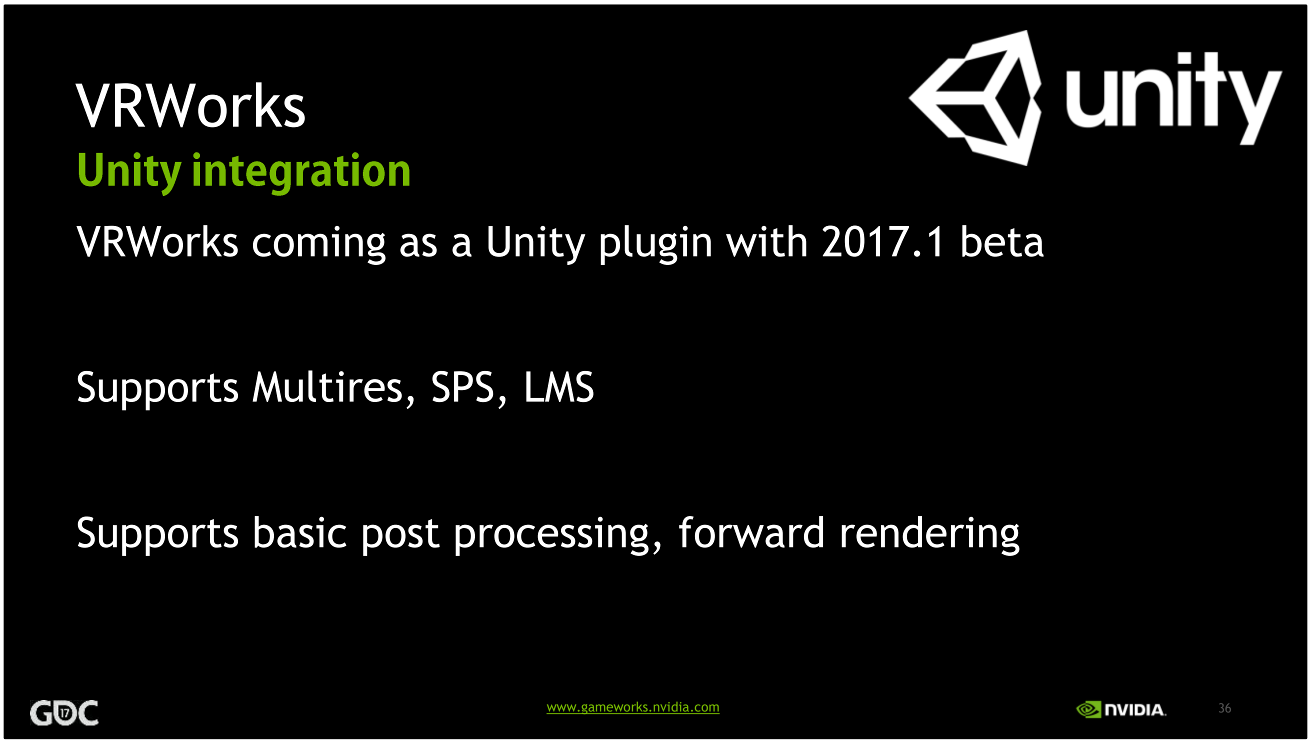
- Available now

Vulkan

- Available now



## VRWorks in Unity



## VRWorks

### Unity integration

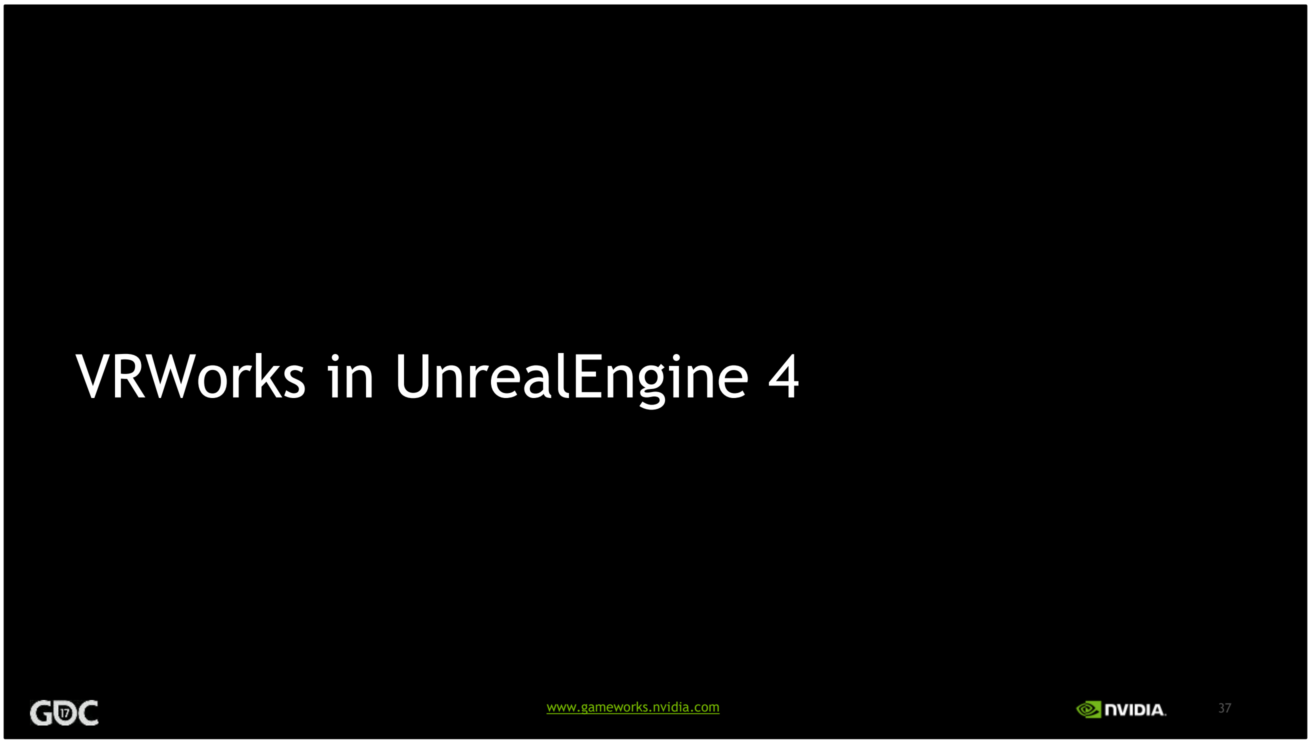
VRWorks coming as a Unity plugin with 2017.1 beta

Supports Multires, SPS, LMS

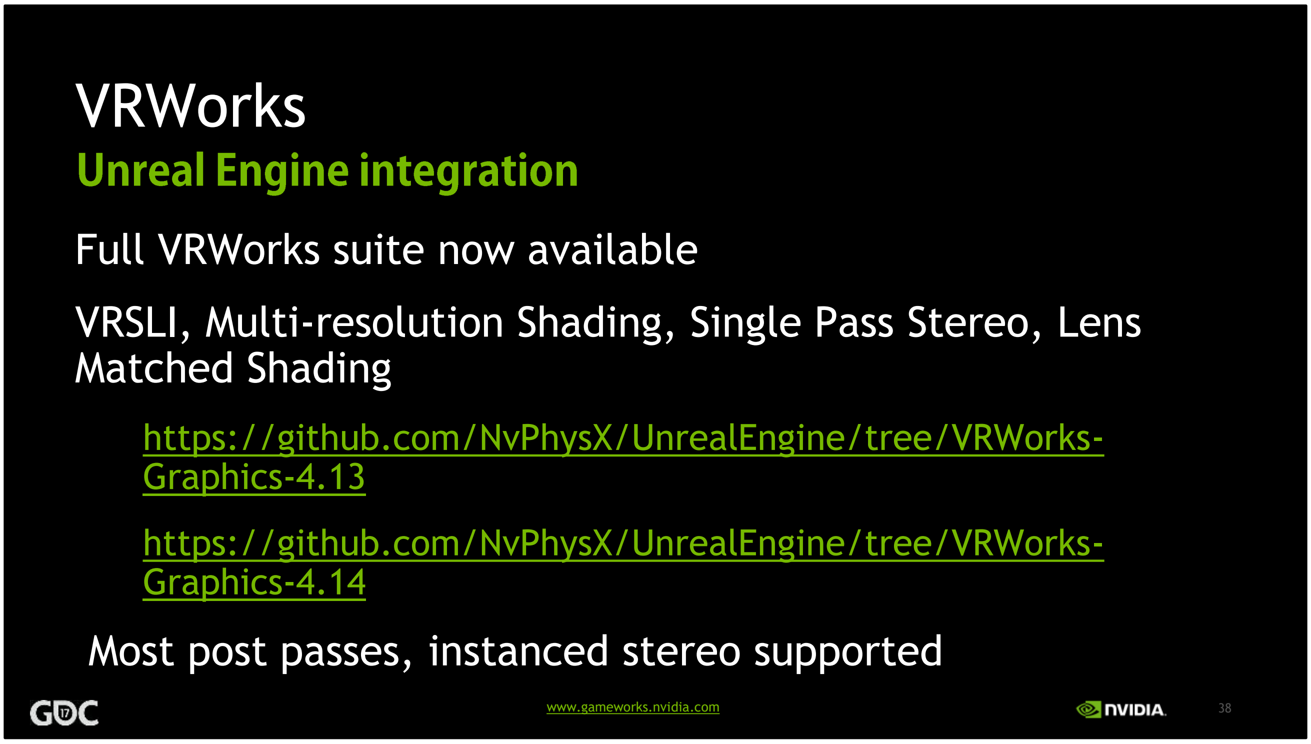
Supports basic post processing, forward rendering

www.gameworks.nvidia.com

36



## VRWorks in UnrealEngine 4



## VRWorks

### Unreal Engine integration

Full VRWorks suite now available

VRSLI, Multi-resolution Shading, Single Pass Stereo, Lens Matched Shading

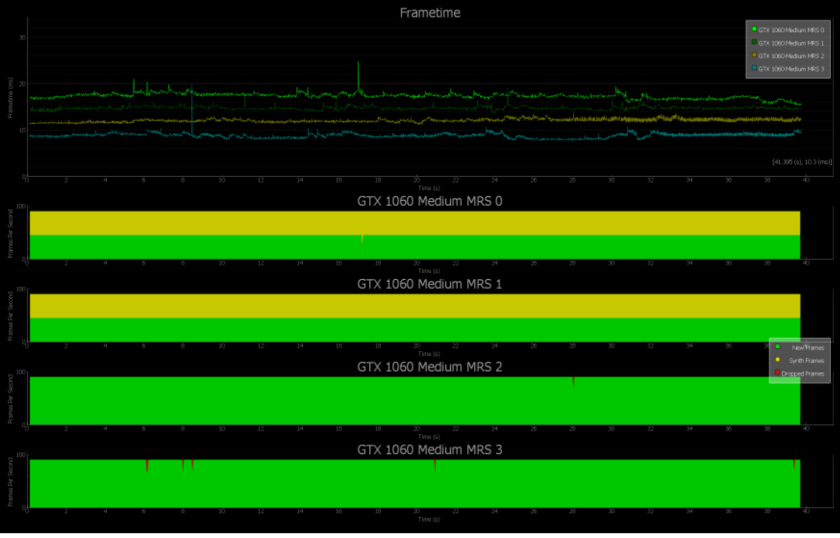
- https://github.com/NvPhysX/UnrealEngine/tree/VRWorks-Graphics-4.13
- https://github.com/NvPhysX/UnrealEngine/tree/VRWorks-Graphics-4.14

Most post passes, instanced stereo supported

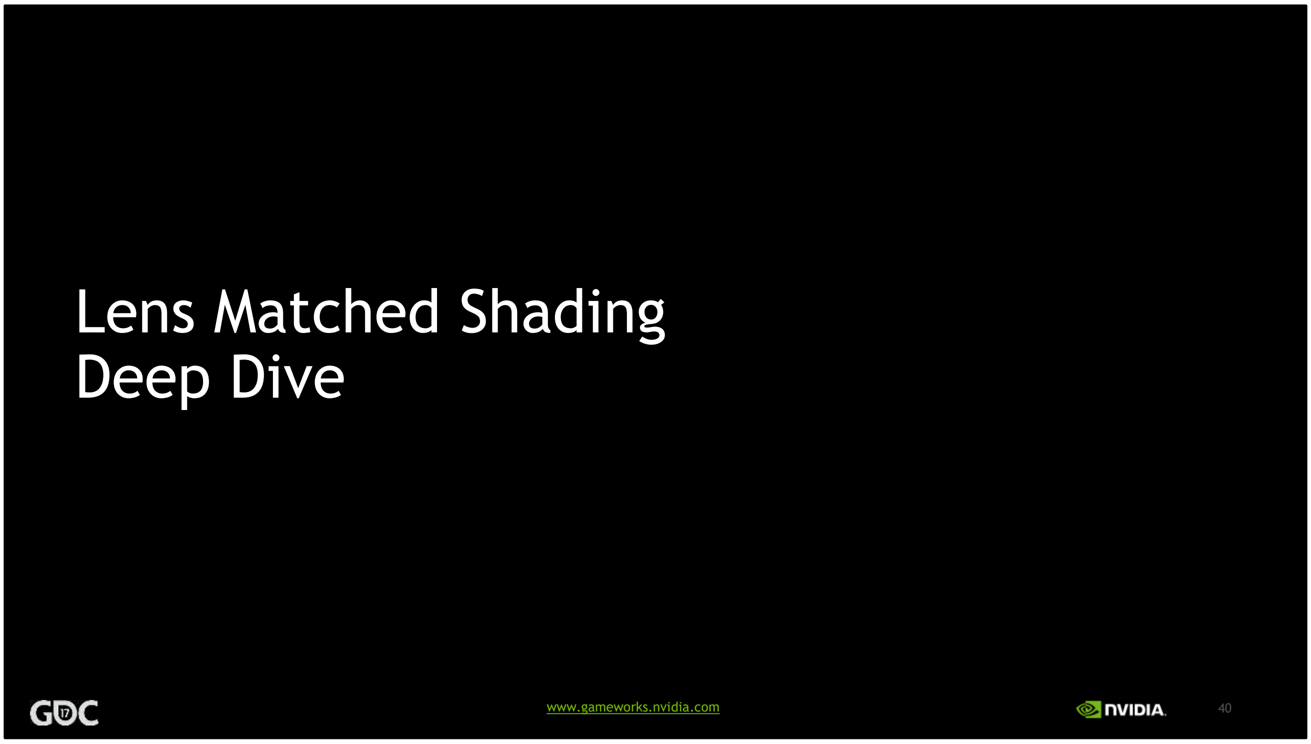
www.gameworks.nvidia.com 38

## Multi-Res Shading in UE4

### Performance



39



## Lens Matched Shading Deep Dive

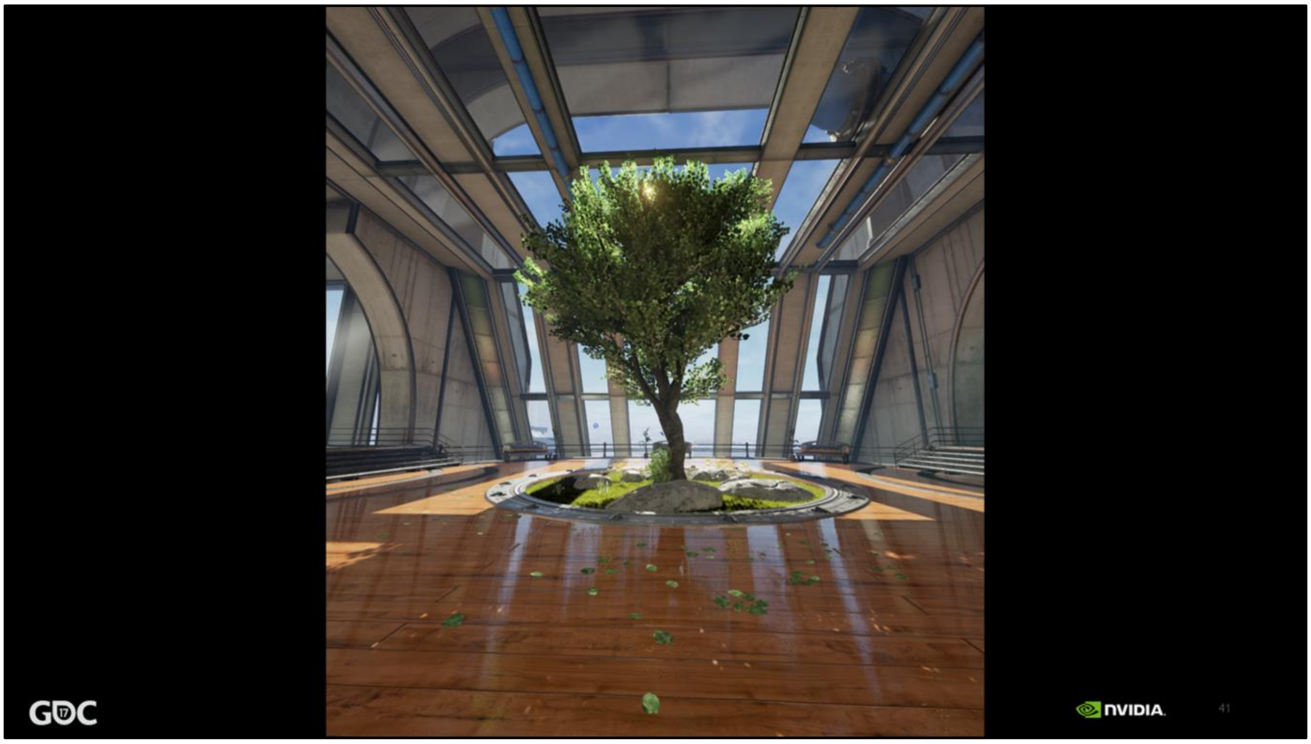
www.gameworks.nvidia.com

40

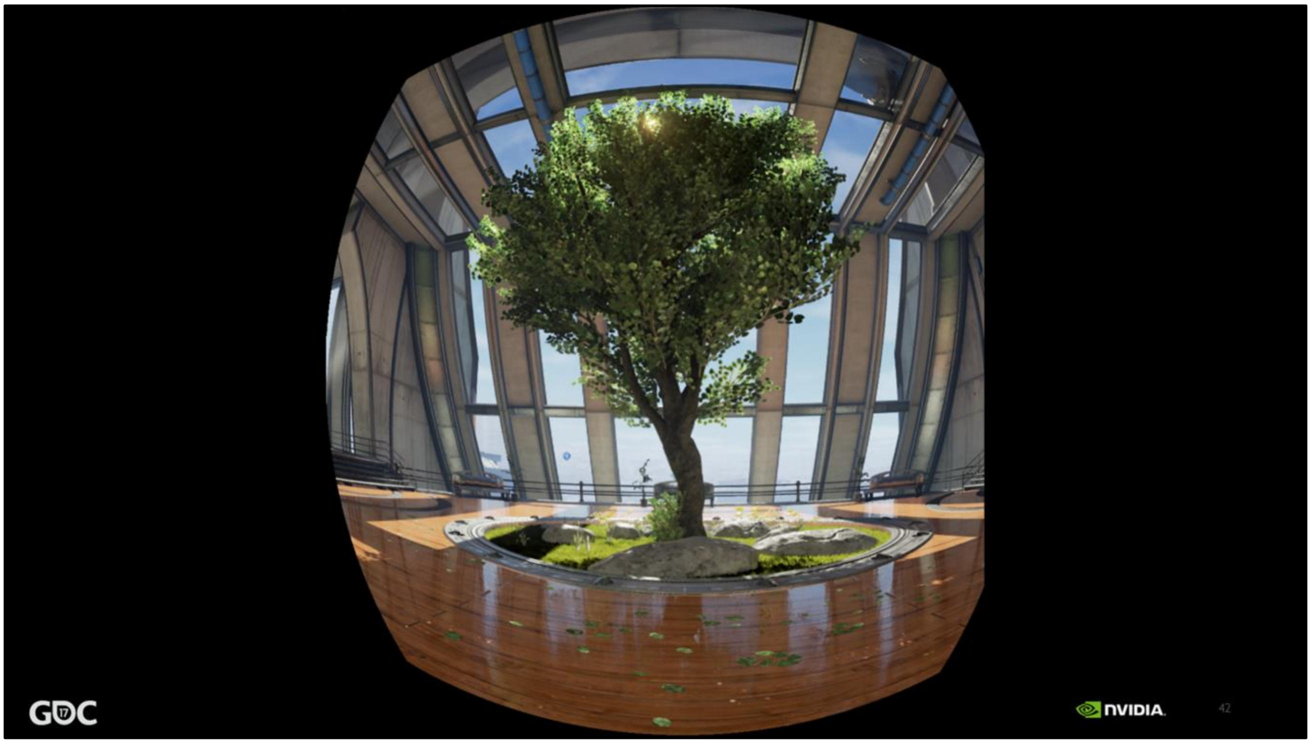
Thank you Cem, that was a great overall introduction to the VRWorks library.

I am Edward Liu, a DevTech at NVIDIA working on graphics and rendering on the geforce platform.

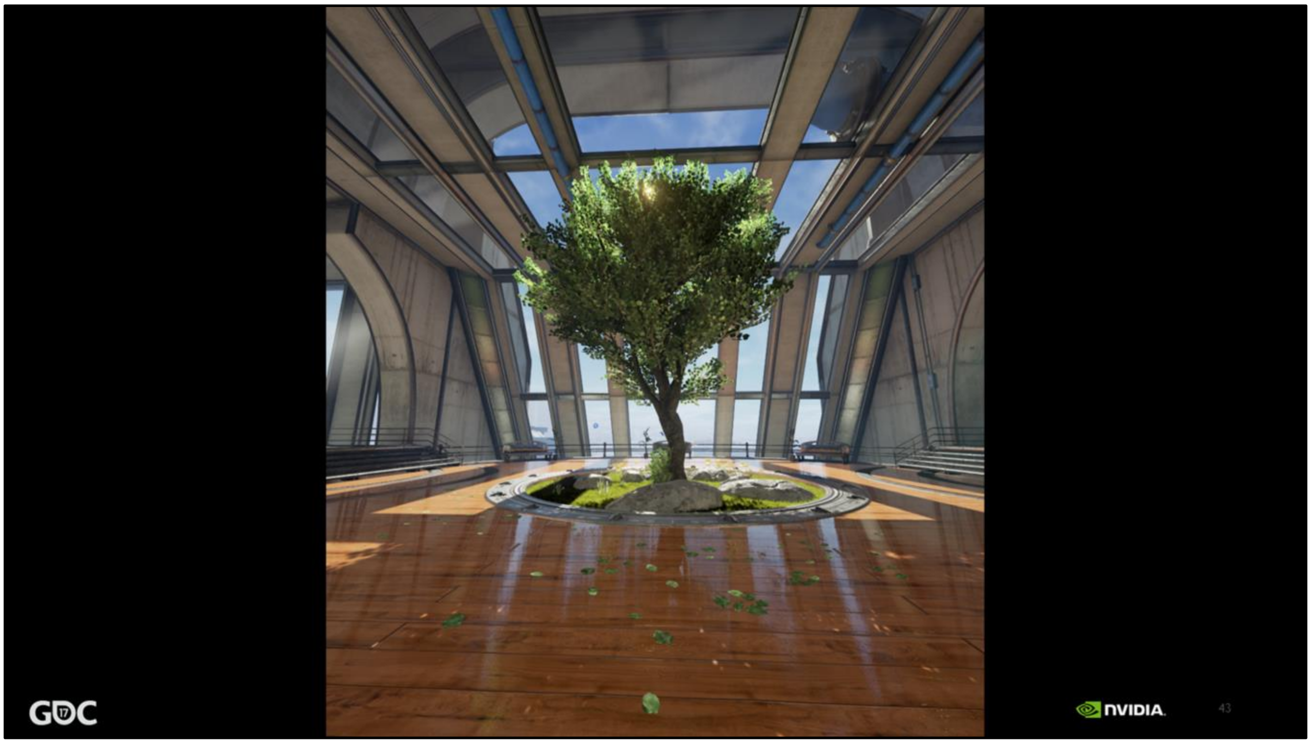
In my part of the talk, I will be providing an in depth introduction to Lens Matched Shading, and also dive deep into the gritty details of putting it into a modern game engine.



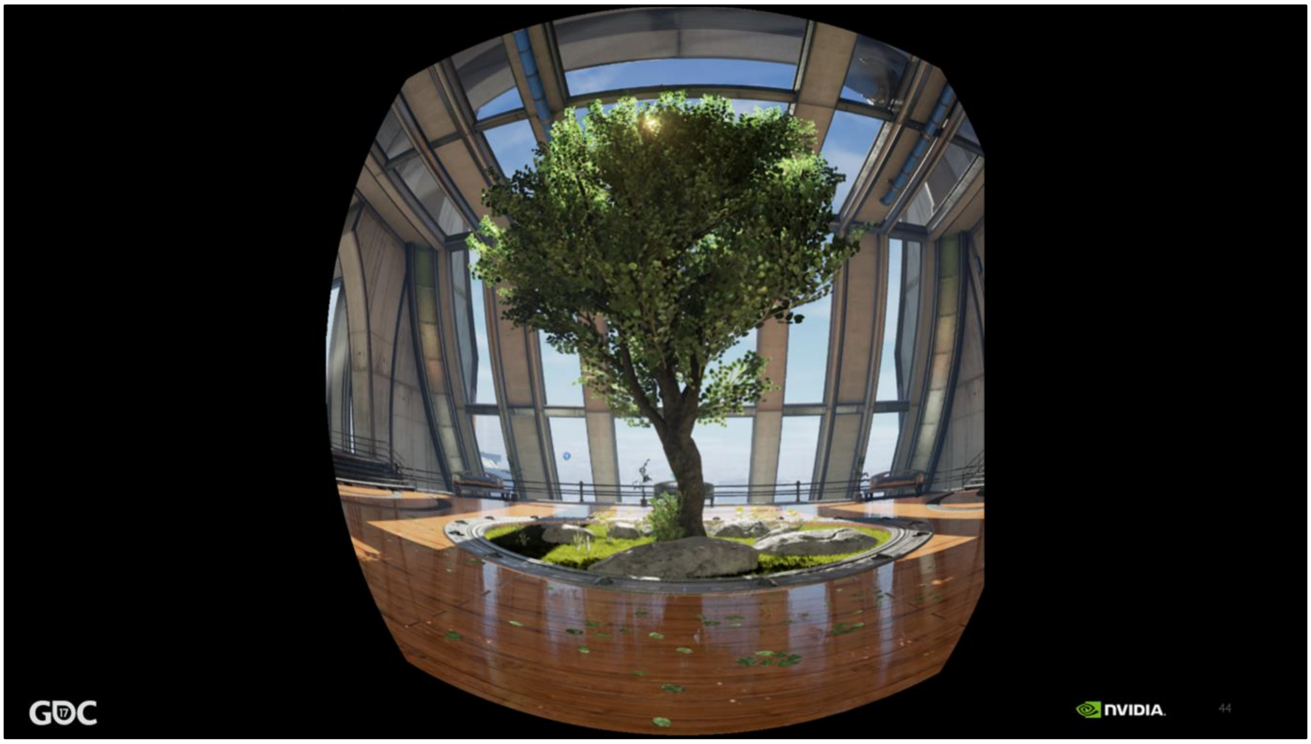
Ok. Let’s begin by look at lens warp. In every VR application, at the end of the rendering pipeline there’s lens warp. It distorts a regularly rendered image in a way that it can be viewed through HMD lenses. Here I have prepared two screenshots comparing the before and after of applying lens warp, which hopefully will provide you with a good intuition of what it does to an image. (click)



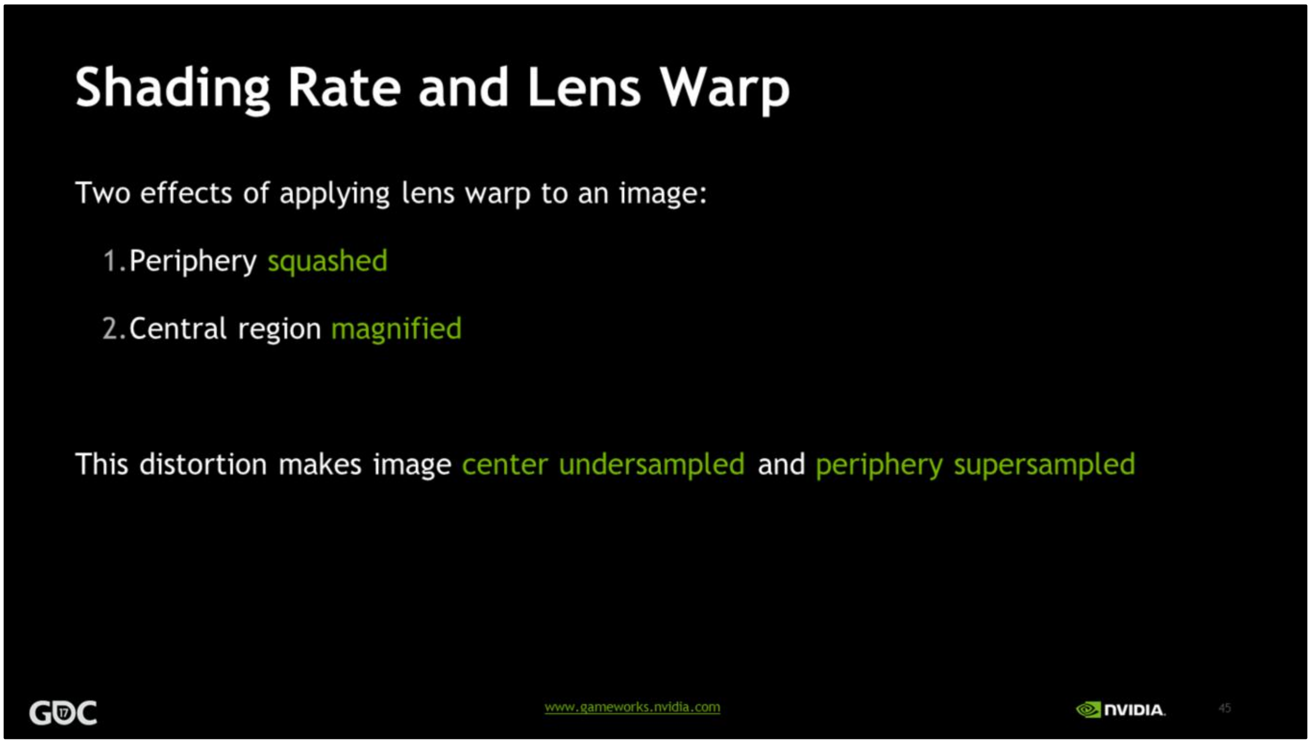
This is the result of applying lens warp to the image on the previous slide. Let me quickly flip back and forth here, so you won't miss any details.



Ok. Let’s begin by look at lens warp. In every VR application, at the end of the rendering pipeline there’s lens warp. It distorts a regularly rendered image in a way that it can be viewed through HMD lenses. Here I have prepared two screenshots comparing the before and after of applying lens warp, which hopefully will provide you with a good intuition of what it does to an image. (click)



This is the result of applying lens warp to the image on the previous slide. Let me quickly flip back and forth here, so you won't miss any details.



## Shading Rate and Lens Warp

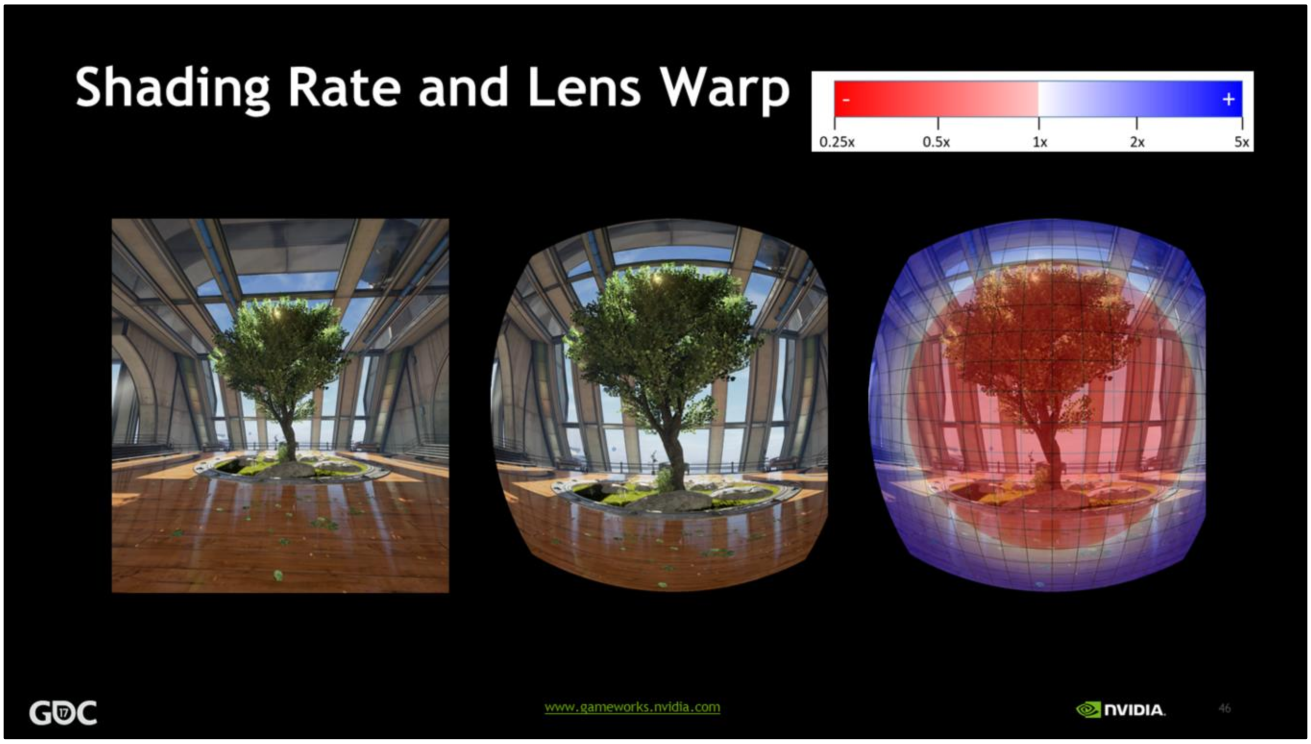
Two effects of applying lens warp to an image:

1. Periphery squashed
2. Central region magnified

This distortion makes image center undersampled and periphery supersampled

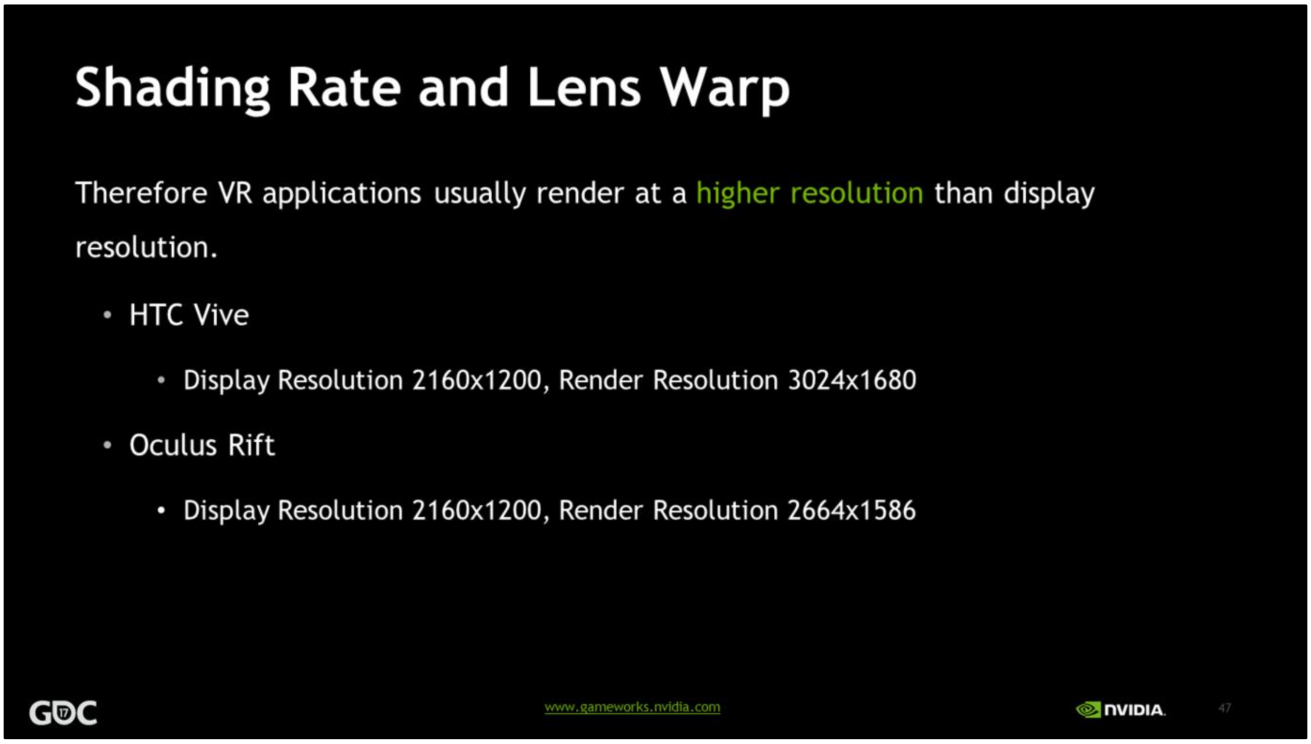
GDC
www.gameworks.nvidia.com
NVIDIA
45

You have probably noticed, lens warp works like a magnifying glass. It significantly enlarges the central part of the image, look at how the tree is enlarged for example. In the meantime, the periphery is squashed badly.



## Shading Rate and Lens Warp

So putting them side by side, applying the lens warp results in a redistributed shading rate as visualized on the right. Here red means the pixel is undersampled, which means the pixel per area retio after lens warp decreased. So as you can see most of the area on the lens warped image is actually severely undersampled.



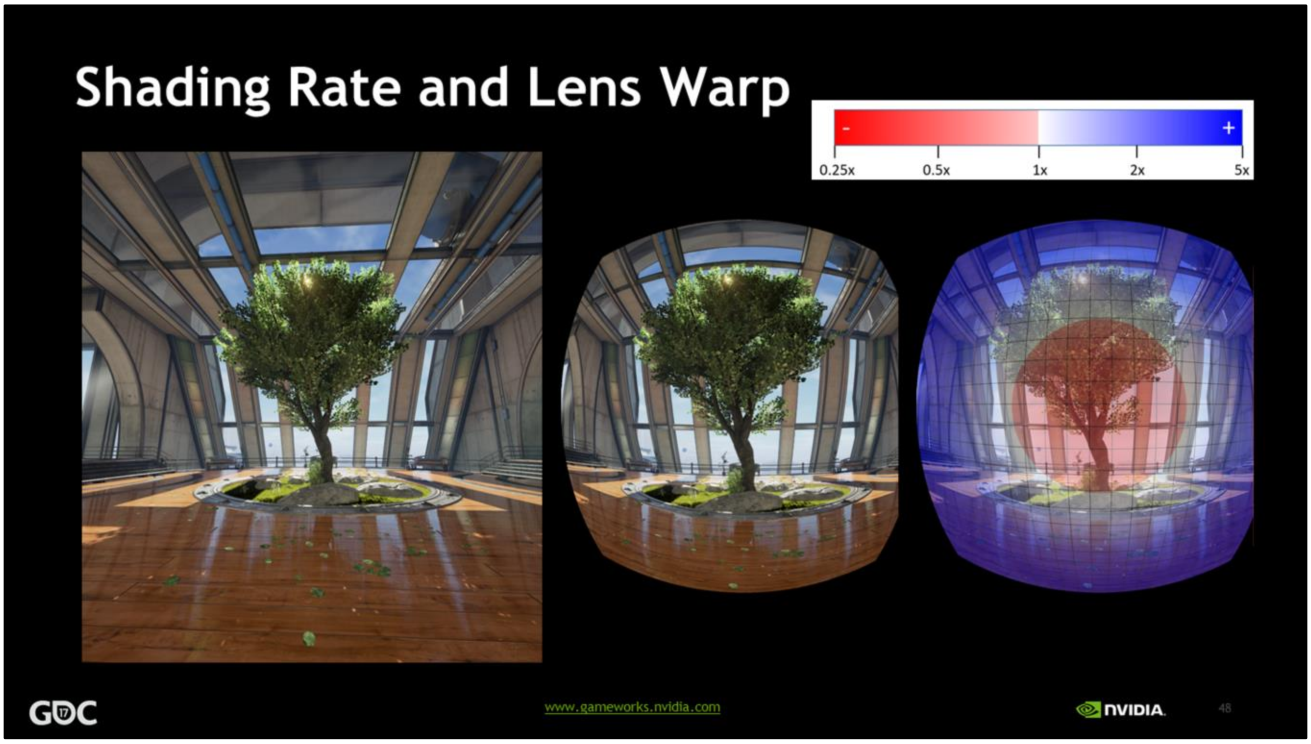
## Shading Rate and Lens Warp

Therefore VR applications usually render at a higher resolution than display resolution.

- HTC Vive
  - Display Resolution 2160x1200, Render Resolution 3024x1680
- Oculus Rift
  - Display Resolution 2160x1200, Render Resolution 2664x1586

GDC www.gameworks.nvidia.com NVIDIA 47

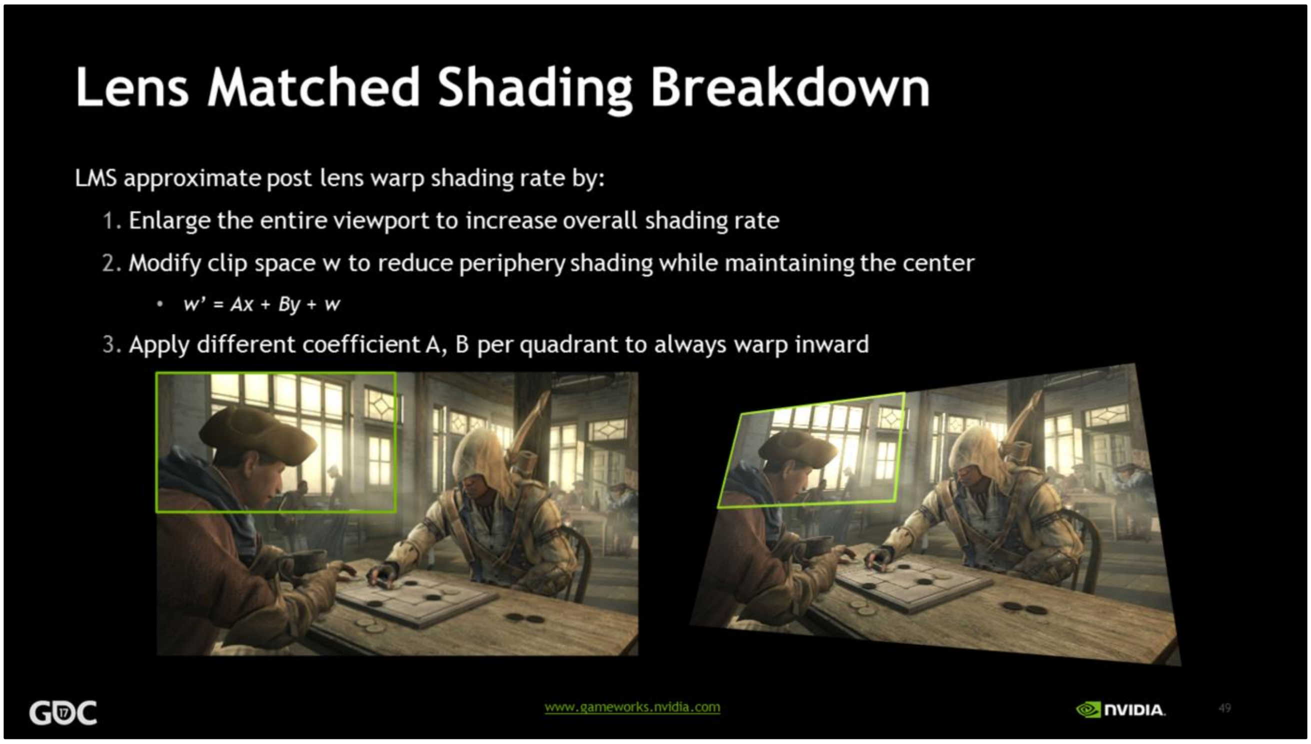
For this reason, all main stream VR vendors advise rendering at a resolution than the final displayed resolution. HTC Vive suggests rendering at 3024x1680, and Oculus Rift suggests rendering at 2664x1586, while the display resolution on both devices are only 2160x1200.



## Shading Rate and Lens Warp

If we render to a supersampled target and put things side by side again, the undersampling problem on the lens warped image is largely mitigated. However, the peripheral region is now highly supersampled, as much as 5x. In other words, lens warp is effectively discarding 80% of the shading work done at a significant portion of the  image, but the central part of the image, which is what the viewers will be focusing on for most of the time, is actually still slightly undersampled.

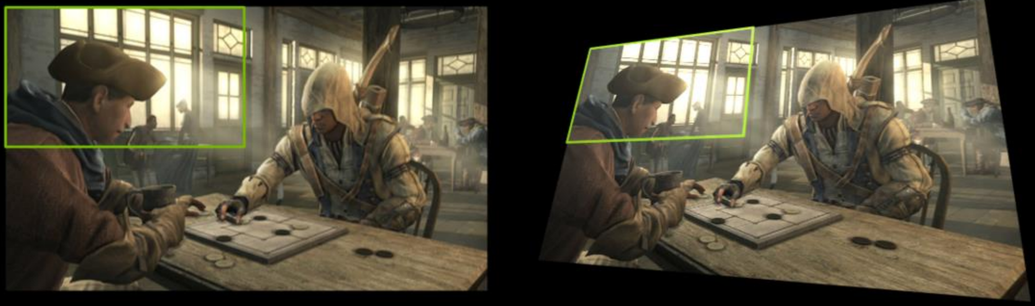
This is of course undesirable.



## Lens Matched Shading Breakdown

LMS approximate post lens warp shading rate by:

1. Enlarge the entire viewport to increase overall shading rate
2. Modify clip space w to reduce periphery shading while maintaining the center
   - *w' = Ax + By + w*
3. Apply different coefficient A, B per quadrant to always warp inward



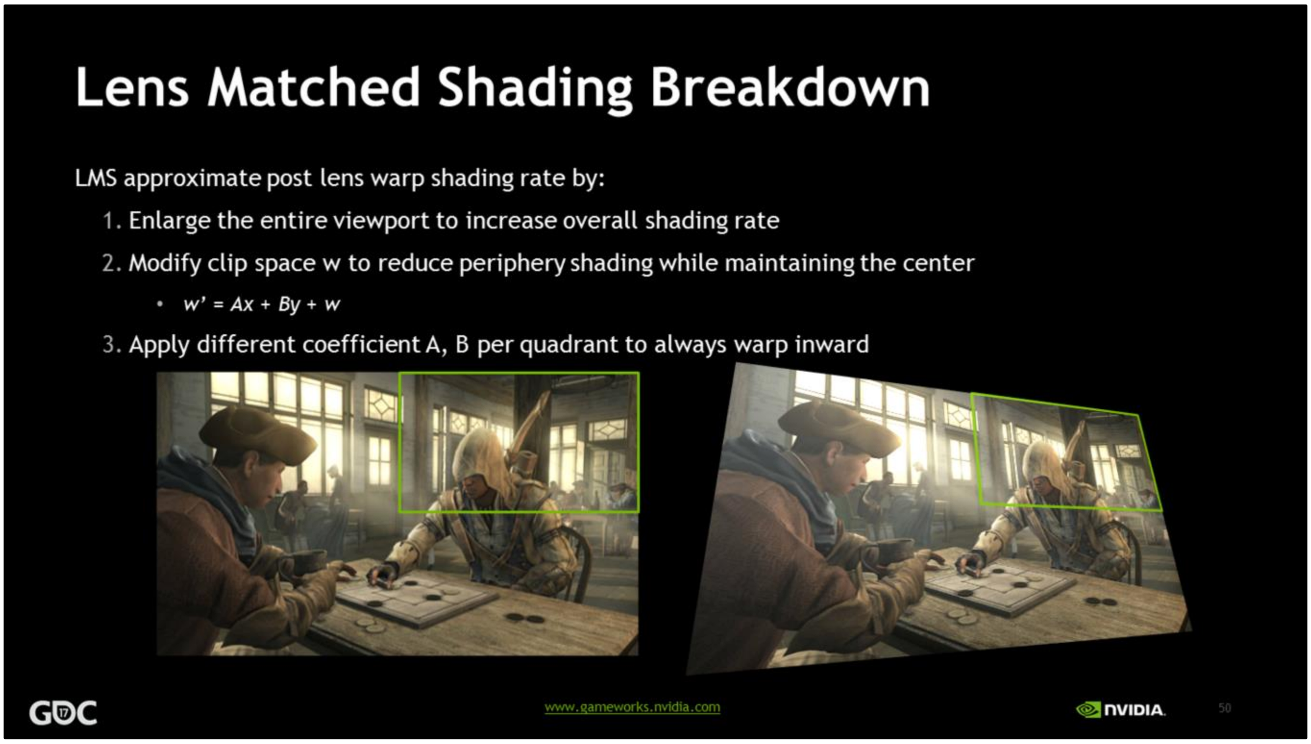
GDC www.gameworks.nvidia.com NVIDIA 49

Lens Matched Shading, or LMS, is designed to address this exact problem. It can be roughly broken down into two main pieces which address the two effects of lens warp accordingly.
First , it increases the overall shading rate evenly by simply using a larger viewport and render target, just like advised by all HMD vendors.
Second, it reduces the periphery shading rate by applying a technique we call modified w, which is a hardware feature in Pascal.
Modified w works by increasing the w component in homogeneous clip space with the equation w' = Ax+By+w. Here the x and y are clip space coordinates, and A and B are warp coefficients.

The image shown on the right is the result of applying modifled w with a single set of warp coefficient across the whole image. Notice that only the top left quadrant is warped inward and shading less pixels.

This is because clip space x and y spans from -1 to 1 domain, LMS therefore requires setting warp coefficients with different signs at each four clip space quadrant to increase the w component everywhere and to always warp the image inward. We do this by breaking a single view into four viewports (clickx3), each assigned different warp coefficients. And LMS uses the multiprojection feature to very efficiently broadcast geometries into those four viewports.

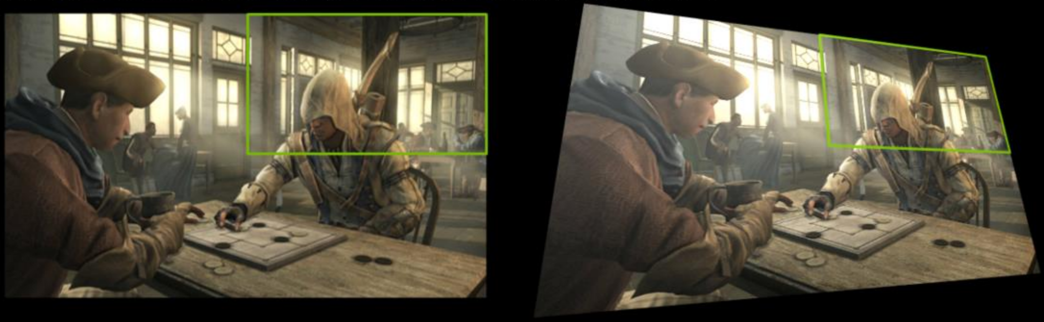
Notice that this will not change the shading rate at all at the very center pixel since x and y are zero, and the shading rate is gradually reduced at a 1 / x rate.



## Lens Matched Shading Breakdown

LMS approximate post lens warp shading rate by:

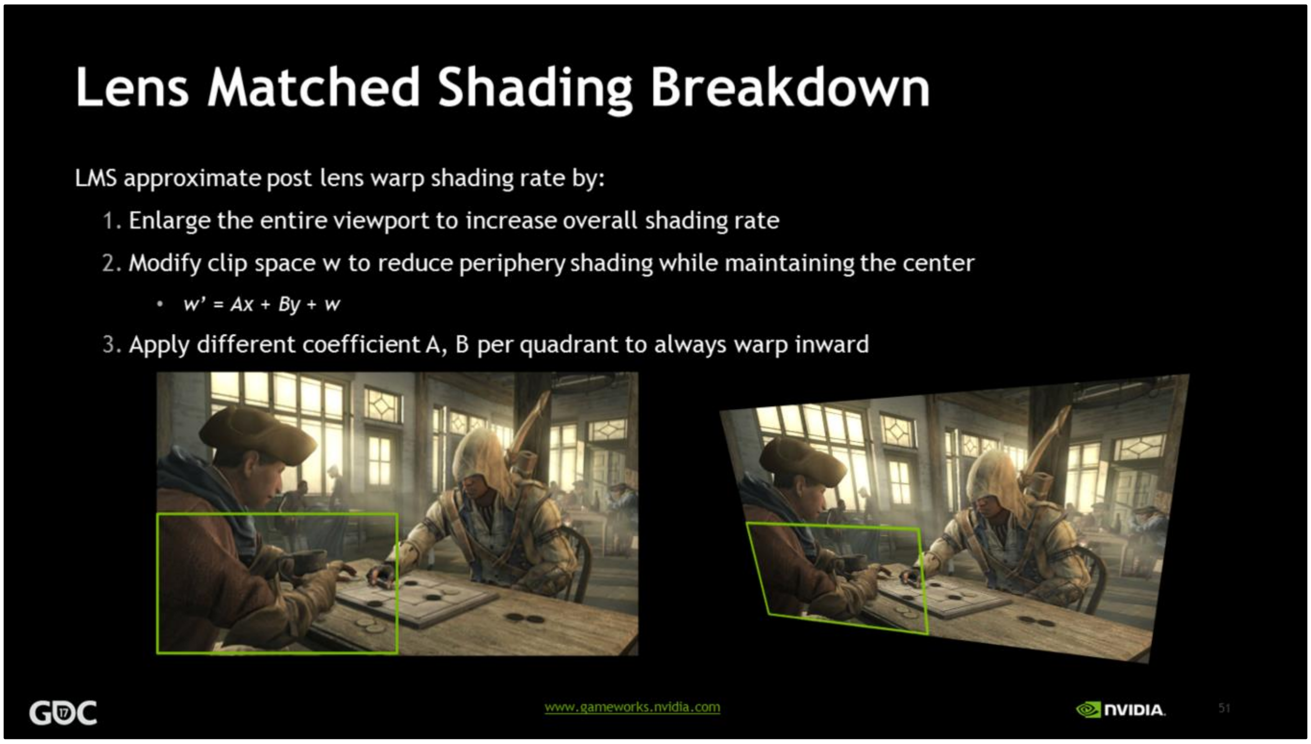
1. Enlarge the entire viewport to increase overall shading rate
2. Modify clip space w to reduce periphery shading while maintaining the center
   - $w' = Ax + By + w$
3. Apply different coefficient A, B per quadrant to always warp inward



GDC 17 www.gameworks.nvidia.com NVIDIA 50

each assigned different warp coefficients. And LMS uses the multiprojection feature to very efficiently broadcast geometries into those four viewports.

Notice that this will not change the shading rate at all at the very center pixel since x and y are zero, and the shading rate is gradually reduced at a 1 / x rate.



## Lens Matched Shading Breakdown

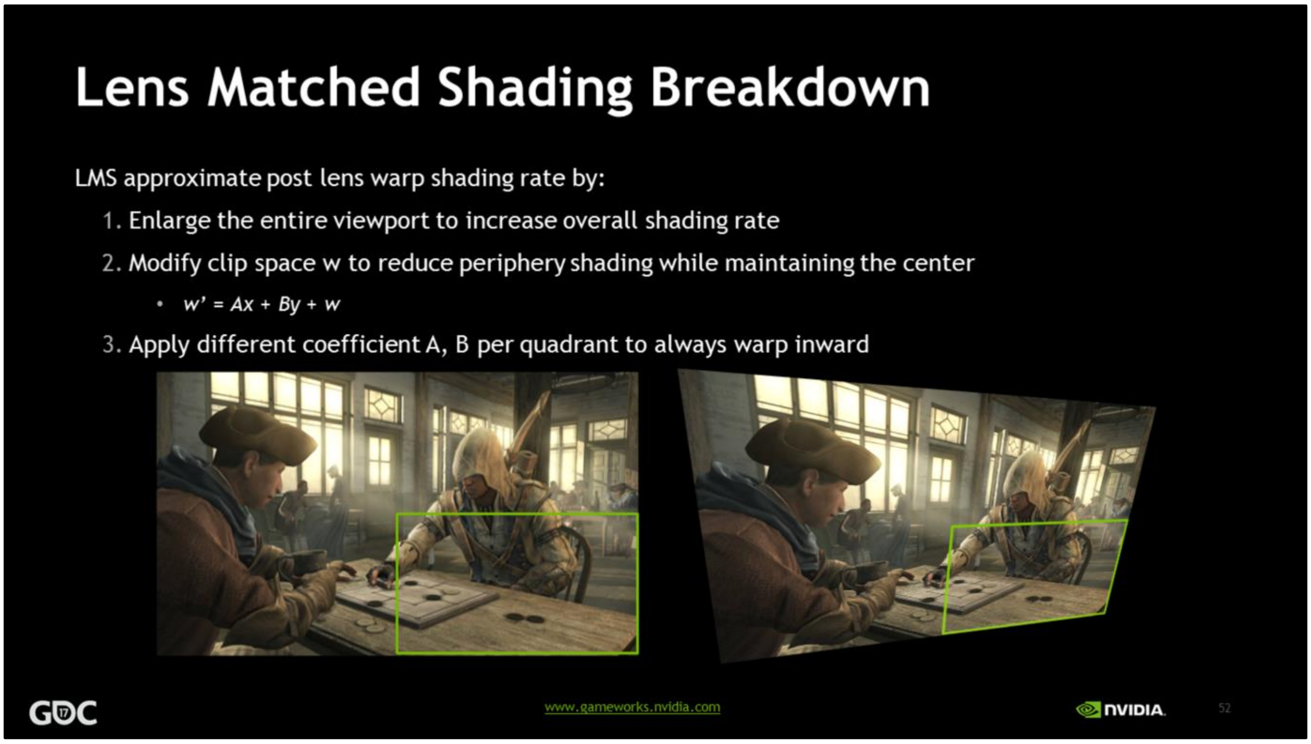
LMS approximate post lens warp shading rate by:

1. Enlarge the entire viewport to increase overall shading rate
2. Modify clip space w to reduce periphery shading while maintaining the center
   - *$w' = Ax + By + w$*
3. Apply different coefficient A, B per quadrant to always warp inward

GDC
www.gameworks.nvidia.com
NVIDIA
51

each assigned different warp coefficients. And LMS uses the multiprojection feature to very efficiently broadcast geometries into those four viewports.

Notice that this will not change the shading rate at all at the very center pixel since x and y are zero, and the shading rate is gradually reduced at a 1 / x rate.



## Lens Matched Shading Breakdown

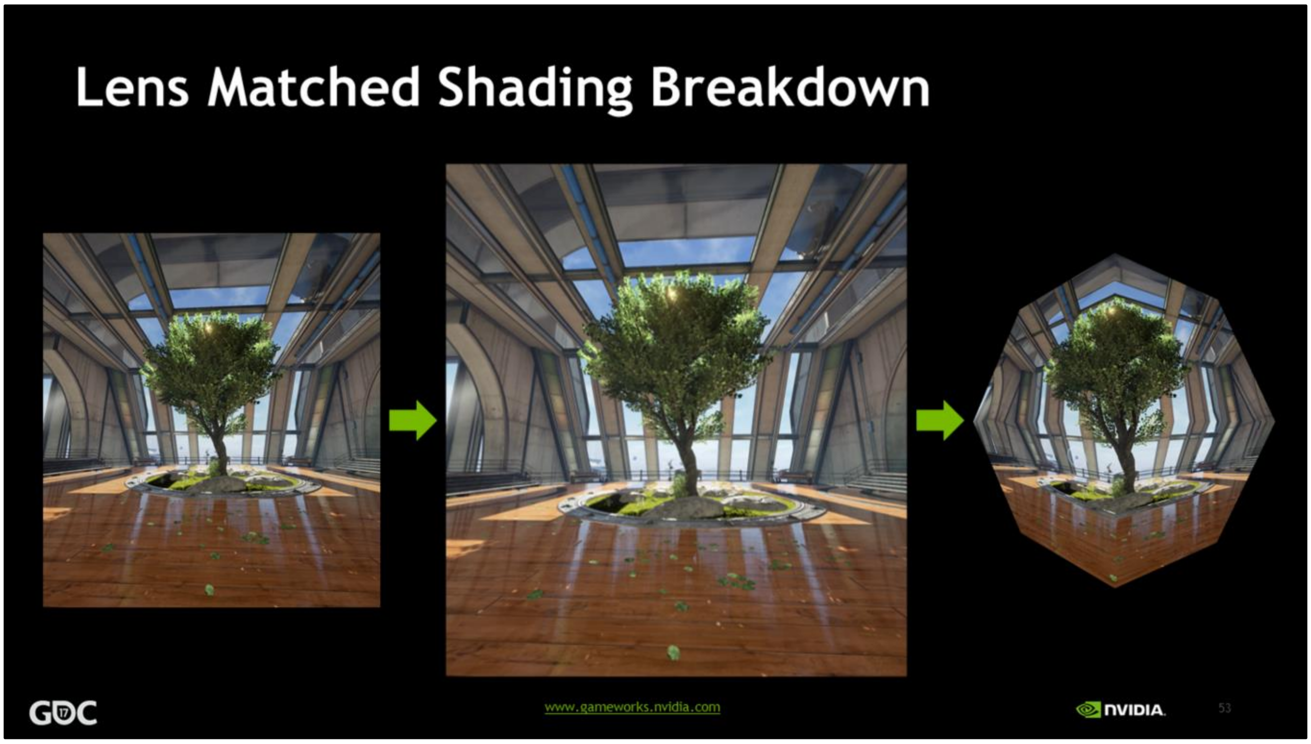
LMS approximate post lens warp shading rate by:

1. Enlarge the entire viewport to increase overall shading rate
2. Modify clip space w to reduce periphery shading while maintaining the center
    - $w' = Ax + By + w$
3. Apply different coefficient A, B per quadrant to always warp inward

GDC www.gameworks.nvidia.com NVIDIA 52

each assigned different warp coefficients. And LMS uses the multiprojection feature to very efficiently broadcast geometries into those four viewports.

Notice that this will not change the shading rate at all at the very center pixel since x and y are zero, and the shading rate is gradually reduced as you move towards the edge of the image.



## Lens Matched Shading Breakdown

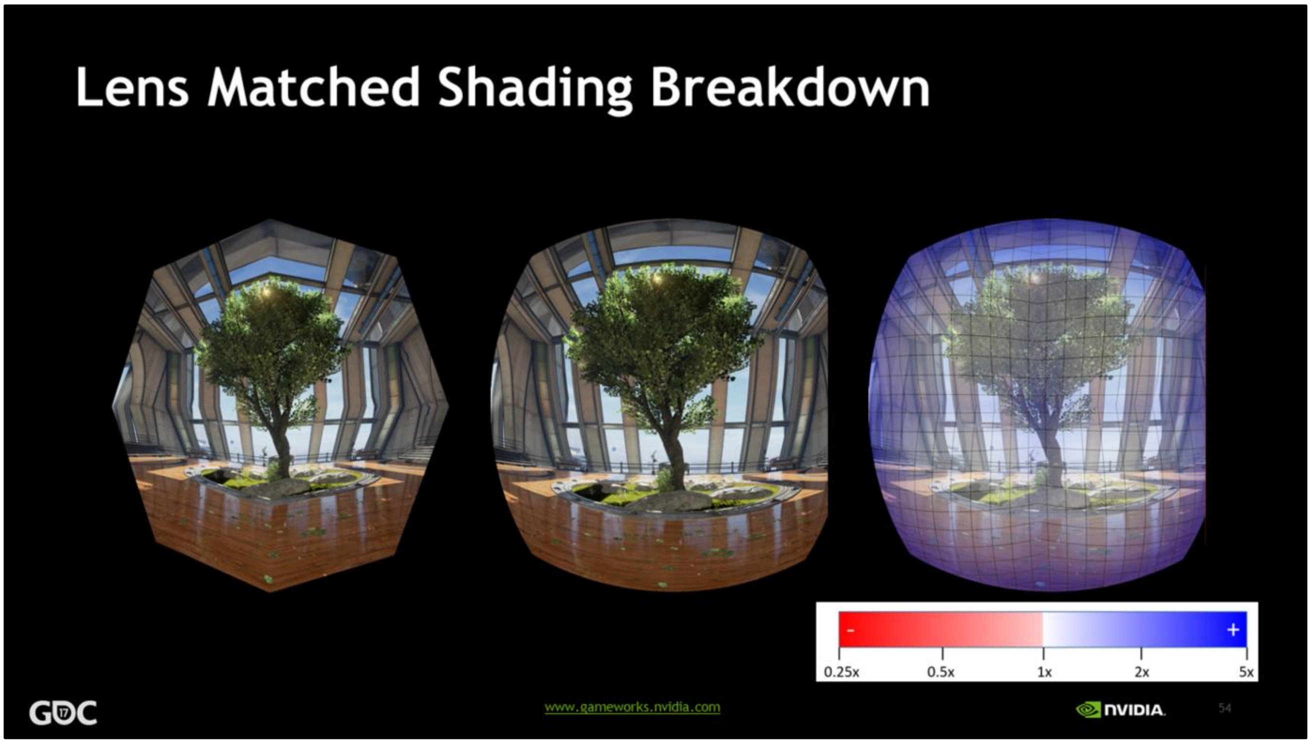
GDC

www.gameworks.nvidia.com

NVIDIA

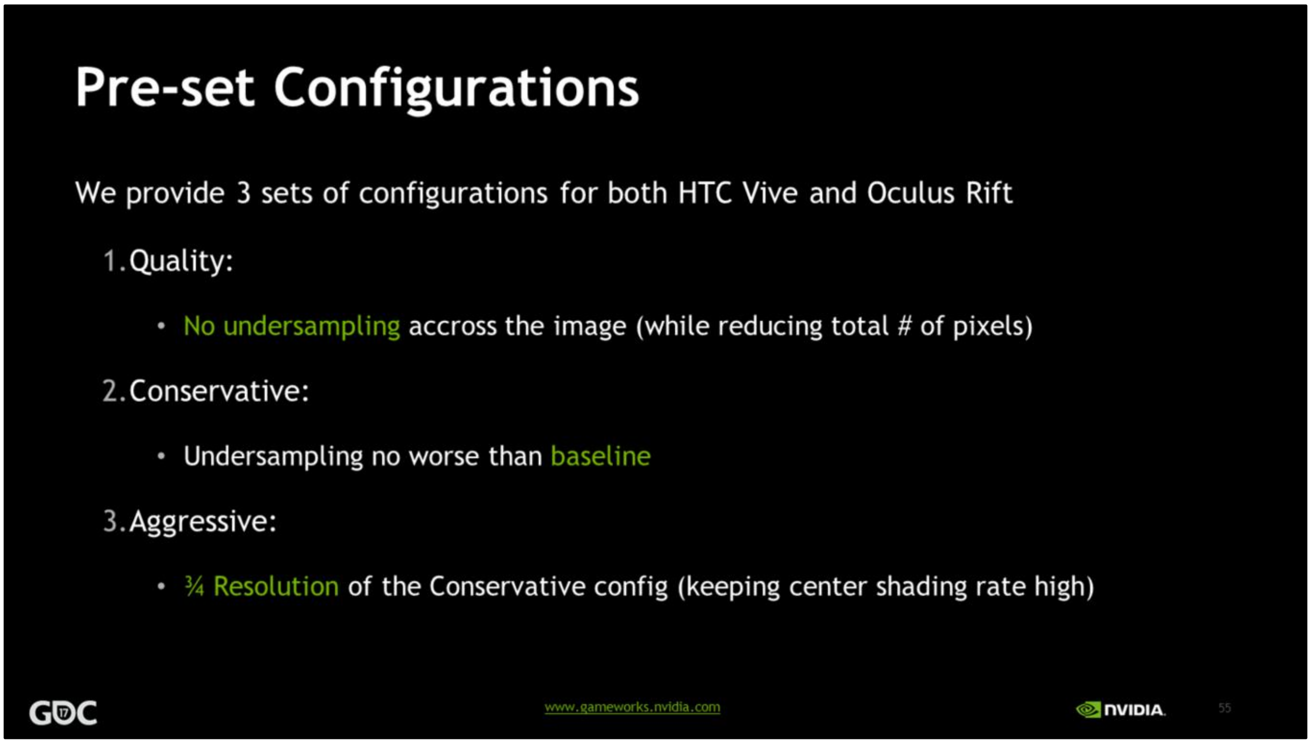
53

So the transformation that LMS does makes the shading rate distribution much more closer to the lens warped image.
(Click)First it renders the image at a higher resolution, this increases shading rate everywhere.
(Click)Then it applies modified w, which in a sense keeps the center shading rate high, and only reduce the shading rate in peripheral region!



## Lens Matched Shading Breakdown

Putting things side by side again, applying lens warp to a LMS transformed image produces a much more balanced shading rate distribution.
We can tell from the visualization that most of the region have a fairly close to 1x shading rate, and the undersampling that used to be in the middle is now gone.
This actually also results in a tremendously reduced shading rate, reducing the # of pixels shaded by 40%.



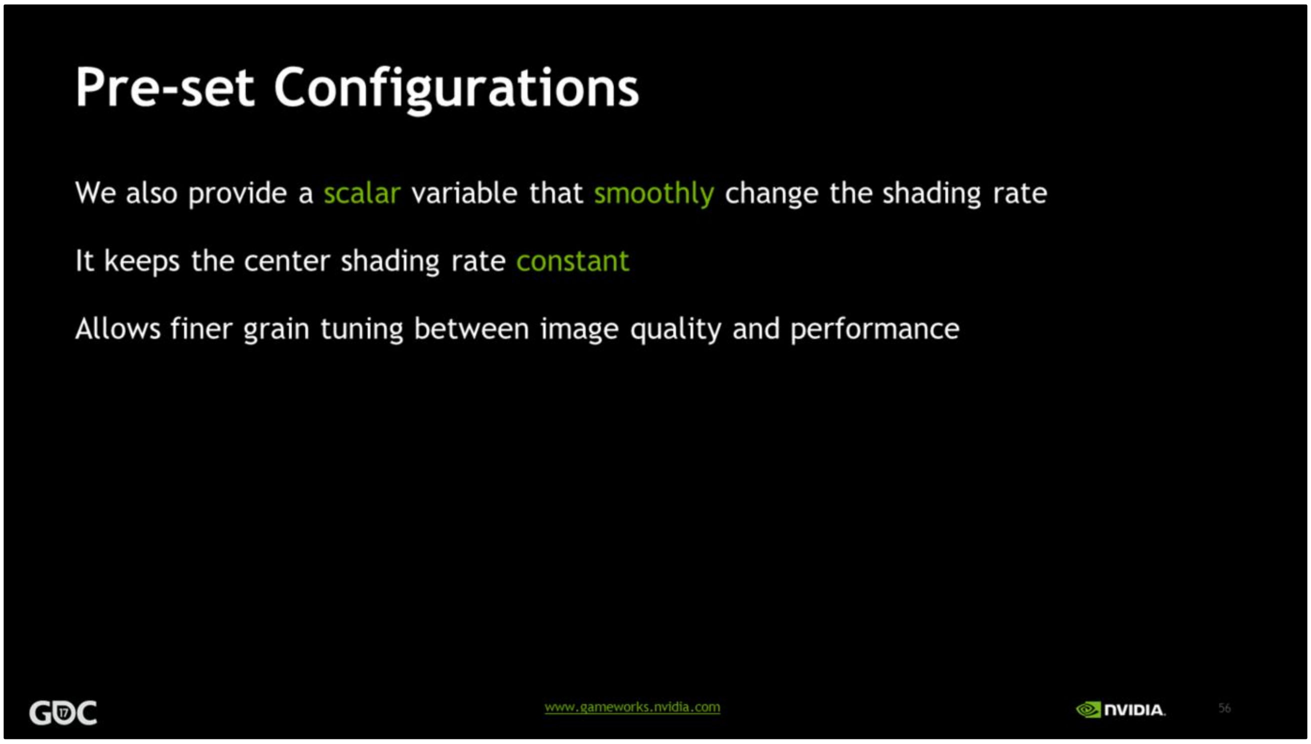
## Pre-set Configurations

We provide 3 sets of configurations for both HTC Vive and Oculus Rift

1. Quality:
   - No undersampling accross the image (while reducing total # of pixels)
2. Conservative:
   - Undersampling no worse than baseline
3. Aggressive:
   - ¾ Resolution of the Conservative config (keeping center shading rate high)

GDC 17 www.gameworks.nvidia.com NVIDIA 55

Obviously different warp coefficients produces different shading rate distribution. In addition, different lens warp function should be matched with different warp coefficients as well. And we've carefully calculated three sets of configurations for both HTC Vive and Oculus Rift.
The first set we call Quality. Using that will make sure there's no undersampling across the entire image.
The second set is Conservative. It allows undersampling, but no worse than the vanilla rendering shading rate.
The third is aggressive, it simply reduces another 25% of the pixels rendered compared with Conservative. Notice the reduced pixels are from peripheral region, so the center is not affected.



## Pre-set Configurations

We also provide a scalar variable that smoothly change the shading rate

It keeps the center shading rate constant

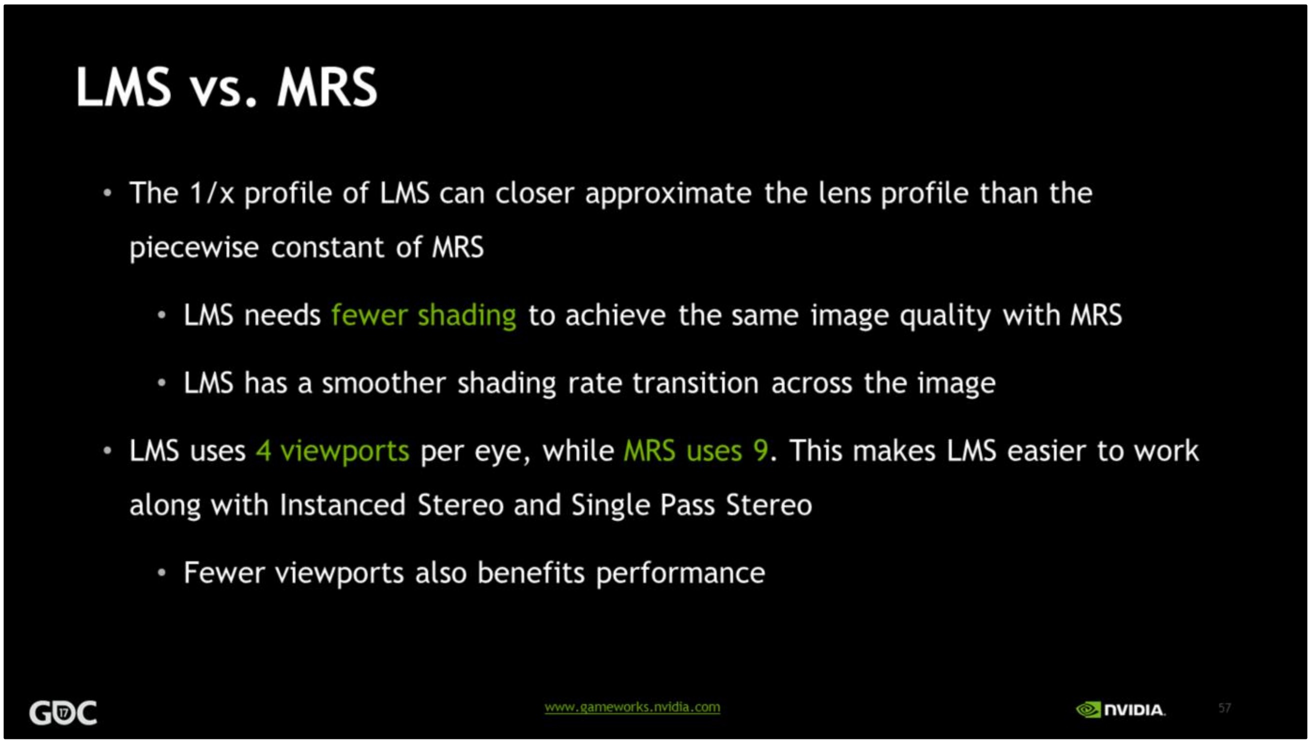
Allows finer grain tuning between image quality and performance

GDC 17

www.gameworks.nvidia.com

NVIDIA. 56

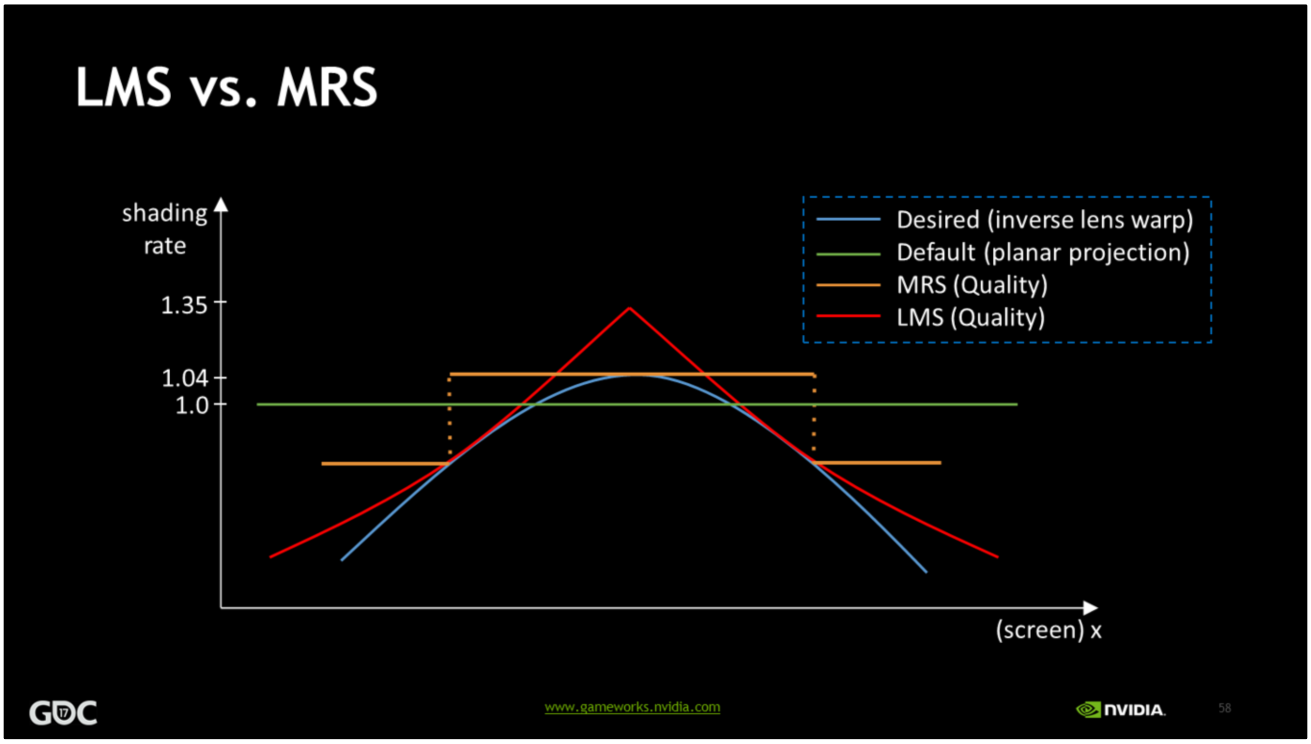
We've also provided a single scalar value that can fine tune the previously mentioned configurations, allowing the shading rate to get somewhere in between those configurations.


## LMS vs. MRS

- The 1/x profile of LMS can closer approximate the lens profile than the piecewise constant of MRS
  - LMS needs fewer shading to achieve the same image quality with MRS
  - LMS has a smoother shading rate transition across the image
- LMS uses 4 viewports per eye, while MRS uses 9. This makes LMS easier to work along with Instanced Stereo and Single Pass Stereo
  - Fewer viewports also benefits performance

GDC www.gameworks.nvidia.com NVIDIA 57

So a natural question to ask is how does LMS compare with MRS.
LMS is superior in many ways. The biggest difference is that MRS tries to approximate the lens warped shading rate with a piecewise constant profile, while LMS uses a 1/x profile.
Because of this, LMS is able to achieve the same image quality with MRS with less shading. Or put it in another way, if they do the same amount of shading, LMS would look slightly better.
There’s no hard shading rate transition across the rendered image. In addition, LMS uses 4 viewports, which makes it possible to be combined with Single Pass Stereo, another VRWorks feature introduced in pascal. Working with Instanced Stereo is also easier.
Finally, fewer viewports also means less culling in the fast geometry shader and therefore performance is benefited as well.

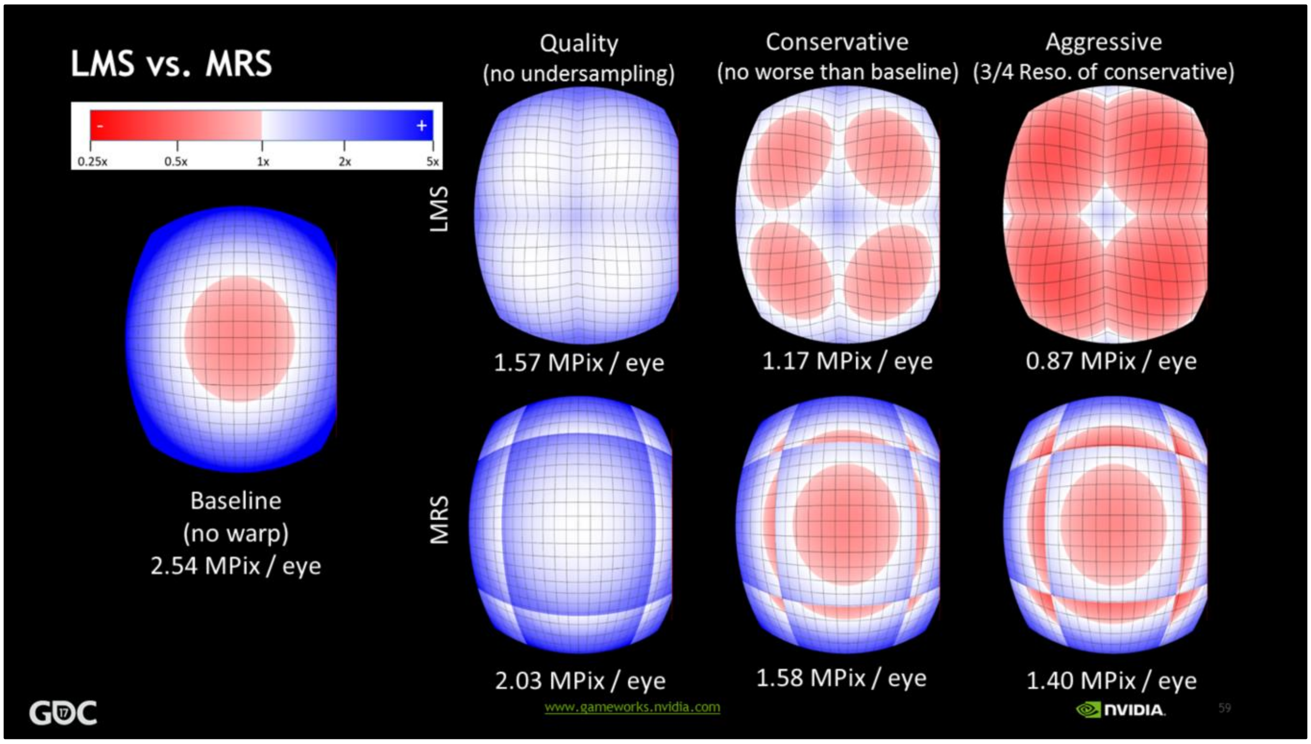


## LMS vs. MRS

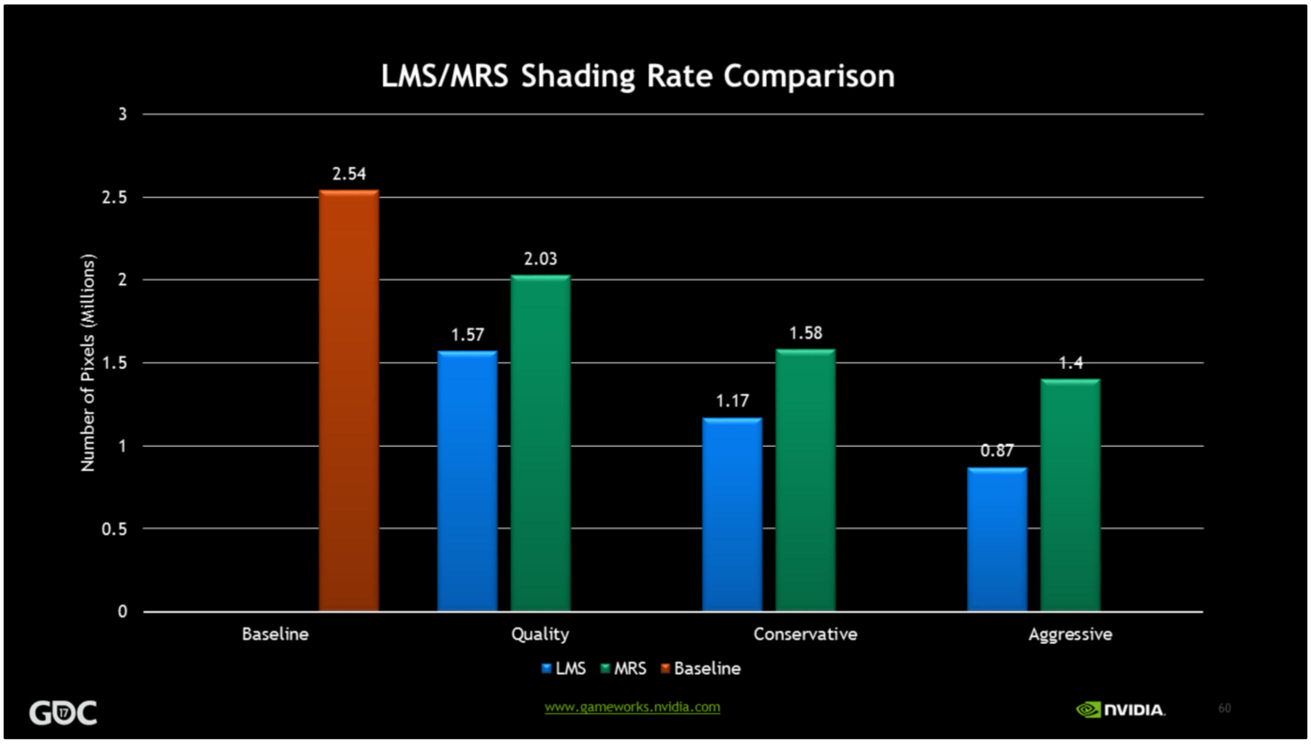
www.gameworks.nvidia.com

This is a 1D shading rate profile along screen space x axis. We've visualized shading rates for lens warp, LMS, MRS and baseline. This should give you an intuition when comparing the LMS shading rate profile with the piecewise constant profile provided by MRS.

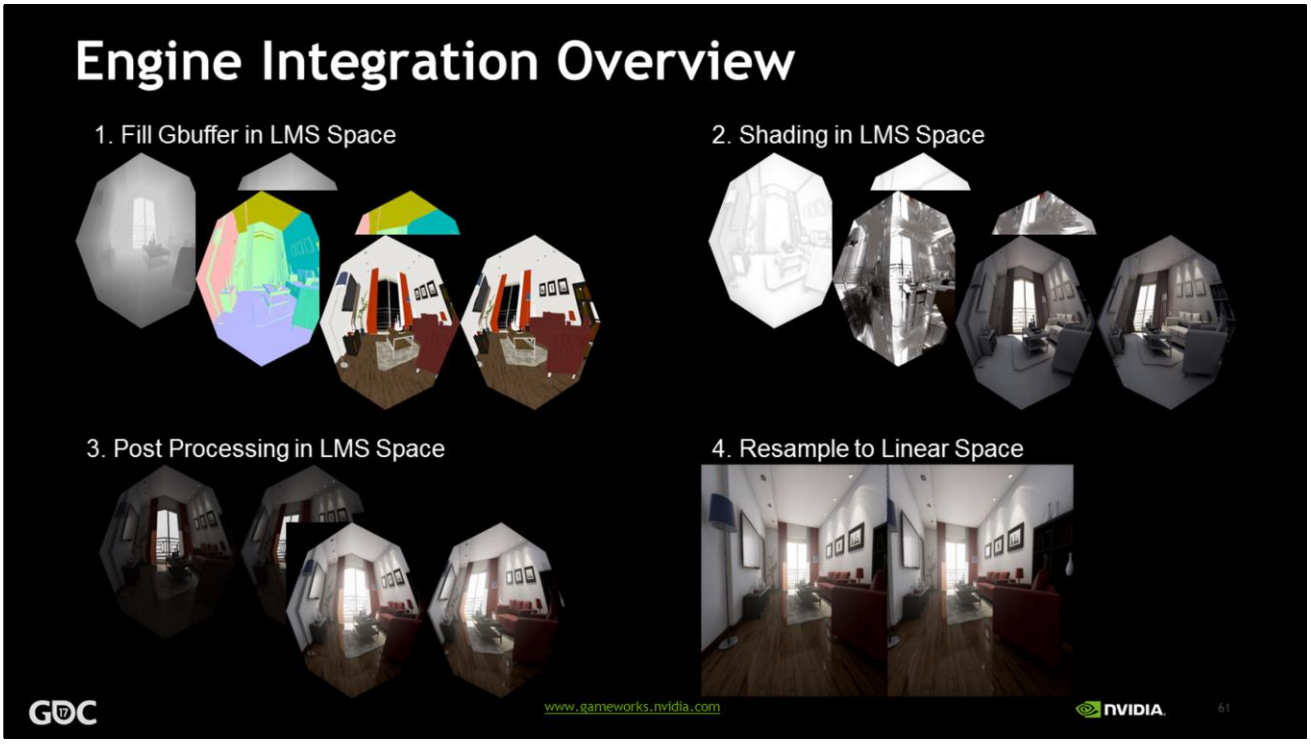
The baseline is shown as the green line. Notice how it undersamples the central region compared with the lens warped shading rate shown in blue. LMS is represented as the red line, it actually supersamples the central region more than MRS does. But the total number of pixels shaded by LMS is still fewer than MRS because it shades much less in the peripheral region.



This slide visualizes the shading rate of all LMS and MRS configurations across the entire view.
We can clearly see that across the spectrum LMS is rendering much less pixels compared with MRS.
In addition, MRS produces a hard shading rate transition at videport boundaries, while LMS has a very smooth shading rate transition across the whole image.
Another thing to notice is that LMS always has higher shading rate at the center part of the image, which is what the viewers look at most of the time and therefore it can increase the perceived image quality.



We can estimate the comparative performance gains by computing the number of pixels shaded for matching LMS and MRS configurations. The lower the number of pixels shaded, the better the performance. While both MRS already provides a decent reduction in shading workloads, LMS shades even fewer pixels than MRS across the configuration spectrum.

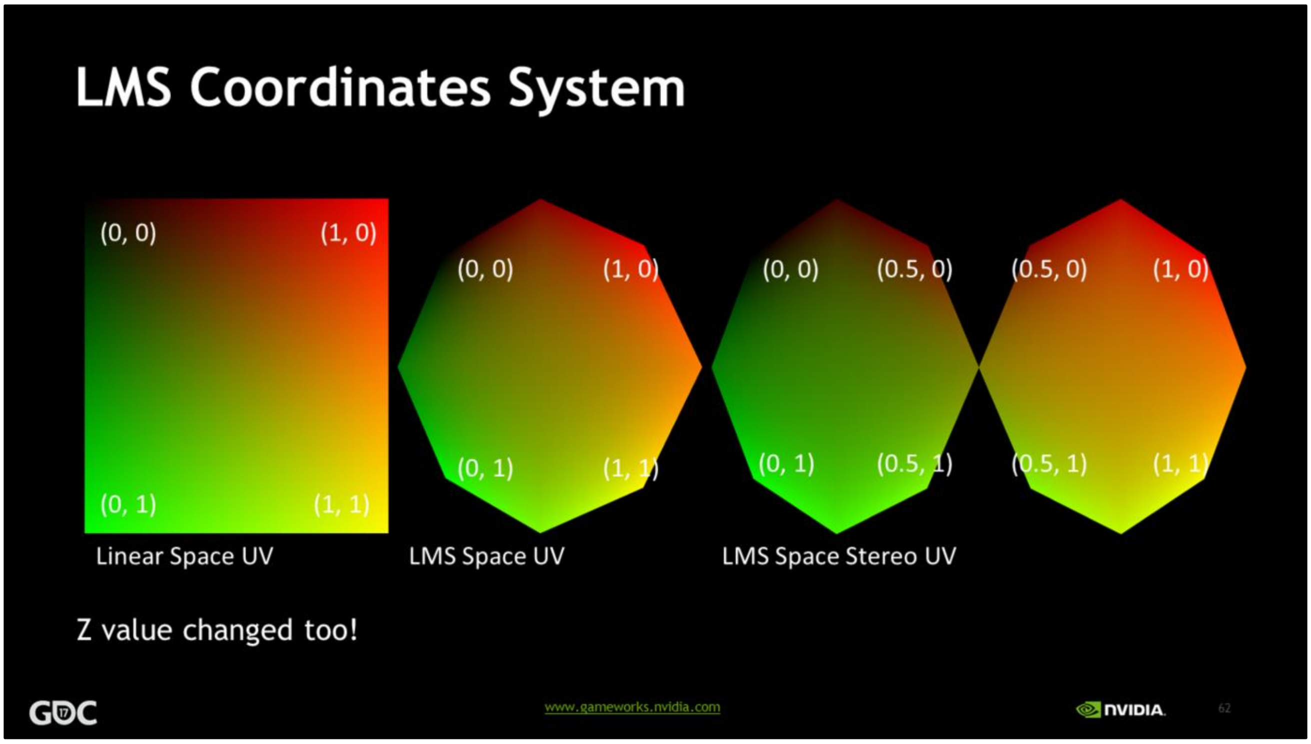


## Engine Integration Overview

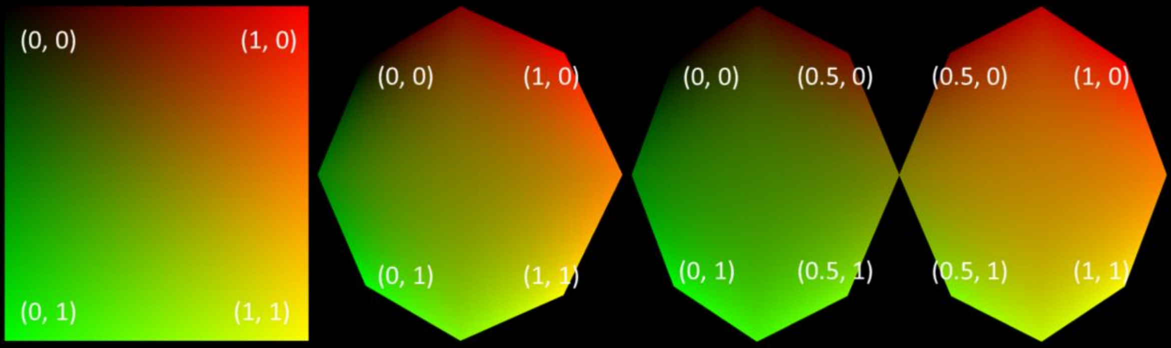
1. Fill Gbuffer in LMS Space
2. Shading in LMS Space
3. Post Processing in LMS Space
4. Resample to Linear Space
GDC
www.gameworks.nvidia.com
NVIDIA
61

We've said enough good things about LMS, and next let's look at what does it take to put LMS in a modern game engine. We have put LMS into Unreal Engine 4 and achieved a good performance and quality bump. Hopefully our work can serve as an example for developers interested in using LMS in their own engines.

So, the overall idea of using LMS in any deferred renderer is very simple. (Click) Firstly you have to draw your basepass with w modification on. And all of your Gbuffer (Normal, depth, albedo etc) will all look like those cute octagons, which we call LMS space. (Click) Then, all the shading should also ideally be performed in LMS space. Since it has less amount of pixels to shade there. (Click)And similarly, post processing should be done in LMS space too. (Click)Finally, the octagon shaped buffer is resampled back to linear before submitting it to HMD runtime, which will then perform lens warp on it.



## LMS Coordinates System

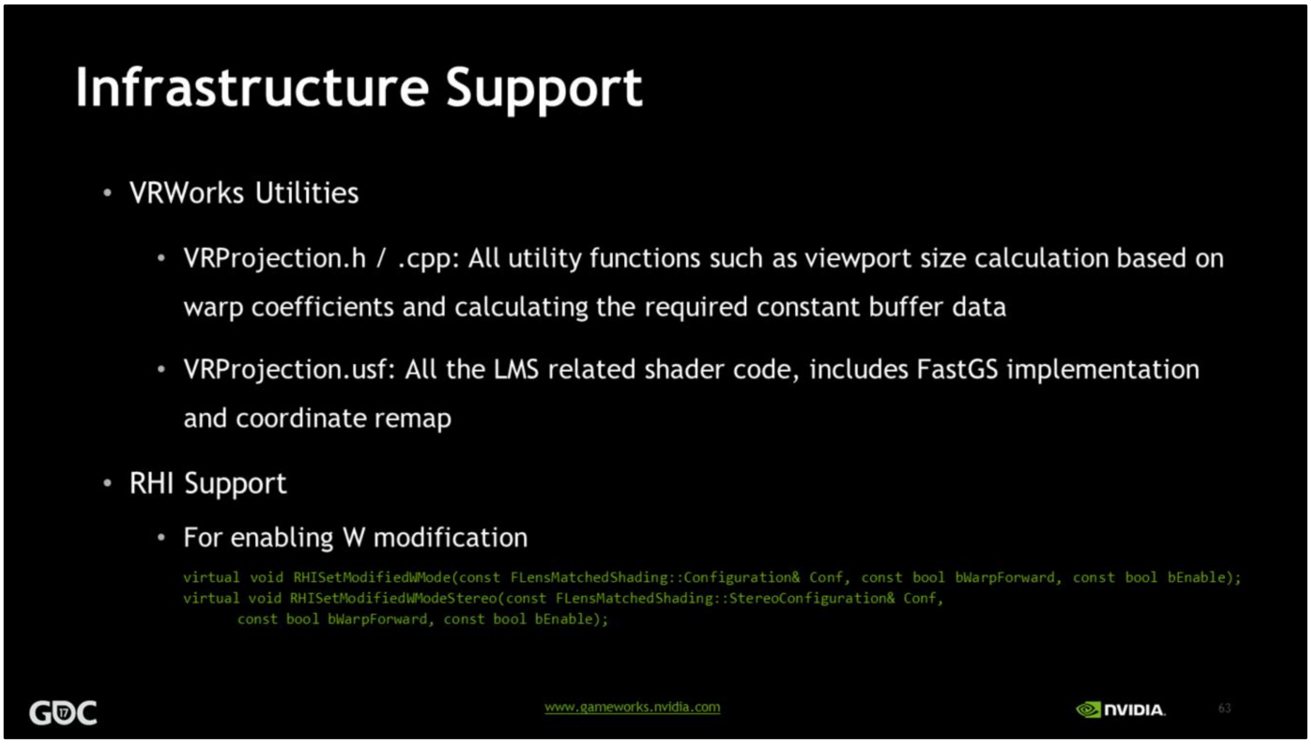


Z value changed too!

GDC www.gameworks.nvidia.com NVIDIA 62

Among other things, a really difficult part of integrating LMS is dealing with the LMS space it introduces. Existing shaders must be carefully adjusted. Because LMS changes the definition of the screen-space coordinate system, we have to be really careful with determining whether a given coordinate is in the original linear space, or the octagon-shaped LMS space when using them in the shaders. When LMS is combined with stereo rendering there is an additional layer of complexity to coordinate conversion since we also need to convert between view space and render target space.

In LMS coordinate space, the depth value of each fragment needs to be adjusted because the w is modified before perspective division. Whenever we need to fetch depth from the depth buffer for things like reconstructing world space position, we also need to remap the Z value fetched to linear space.



## Infrastructure Support

- VRWorks Utilities
  - VRProjection.h / .cpp: All utility functions such as viewport size calculation based on warp coefficients and calculating the required constant buffer data
  - VRProjection.usf: All the LMS related shader code, includes FastGS implementation and coordinate remap
- RHI Support
  - For enabling W modification

```
virtual void RHISetModifiedWMode(const FLensMatchedShading::Configuration& Conf, const bool bWarpForward, const bool bEnable);
virtual void RHISetModifiedWModeStereo(const FLensMatchedShading::StereoConfiguration& Conf,
      const bool bWarpForward, const bool bEnable);
```

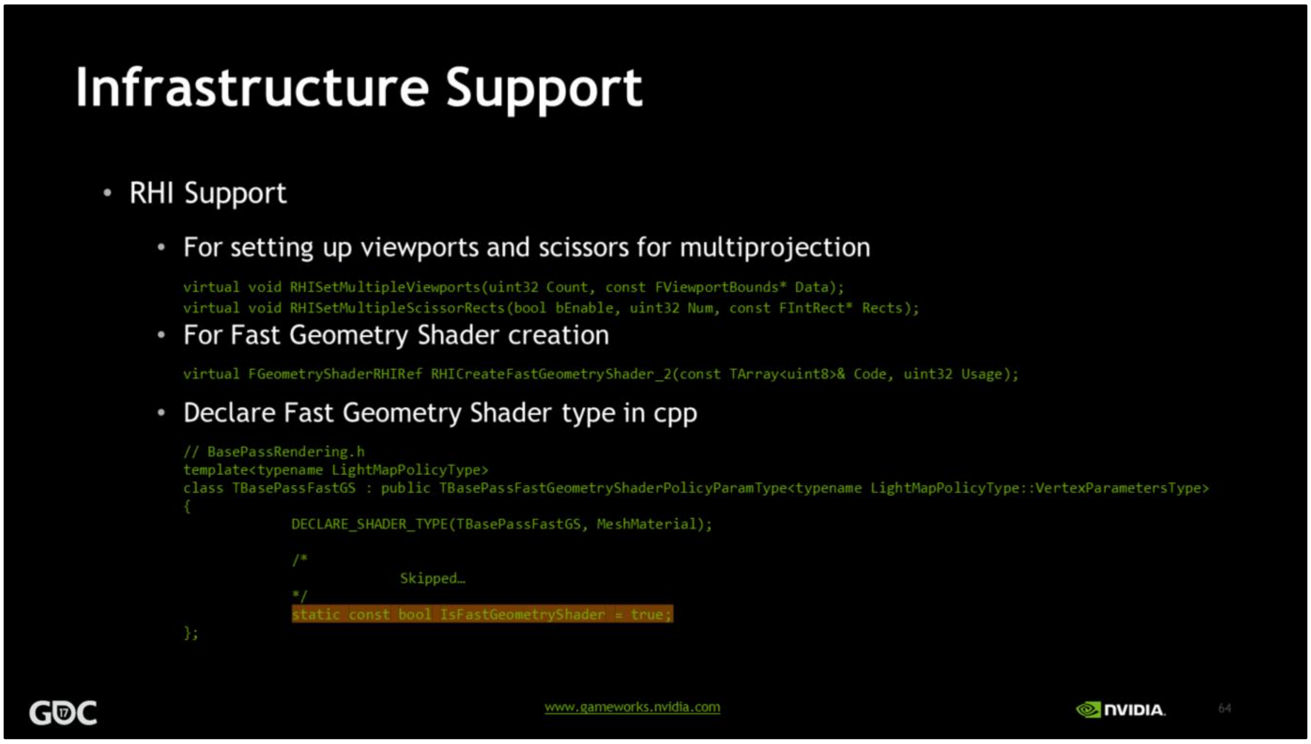
GDC

www.gameworks.nvidia.com

NVIDIA 63

First let’s go over some of the VRWorks utilities functions and new RHIs that we have added for integrating VRWorks.
We have put everything related to the configurations, viewport, scissor calculations and calculating the newly required constant buffer data into VRProjection.h and .cpp.
In addition, VRProjection.usf is the shader file where we have put all the helper functions for Coordinate remap. You will find them be named as MapLinearToVRProj or the otherway around.
It also includes FastGS implementation, which is needed by the multi viewport projection.

On the RHI side, two new functions were added to enable modified w mode, which basically pass the A, B coefficients stored in Conf and calls NvAPI underneath.



## Infrastructure Support

- RHI Support
  - For setting up viewports and scissors for multiprojection

```
virtual void RHISetMultipleViewports(uint32 Count, const FViewportBounds* Data);
virtual void RHISetMultipleScissorRects(bool bEnable, uint32 Num, const FIntRect* Rects);
```

  - For Fast Geometry Shader creation

```
virtual FGeometryShaderRHIRef RHICreateFastGeometryShader_2(const TArray<uint8>& Code, uint32 Usage);
```

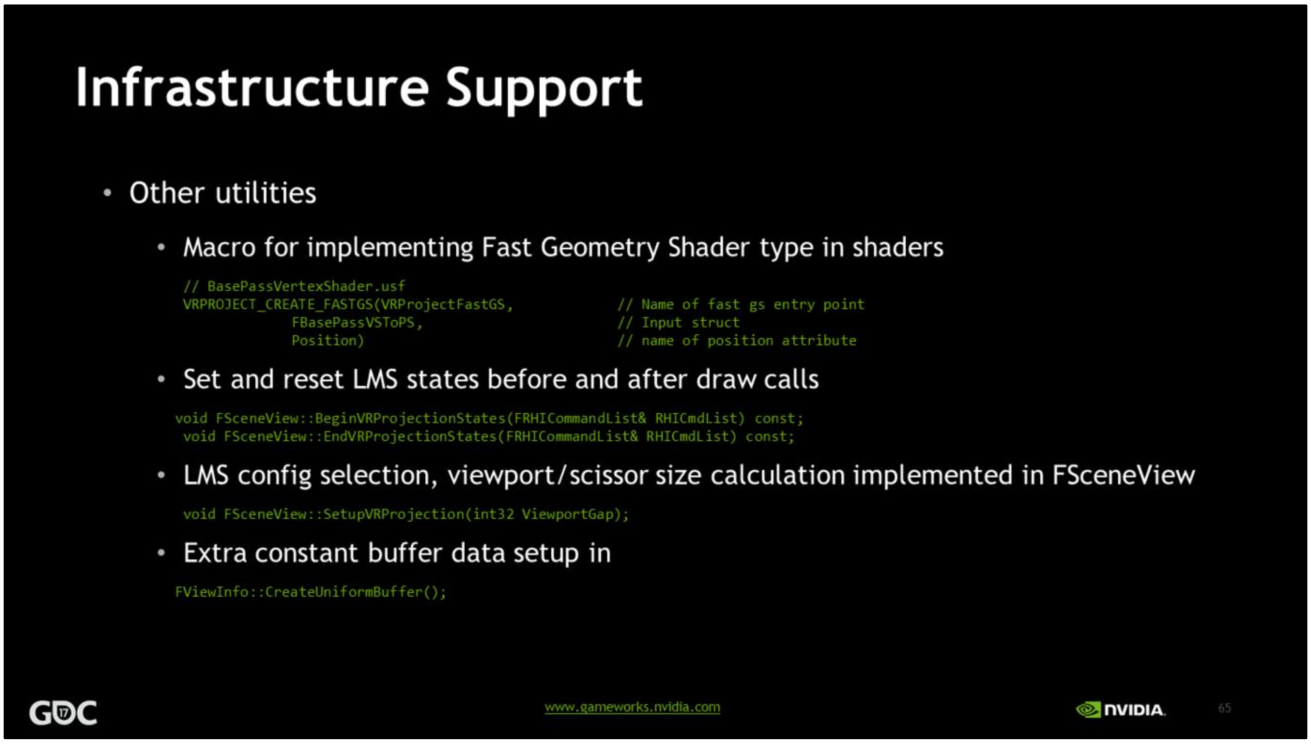
  - Declare Fast Geometry Shader type in cpp

```
// BasePassRendering.h
template<typename LightMapPolicyType>
class TBasePassFastGS : public TBasePassFastGeometryShaderPolicyParamType<typename LightMapPolicyType::VertexParametersType>
{
            DECLARE_SHADER_TYPE(TBasePassFastGS, MeshMaterial);

            /*
                        Skipped…
            */
            static const bool IsFastGeometryShader = true;
};
```

GDC www.gameworks.nvidia.com NVIDIA 64

We also need RHI level support for setting up multiple viewports, scissors, and declare fast GS for all the geometry types in UE4.



## Infrastructure Support

- Other utilities
  - Macro for implementing Fast Geometry Shader type in shaders

```
// BasePassVertexShader.usf
VRPROJECT_CREATE_FASTGS(VRProjectFastGS,            // Name of fast gs entry point
             FBasePassVSToPS,                       // Input struct
             Position)                              // name of position attribute
```

  - Set and reset LMS states before and after draw calls

```
void FSceneView::BeginVRProjectionStates(FRHICommandList& RHICmdList) const;
 void FSceneView::EndVRProjectionStates(FRHICommandList& RHICmdList) const;
```

  - LMS config selection, viewport/scissor size calculation implemented in FSceneView

```
void FSceneView::SetupVRProjection(int32 ViewportGap);
```

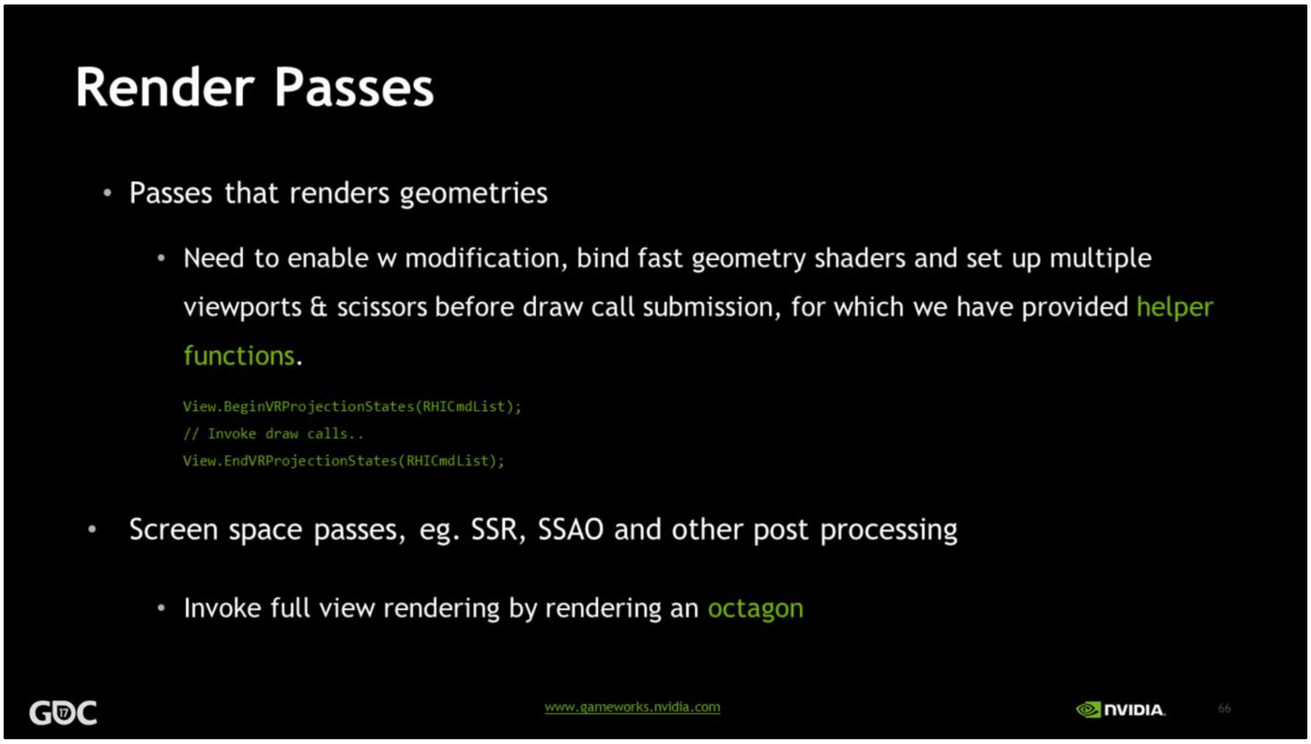
  - Extra constant buffer data setup in

```
FViewInfo::CreateUniformBuffer();
```

www.gameworks.nvidia.com

A macro is provided for implementing fast GS in shaders. It only takes an input and and output struct, and the name of the position attribute. It will expand to the fast GS that does multiprojection for you. They are in vertex shader files.
We also provided helpers to quickly disable and enable LMS related states.

Obviously lots of implementation details are glossed over, I don’t want to turn this talk into a code review, Please refer to source for details.


## Render Passes

- Passes that renders geometries
  - Need to enable w modification, bind fast geometry shaders and set up multiple viewports & scissors before draw call submission, for which we have provided helper functions.

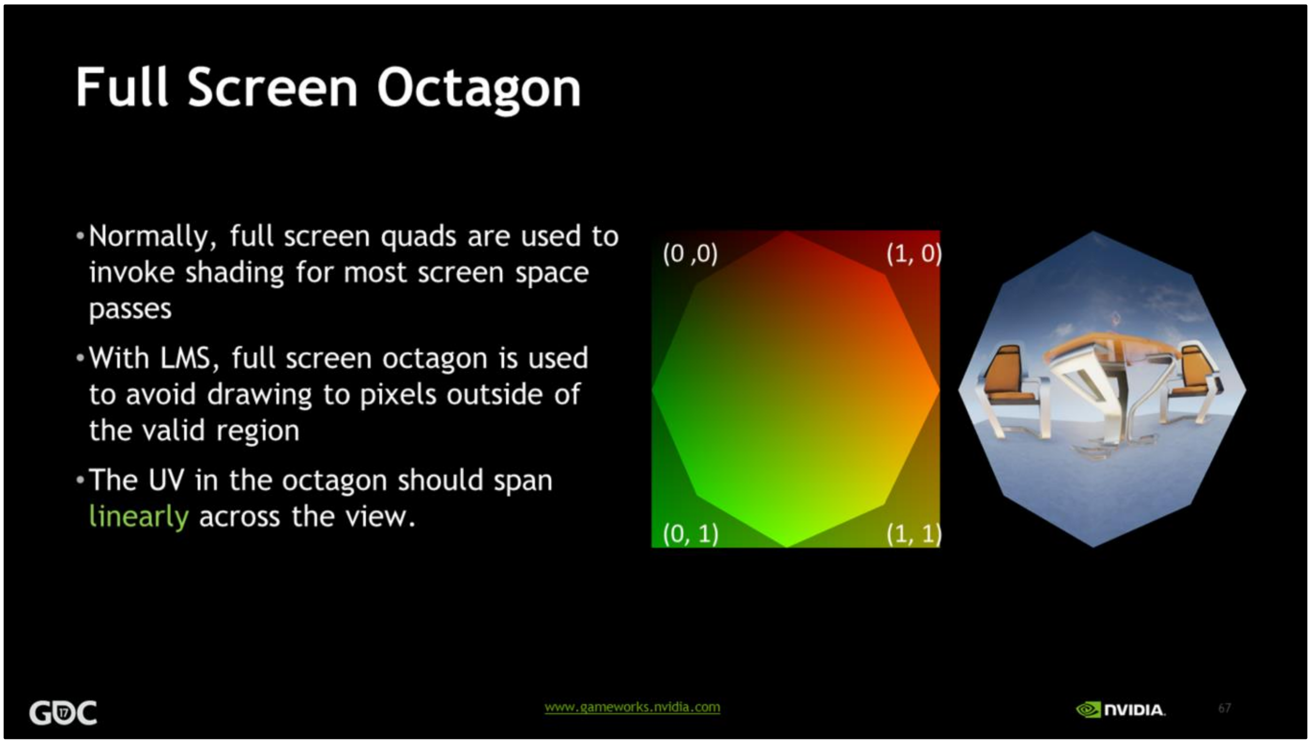
```
View.BeginVRProjectionStates(RHICmdList);
// Invoke draw calls..
View.EndVRProjectionStates(RHICmdList);
```

- Screen space passes, eg. SSR, SSAO and other post processing
  - Invoke full view rendering by rendering an octagon

GDC www.gameworks.nvidia.com NVIDIA 66

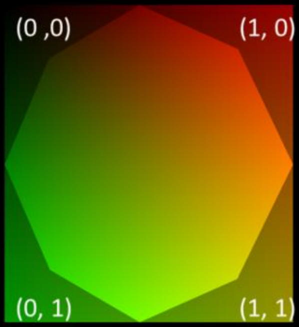
Now let’s look at different render passes. Broadly speaking there are two types of rendering passes.
There are passes that renders geometries to the screen, including the base pass, depth pre-pass, deferred lighting, shadow map projection, decals and so on. For these, we need to bind the fast geometry shader, set multiple viewports, scissors and enable w modification before submitting the geometry. We have provided Begin/EndVRProjectionStates helper functions for these. So in practice we just call those helper functions before and after the draw call we are modifying.

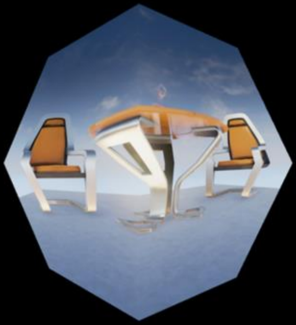
Screen space effects is the other type of pass, common examples include SSR, SSAO and other post processing passes. For most screen space passes, we invoke shading with a full screen octagon.



## Full Screen Octagon

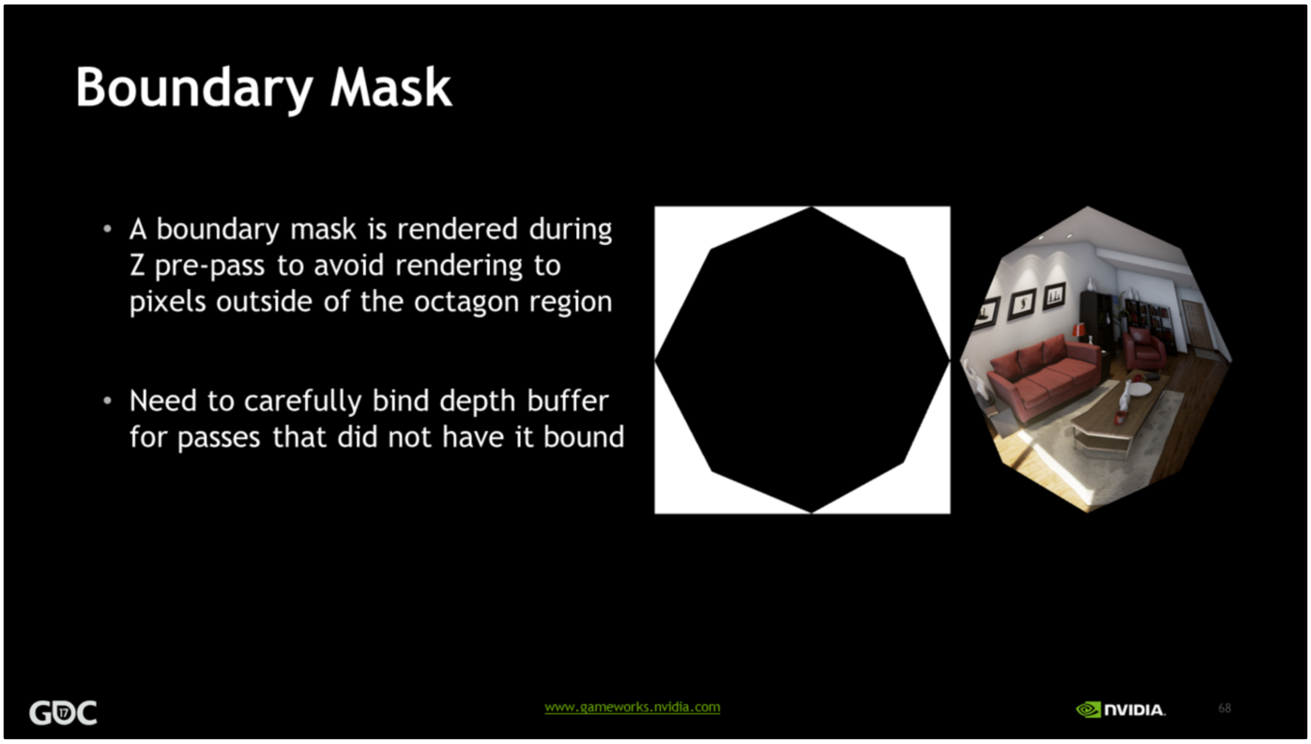
- Normally, full screen quads are used to invoke shading for most screen space passes
- With LMS, full screen octagon is used to avoid drawing to pixels outside of the valid region
- The UV in the octagon should span linearly across the view.





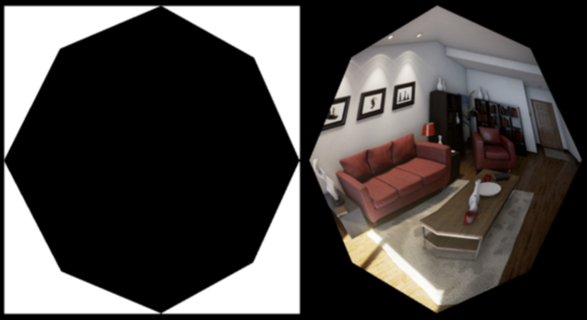
GDC www.gameworks.nvidia.com NVIDIA 67

So what's a full screen octagon. Normally UE4 draws a full screen quad covering the entire view to invoke most post processing passes. With LMS enabled, a full screen quad would cover those corner pixels outside of our octagon shaped buffer. Therefore we draw a full screen octagon that covers the exact same area as the octagon in the underlying texture instead. A full screen octagon is just a carved out octagon portion of the original full screen quad. So the UV still spans linearly inside the octagon.



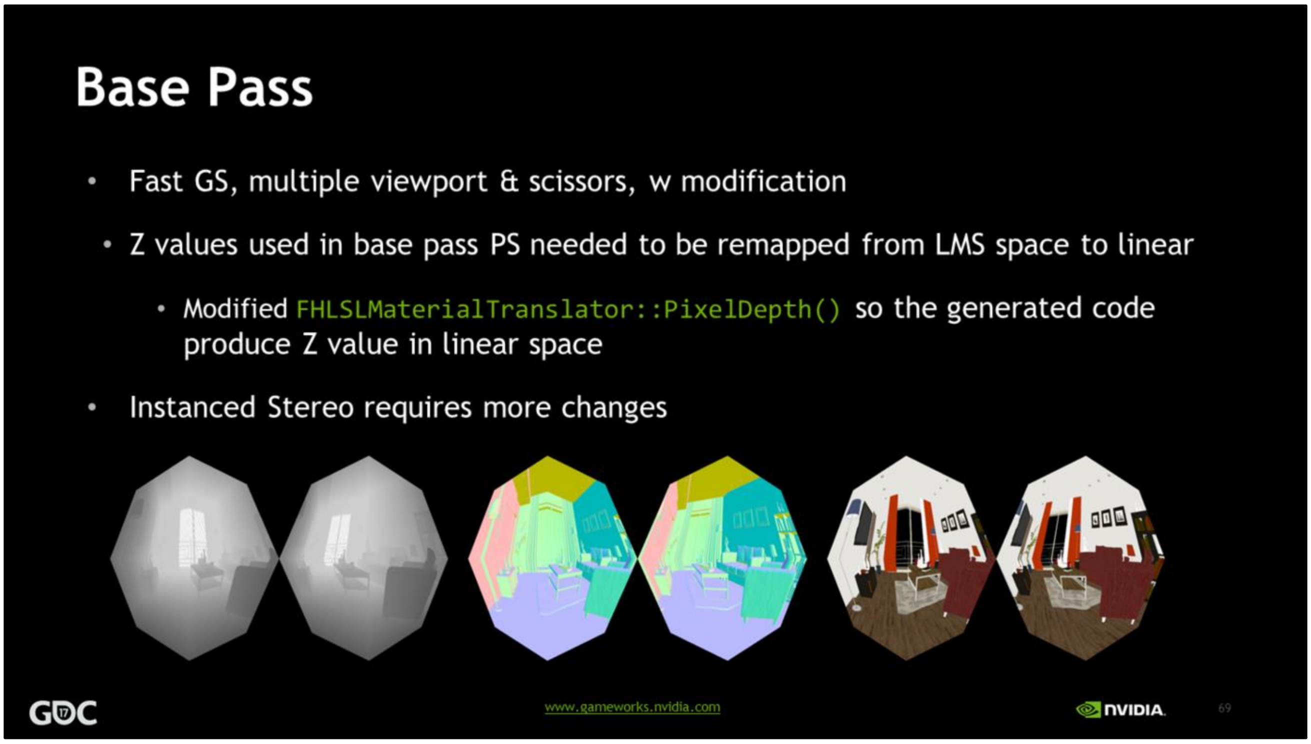
## Boundary Mask

- A boundary mask is rendered during Z pre-pass to avoid rendering to pixels outside of the octagon region
- Need to carefully bind depth buffer for passes that did not have it bound



GDC www.gameworks.nvidia.com NVIDIA 68

While the full screen octagon is responsible for culling those corner for screen space passes. For passes that render geometry, we instead rely on what we refer to as boundary mask to do the job.
Actually, applying w modification alone won't produce the octagon shaped buffer, we need a way to explicitly kill shadings out of the octagon shape in order to save those works.
Therefore during Z pre-pass, we render a boundary mask that set the Z values out side of the octagons to the closest, so that all subsequent passes will only write to pixels within the octagon. This also means that we need to carefully bind Z buffer in passes that originally don't have it bound.



## Base Pass

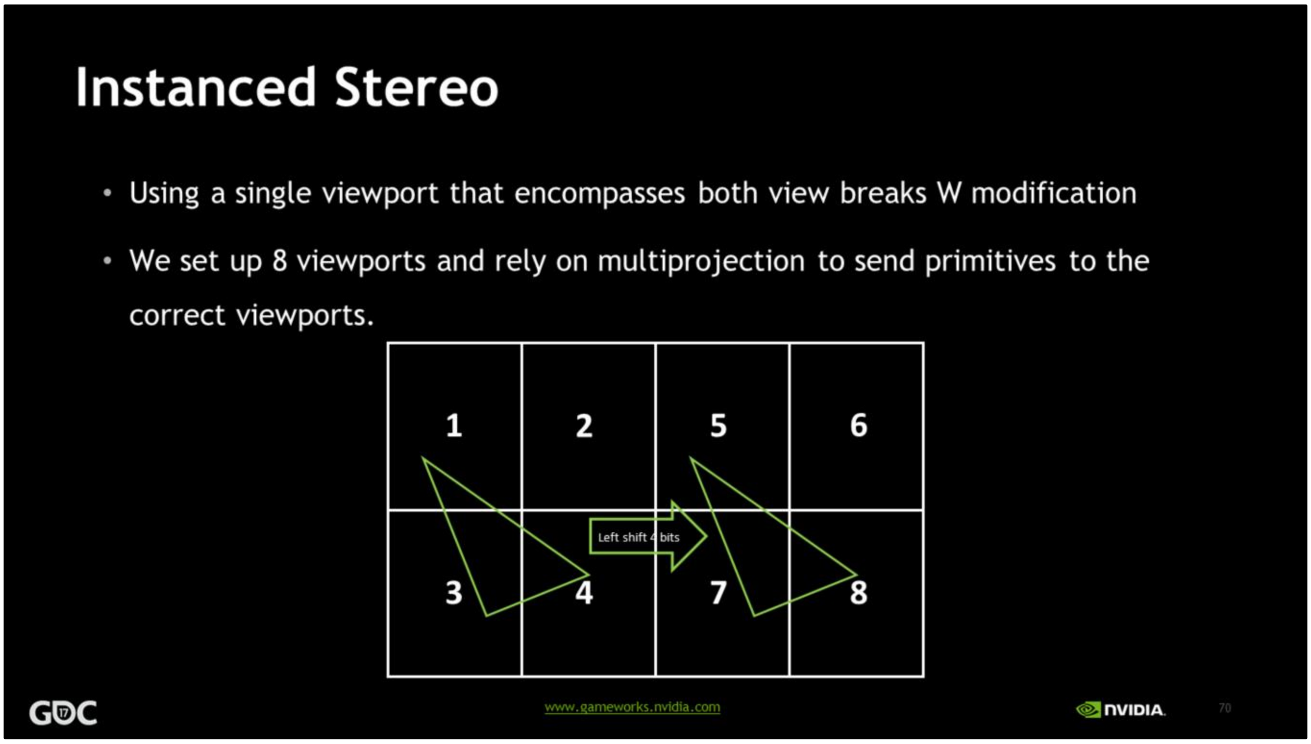
- Fast GS, multiple viewport & scissors, w modification
  - Z values used in base pass PS needed to be remapped from LMS space to linear
    - Modified `FHLSLMaterialTranslator::PixelDepth()` so the generated code produce Z value in linear space
- Instanced Stereo requires more changes

GDC
www.gameworks.nvidia.com
NVIDIA
69

Now let’s specifically look at base pass. We’ve actually covered all the ingredients already, just setup an additional FastGS for every primitive rendered, set up multiple viewports and scissors and also enable w modification.

Another thing we need to address is that, w modification also affects the output Z in base pass. Z value can be used in UE4’s material graph to do arbitrary things. And they assume the Z value is linear. So we also modified the material graph code generator to output code that convert Z from LMS space to linear space when LMS is enabled.

However, to work with Instanced, more changes are required.



## Instanced Stereo

- Using a single viewport that encompasses both view breaks W modification
- We set up 8 viewports and rely on multiprojection to send primitives to the correct viewports.

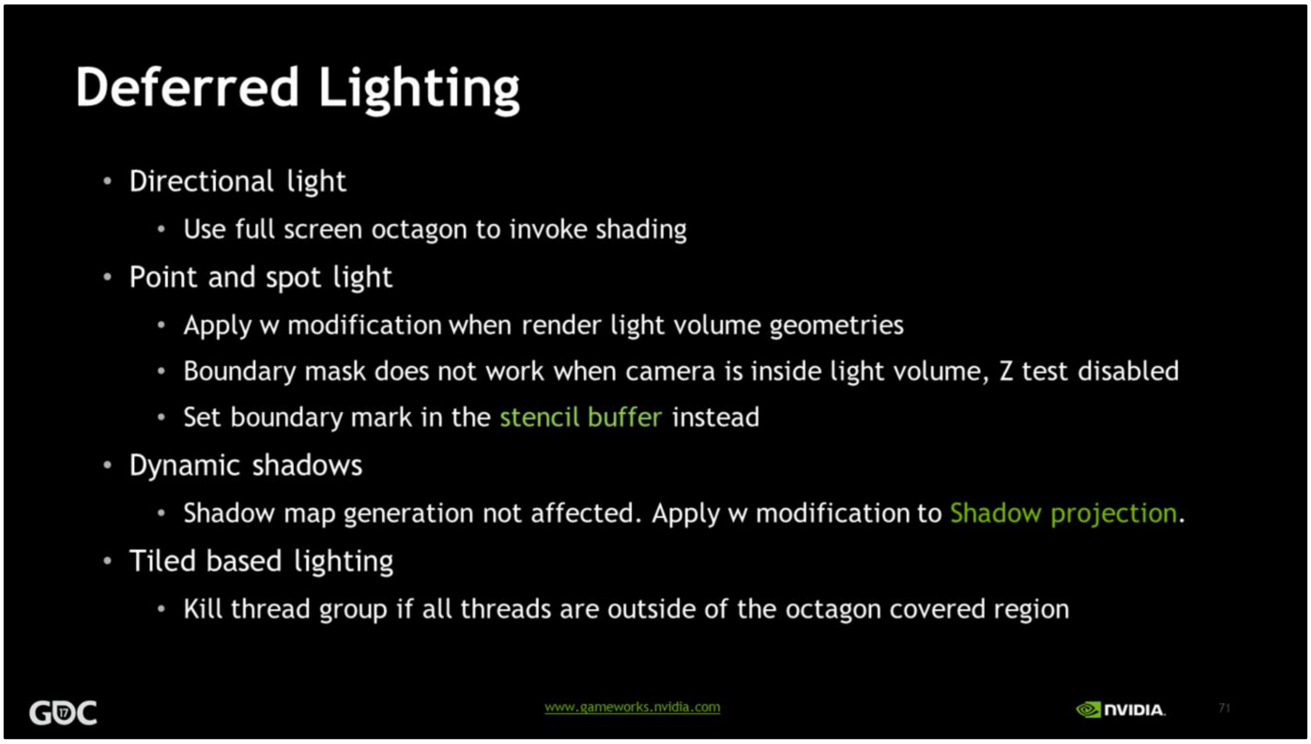
UE4 uses a single viewports that includes both the left and right views when instanced stereo is enabled, and it leverage vertex shaders to shift vertices to the left or right half of the view based on the view index.

Doing things this way breaks modified w, since it assumes the clip space spans [-1, 1] in x direction. And the shifting makes the left eye view span only [-1, 0] and the right eye span [0, 1] in clip space.

Fortunately the walk around is simple, just setup 8 viewports and scissors, 4 for the left view and 4 for the right, and rely on multiprojections to send primitives to the correct viewports.

This is done in FastGS. Just left shift the viewport mask for 4 bits if the view index indicates the primitive should be rendered to the right viewport.

This way each view still span the entire [-1, 1] space so w modification still works.



## Deferred Lighting

- Directional light
  - Use full screen octagon to invoke shading
- Point and spot light
  - Apply w modification when render light volume geometries
  - Boundary mask does not work when camera is inside light volume, Z test disabled
  - Set boundary mark in the stencil buffer instead
- Dynamic shadows
  - Shadow map generation not affected. Apply w modification to Shadow projection.
- Tiled based lighting
  - Kill thread group if all threads are outside of the octagon covered region

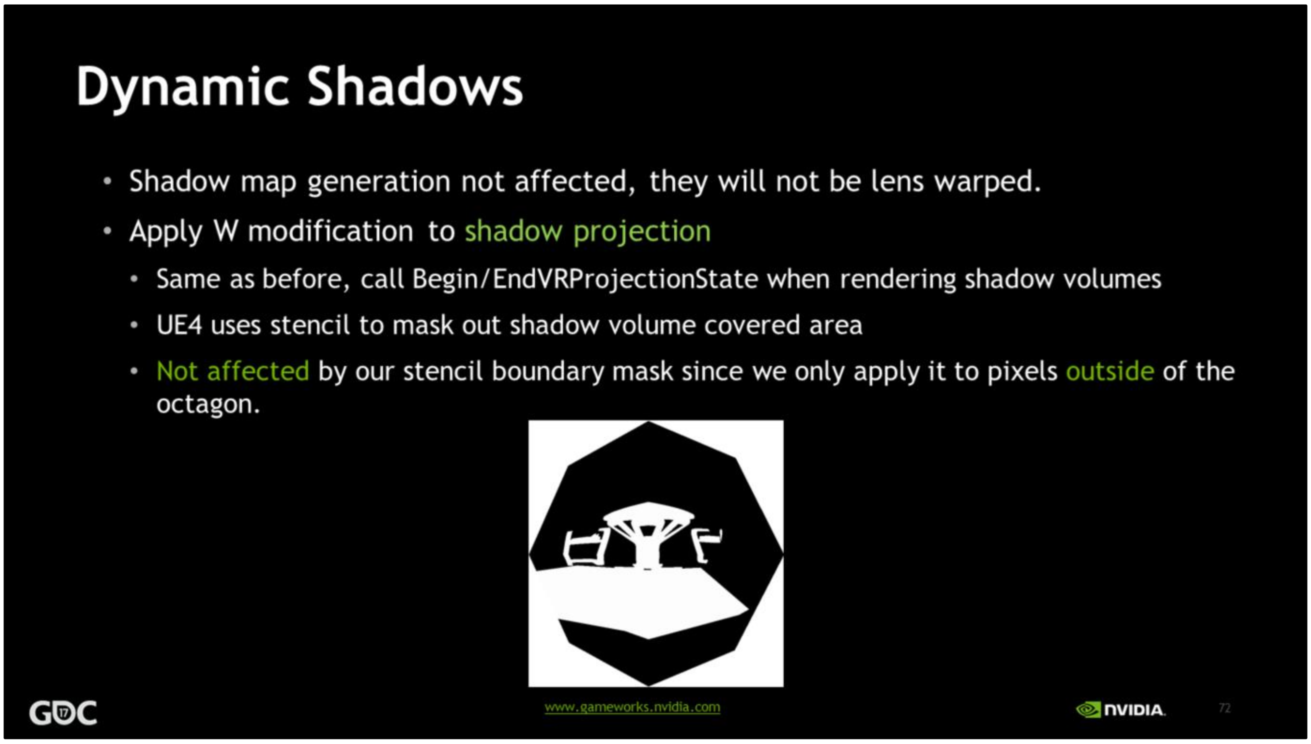
GDC www.gameworks.nvidia.com NVIDIA 71

For deferred lighting, directional light support is really simple, originally UE4 draws a fullscreen quad, so we just draw a full screen octagon as mentioned before.

For points and spot lights, since light volume geometry is rendered, we have to call Begin/EndVRProjectionStates around the draw call, and set up FastGS for them as well.
In the scenario where the camera is inside the light volume, Z test is disabled since all the pixels lit will be in front of the primitives drawn. In this case, our depth boundary mask won't work. However, lighting calculation is often one of the expensive pass in a frame, so we still have to cull. So we set the boundary mask in stencil buffer outside of the octagon region and use stencil test to kill redundant pixels instead.

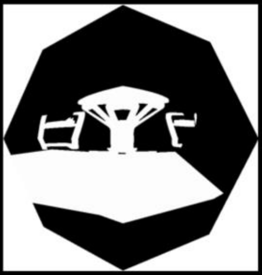
For shadows, shadow map generation is not affected since it will not go through lens warp. Only shadow projection should be modified to use w modification.

For tiled based lighting, as well as other CS passes like Environmental reflections, we also kill the thread group if all threads within that group are outside of the octagon covered region.



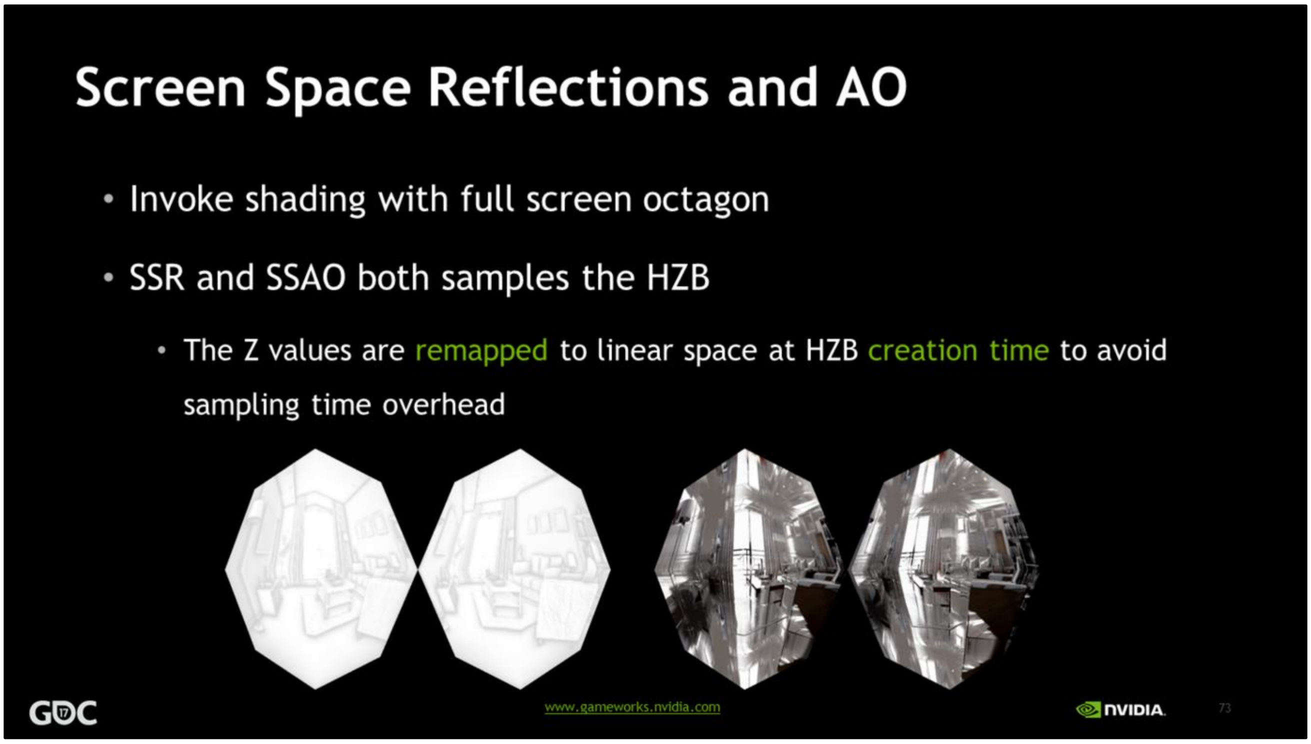
## Dynamic Shadows

- Shadow map generation not affected, they will not be lens warped.
- Apply W modification to shadow projection
  - Same as before, call Begin/EndVRProjectionState when rendering shadow volumes
  - UE4 uses stencil to mask out shadow volume covered area
  - Not affected by our stencil boundary mask since we only apply it to pixels outside of the octagon.



GDC 17 www.gameworks.nvidia.com NVIDIA 72

Next, for dynamic shadows. Shadow map generation is not affected at all since they will not go through lens warp.
Shadow projections should be modified to apply w modification to render in LMS space.
Any who's familiar with UE4's renderer would probably know that it also uses stencil buffer to mask pixels cover by shadow volumes. Since previously our stencil boundary mask was only applied to pixels outside of the octagon, it won't interfere with UE4's existing shadow volume stencil mask.

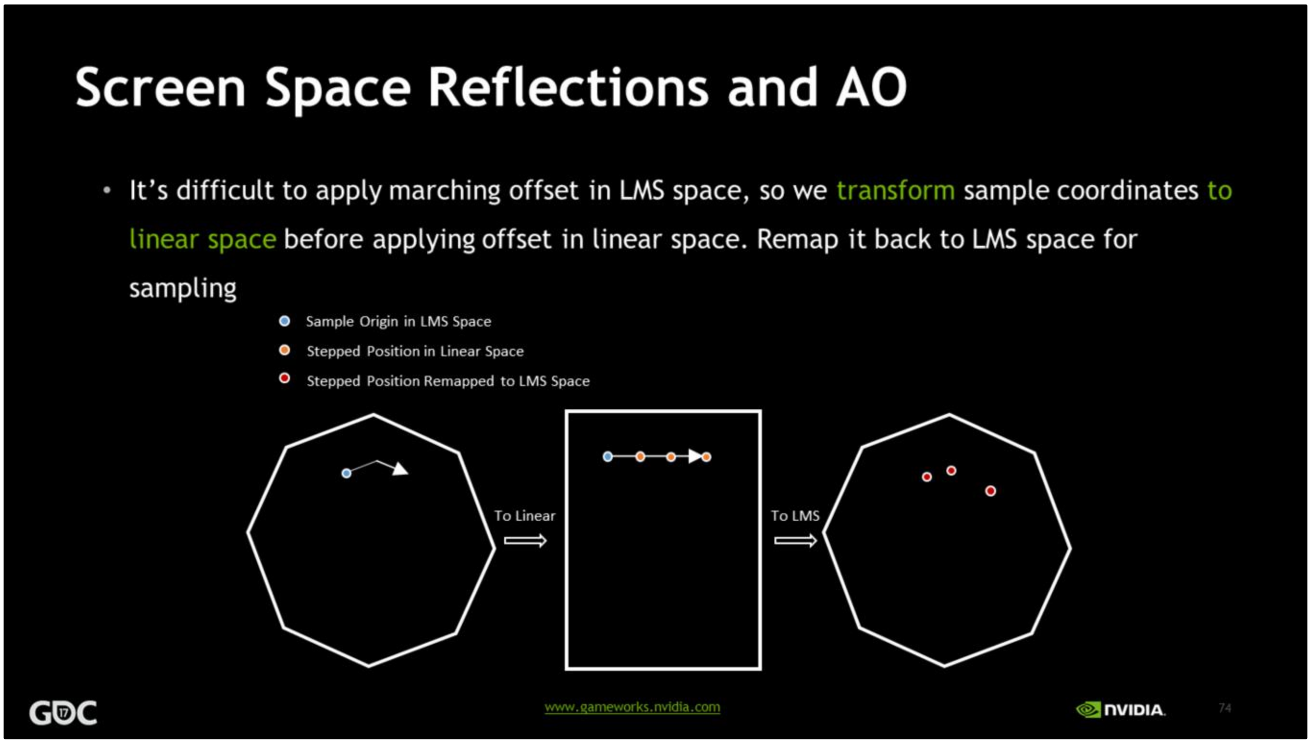


## Screen Space Reflections and AO

- Invoke shading with full screen octagon
- SSR and SSAO both samples the HZB
  - The Z values are remapped to linear space at HZB creation time to avoid sampling time overhead

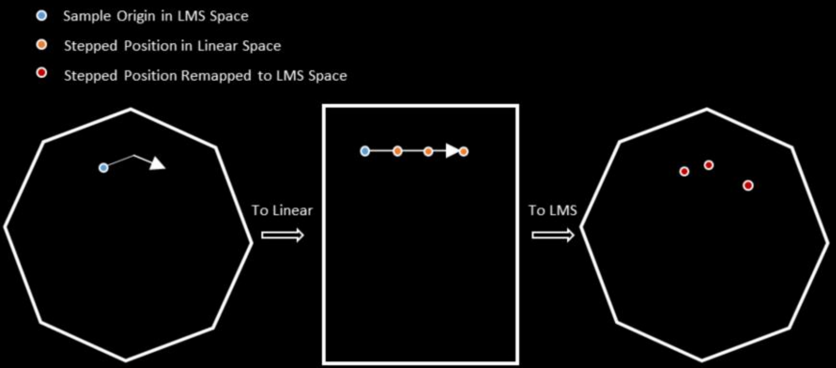
GDC www.gameworks.nvidia.com NVIDIA 73

For passes like SSR and SSAO, things become trickier.
First, they both samples the hierarchical Z buffer multiple times per pixel. And as mentioned previously, Z values needs to be remapped to linear space before used for computation. Doing this remapping multiple times per pixels is obviously bad for performance, so we remap the Z to linear space during HZB creation time to avoid redundant work during sampling.



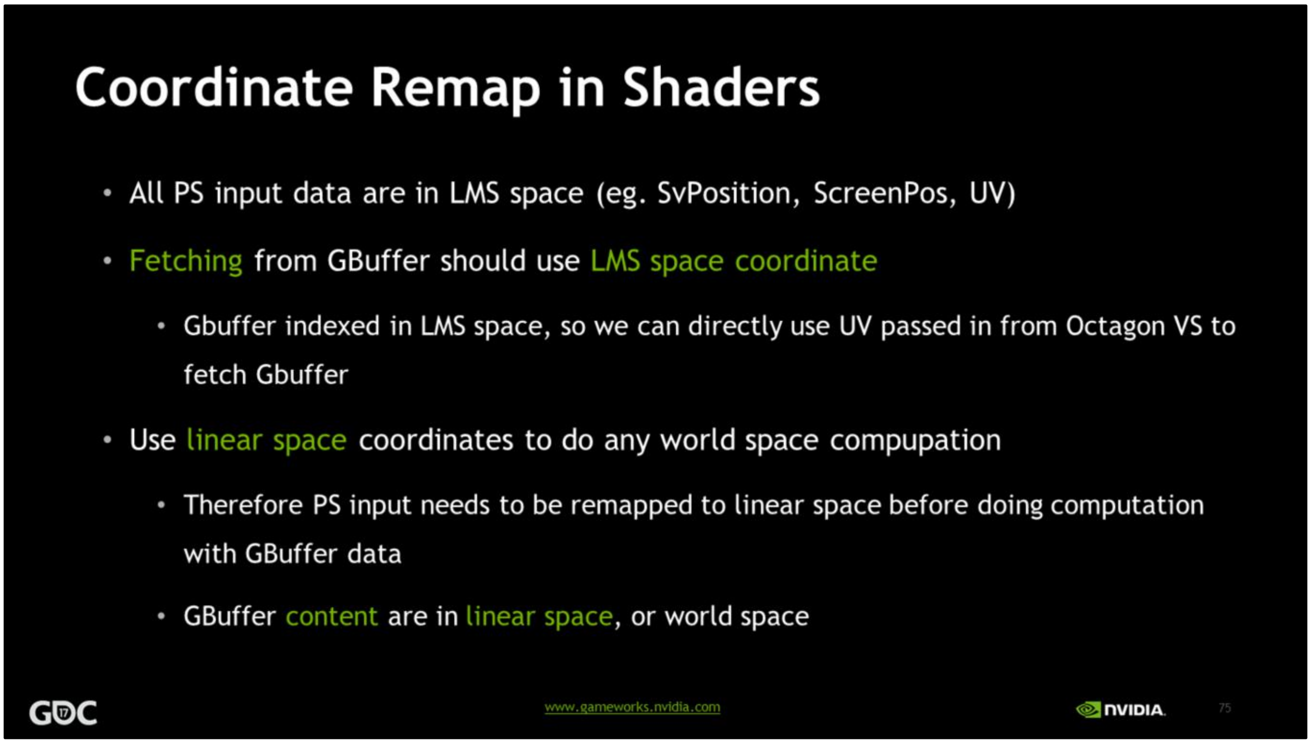
## Screen Space Reflections and AO

- It's difficult to apply marching offset in LMS space, so we transform sample coordinates to linear space before applying offset in linear space. Remap it back to LMS space for sampling



● Sample Origin in LMS Space
● Stepped Position in Linear Space
● Stepped Position Remapped to LMS Space
To Linear
To LMS

In addition, they both need to offset the UV for either ray marching or neighborhood sampling. But our UV are in LMS space, and the offset are in linear space. So we transform all the coordinates to linear space and apply the offset there, before remapping it back to LMS space to do sampling.



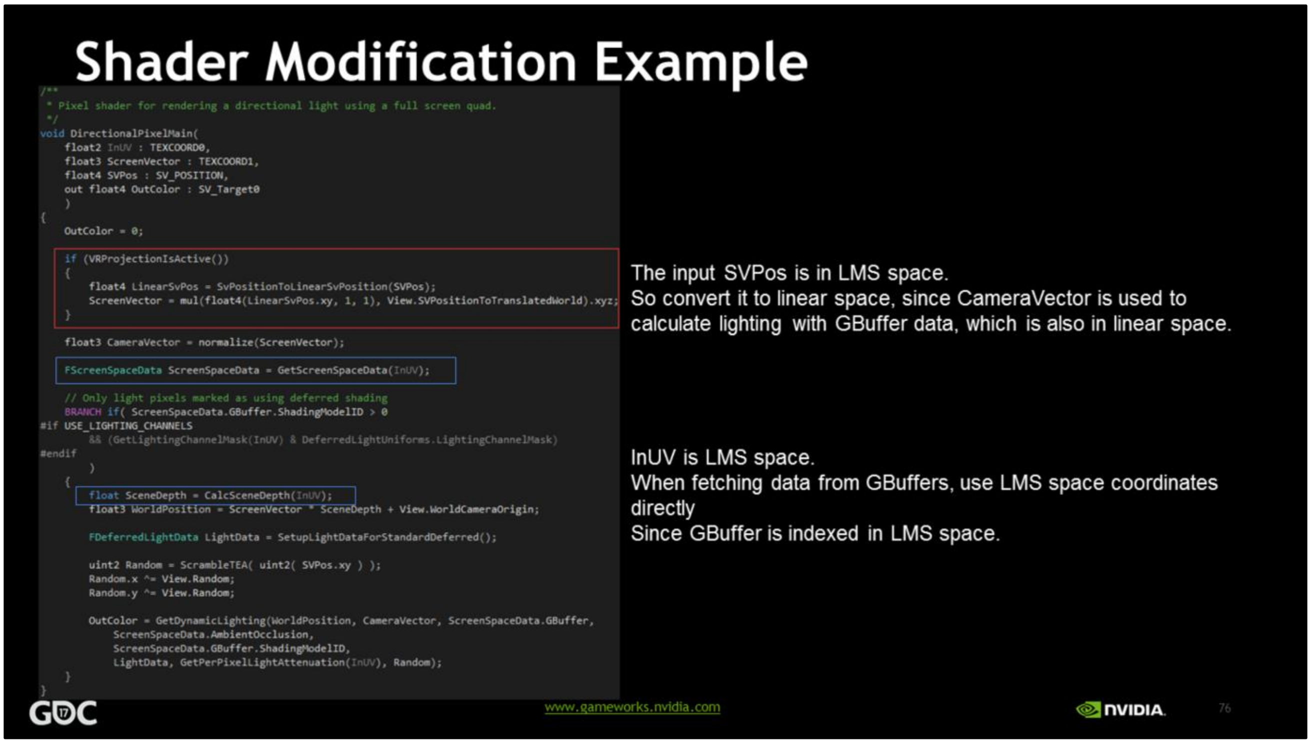
## Coordinate Remap in Shaders

- All PS input data are in LMS space (eg. SvPosition, ScreenPos, UV)
- Fetching from GBuffer should use LMS space coordinate
  - Gbuffer indexed in LMS space, so we can directly use UV passed in from Octagon VS to fetch Gbuffer
- Use linear space coordinates to do any world space compupation
  - Therefore PS input needs to be remapped to linear space before doing computation with GBuffer data
  - GBuffer content are in linear space, or world space

GDC www.gameworks.nvidia.com NVIDIA 75

And finally, for all the passes mentioned before, and other passes that are not mentioned, their shaders also needs to be adjusted to work with the additional LMS space we've introduced. This can be confusing at time, so I am summarizing the rules here.
First, we assume that all the input from vertex shaders are in LMS space. This includes any input SVPosition, Uvs and ScreenVectors, from either the fullscreen octagon or regular geometries.

When we need to fetch data from Gbuffer, or other intermediate buffers, we don't need to apply any transform and should just use those coordinates out of the box, since the all octagon shaped buffers are already indexed in LMS space.
However, the data in Gbuffers are still in the world space. So we need to make sure to use linear space coordinates when doing computation with data fetched from Gbuffer. This means that the PS input will need to be remapped.



## Shader Modification Example

```
/**
 * Pixel shader for rendering a directional light using a full screen quad.
 */
void DirectionalPixelMain(
    float2 InUV : TEXCOORD0,
    float3 ScreenVector : TEXCOORD1,
    float4 SVPos : SV_POSITION,
    out float4 OutColor : SV_Target0
    )
{
    OutColor = 0;

    if (VRProjectionIsActive())
    {
        float4 LinearSvPos = SvPositionToLinearSvPosition(SVPos);
        ScreenVector = mul(float4(LinearSvPos.xy, 1, 1), View.SVPositionToTranslatedWorld).xyz;
    }

    float3 CameraVector = normalize(ScreenVector);

    FScreenSpaceData ScreenSpaceData = GetScreenSpaceData(InUV);

    // Only light pixels marked as using deferred shading
    BRANCH if( ScreenSpaceData.GBuffer.ShadingModelID > 0
#if USE_LIGHTING_CHANNELS
        && (GetLightingChannelMask(InUV) & DeferredLightUniforms.LightingChannelMask)
#endif
        )
    {
        float SceneDepth = CalcSceneDepth(InUV);
        float3 WorldPosition = ScreenVector * SceneDepth + View.WorldCameraOrigin;

        FDeferredLightData LightData = SetupLightDataForStandardDeferred();

        uint2 Random = ScrambleTEA( uint2( SVPos.xy ) );
        Random.x ^= View.Random;
        Random.y ^= View.Random;

        OutColor = GetDynamicLighting(WorldPosition, CameraVector, ScreenSpaceData.GBuffer,
            ScreenSpaceData.AmbientOcclusion,
            ScreenSpaceData.GBuffer.ShadingModelID,
            LightData, GetPerPixelLightAttenuation(InUV), Random);
    }
}
```

The input SVPos is in LMS space.
So convert it to linear space, since CameraVector is used to calculate lighting with GBuffer data, which is also in linear space.

InUV is LMS space.
When fetching data from GBuffers, use LMS space coordinates directly
Since GBuffer is indexed in LMS space.

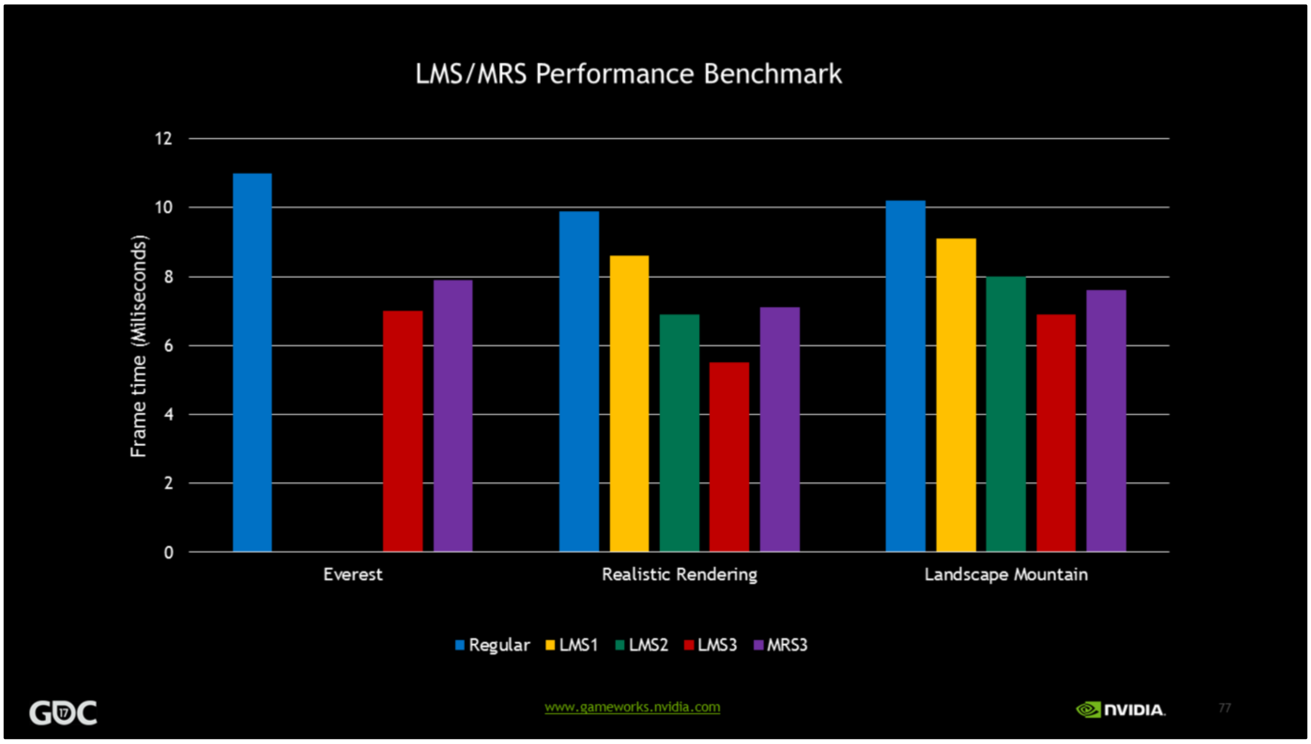
GDC 17 www.gameworks.nvidia.com NVIDIA 76

I know the previous slide could be confusing. So here is an concrete example.
This is the deferred directional light pixel shader. Some of the input like ScreenVector is used to calculate CameraVector, which will later be passed to GetDynamicLighting to compute lighting intensity. InUV on the other hand is used to fetch data from the Gbuffer.

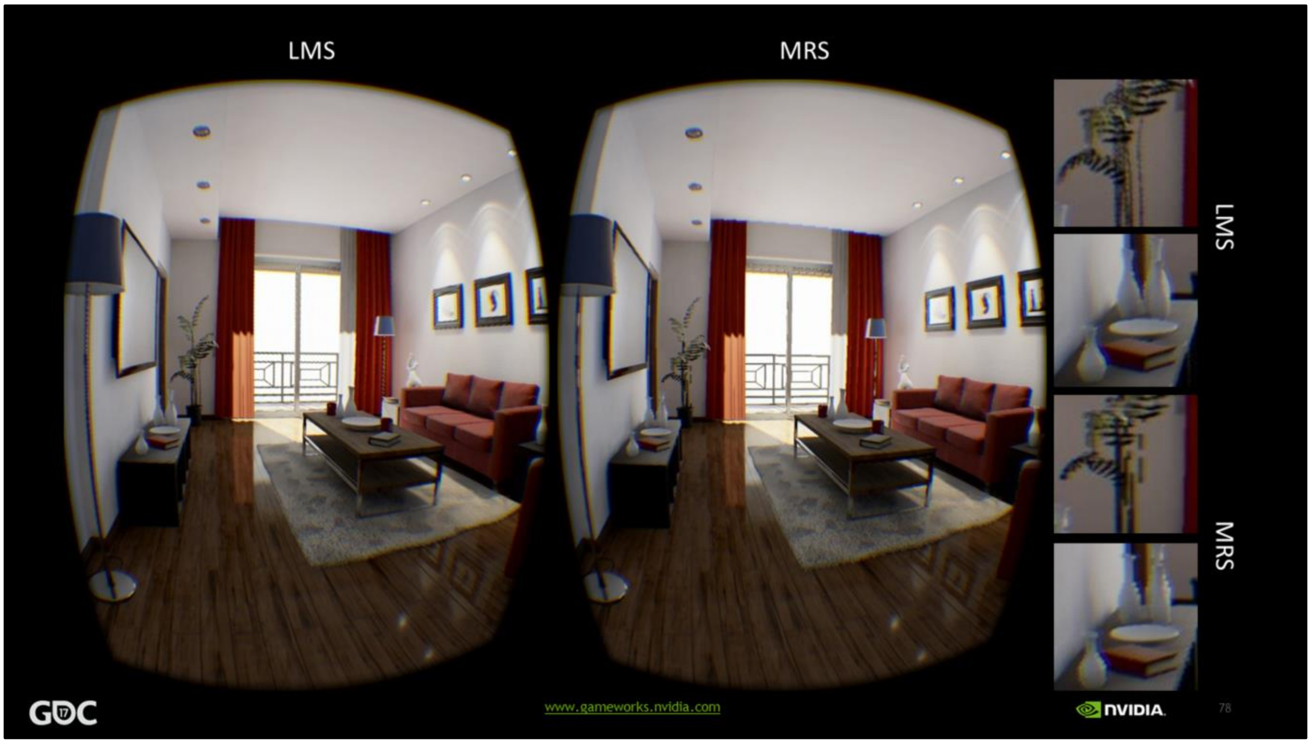
The inputs to this shader, InUV, ScreenVector and SVPos are all in LMS space as mentioned before.
So that’s why in the first if clause highlighted in red, we need to transform the SVPos and ScreenVector to linear space, before using it to calculate CameraVector. Since the lighting compupation requires data to be in linear world space.

On the other hand, we use the InUV as it is in LMS space to fetch from the Gbuffer.
The calls are high lighted in blue.

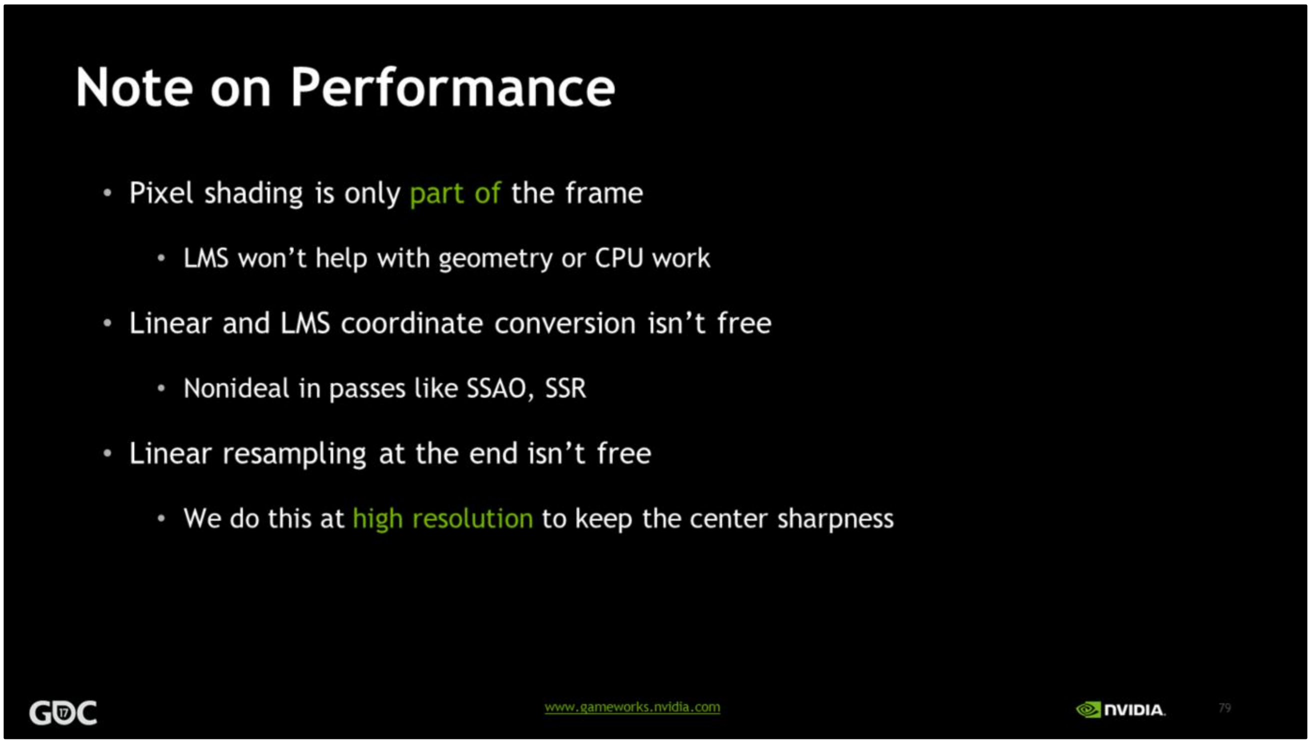
Hopefully this helps with making things a bit clearer for you. And for all other passes that weren’t explicitly mentioned in this talk, they follow the exact same principals introduced here.



Now let's look at some performance after using LMS and also compare them with the most aggressive MRS configurations.
Clearly LMS provides a bigger performance improvement compared with MRS. Even in a real game like Everest, developer actually reported that LMS provides both better performance and image quality.



Now for image quality, both LMS and MRS should provide little visual difference when viewed through an real HMD, In fact LMS should even resolve thin geometries better because of the fact that it actually super samples the central region.



## Note on Performance

- Pixel shading is only part of the frame
  - LMS won't help with geometry or CPU work
- Linear and LMS coordinate conversion isn't free
  - Nonideal in passes like SSAO, SSR
- Linear resampling at the end isn't free
  - We do this at high resolution to keep the center sharpness

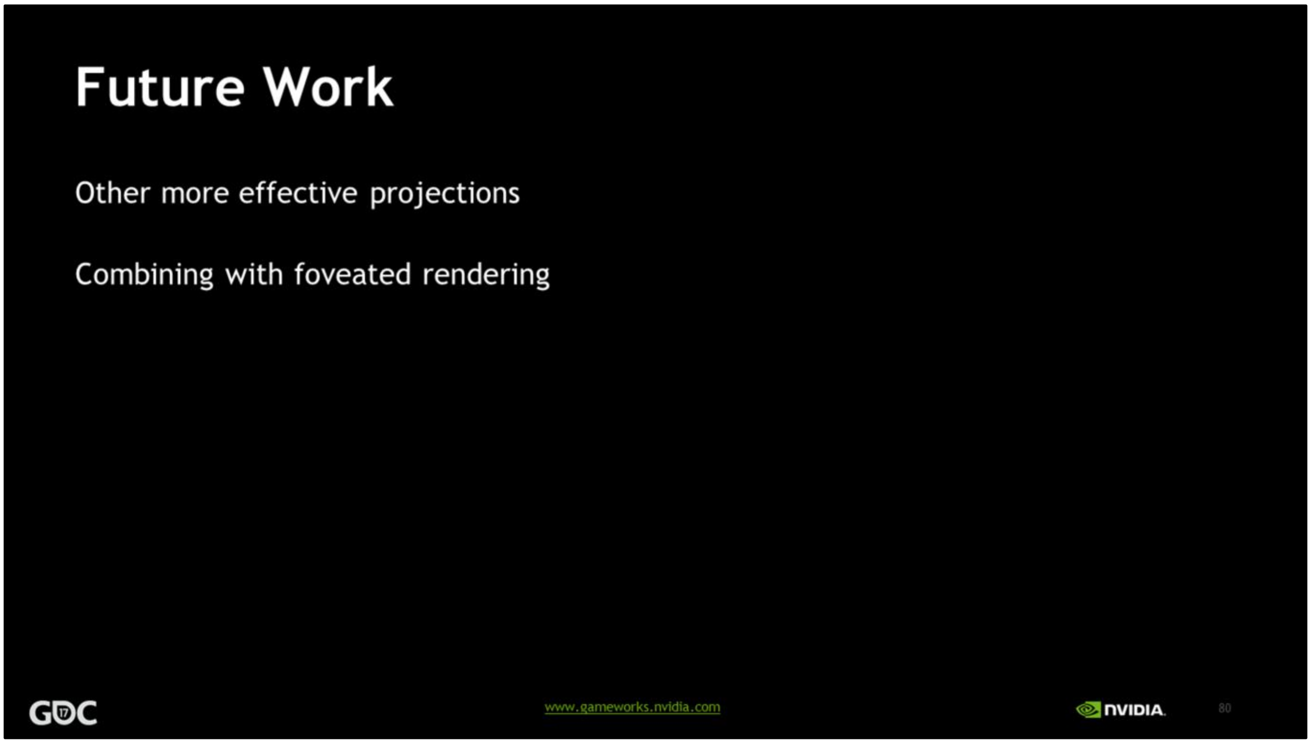
GDC 17 www.gameworks.nvidia.com NVIDIA 79

Finally,  couple things to note about performance. You might have noticed that reduced frame time isn't as much as the amount of pixels shaded. There are many reasons for this:

Pixel shading is only part of the frame. If your pipeline is geometry bound or even CPU bound, Lens Matched Shading isn't likely to help.

The coordinate remap between linear space and LMS space definitely isn't free, and we have to do it quite a few times in passes like SSR and SSAO.

The additional pass for resampling to linear space, especially when done at a much higher resolution in order to maintain center sharpness can impact performance too.



## Future Work

Other more effective projections

Combining with foveated rendering

## Demo

## NVIDIA Nsight VSE

www.gameworks.nvidia.com

82

## NSIGHT

### What is Nsight VSE?

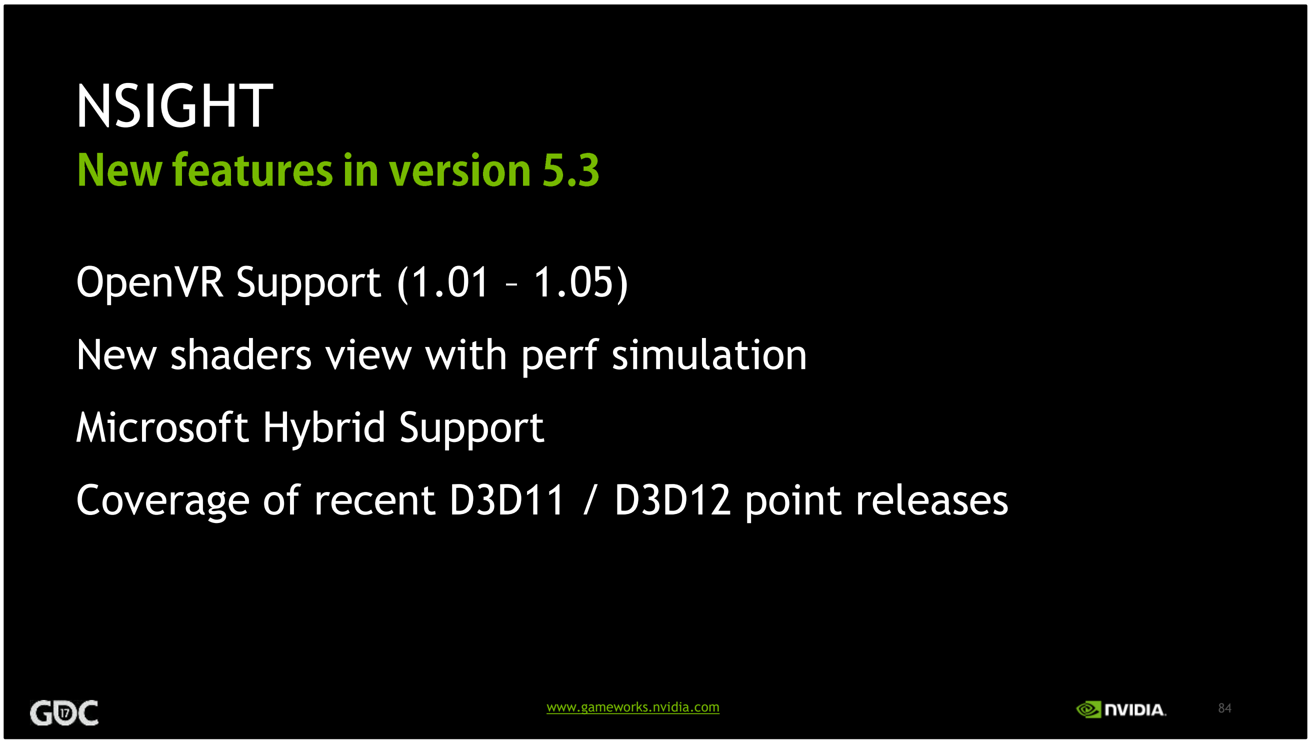
Understand CPU/GPU interaction

Explore and debug your frame as it is rendered

Profile your frame to understand hotspots and bottlenecks

Save your frame for targeted analysis and experimentation

Leverage the Microsoft Visual Studio platform



## NSIGHT

### New features in version 5.3

OpenVR Support (1.01 – 1.05)

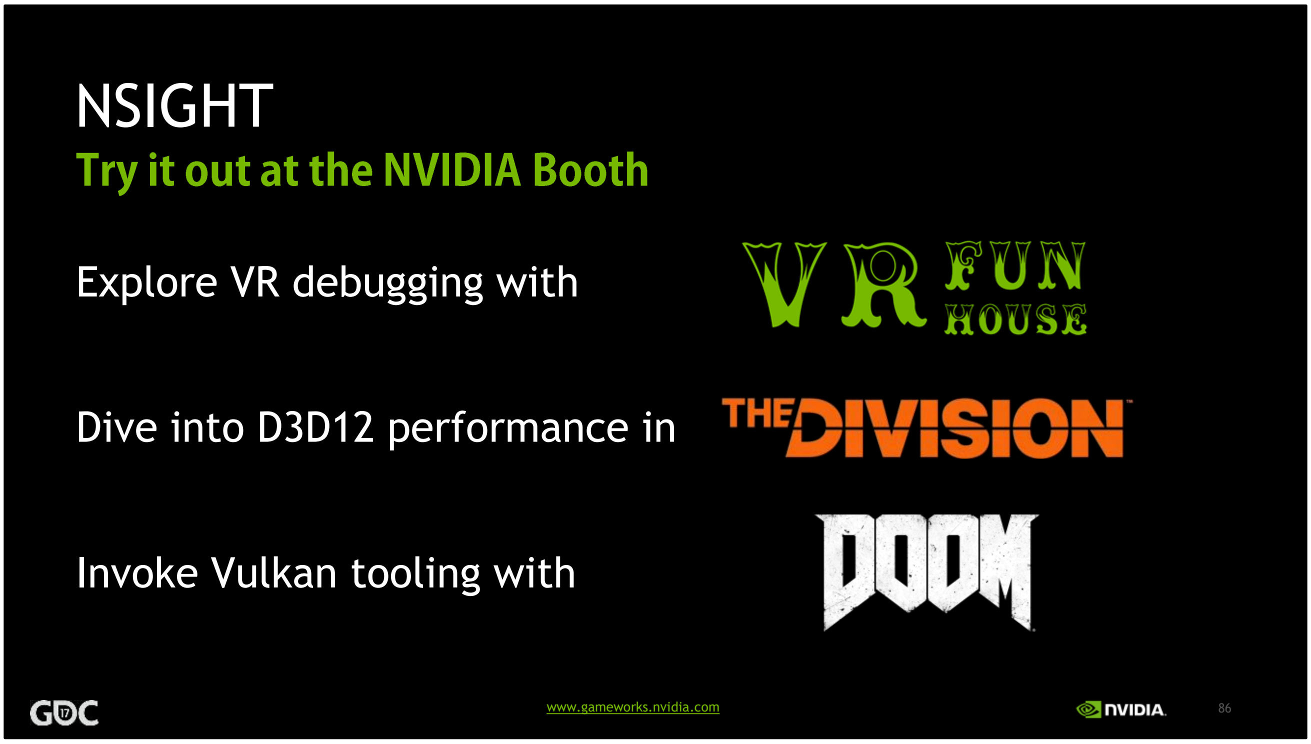
New shaders view with perf simulation

Microsoft Hybrid Support

Coverage of recent D3D11 / D3D12 point releases

## NSIGHT
### Demo

www.gameworks.nvidia.com



## NSIGHT

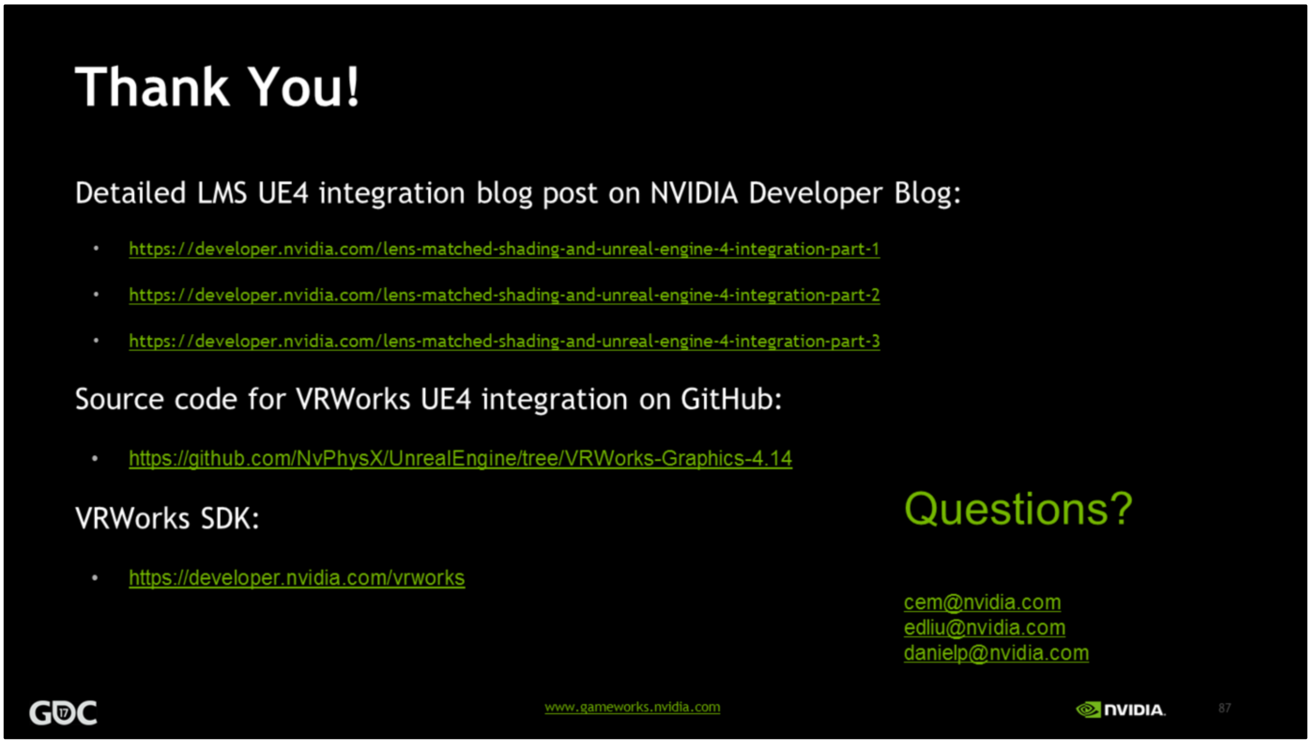
**Try it out at the NVIDIA Booth**

Explore VR debugging with VR FUN HOUSE

Dive into D3D12 performance in THE DIVISION

Invoke Vulkan tooling with DOOM

GDC 17 www.gameworks.nvidia.com NVIDIA 86



## Thank You!

Detailed LMS UE4 integration blog post on NVIDIA Developer Blog:

- https://developer.nvidia.com/lens-matched-shading-and-unreal-engine-4-integration-part-1
- https://developer.nvidia.com/lens-matched-shading-and-unreal-engine-4-integration-part-2
- https://developer.nvidia.com/lens-matched-shading-and-unreal-engine-4-integration-part-3

Source code for VRWorks UE4 integration on GitHub:

- https://github.com/NvPhysX/UnrealEngine/tree/VRWorks-Graphics-4.14

VRWorks SDK:

- https://developer.nvidia.com/vrworks

Questions?

cem@nvidia.com
edliu@nvidia.com
danielp@nvidia.com