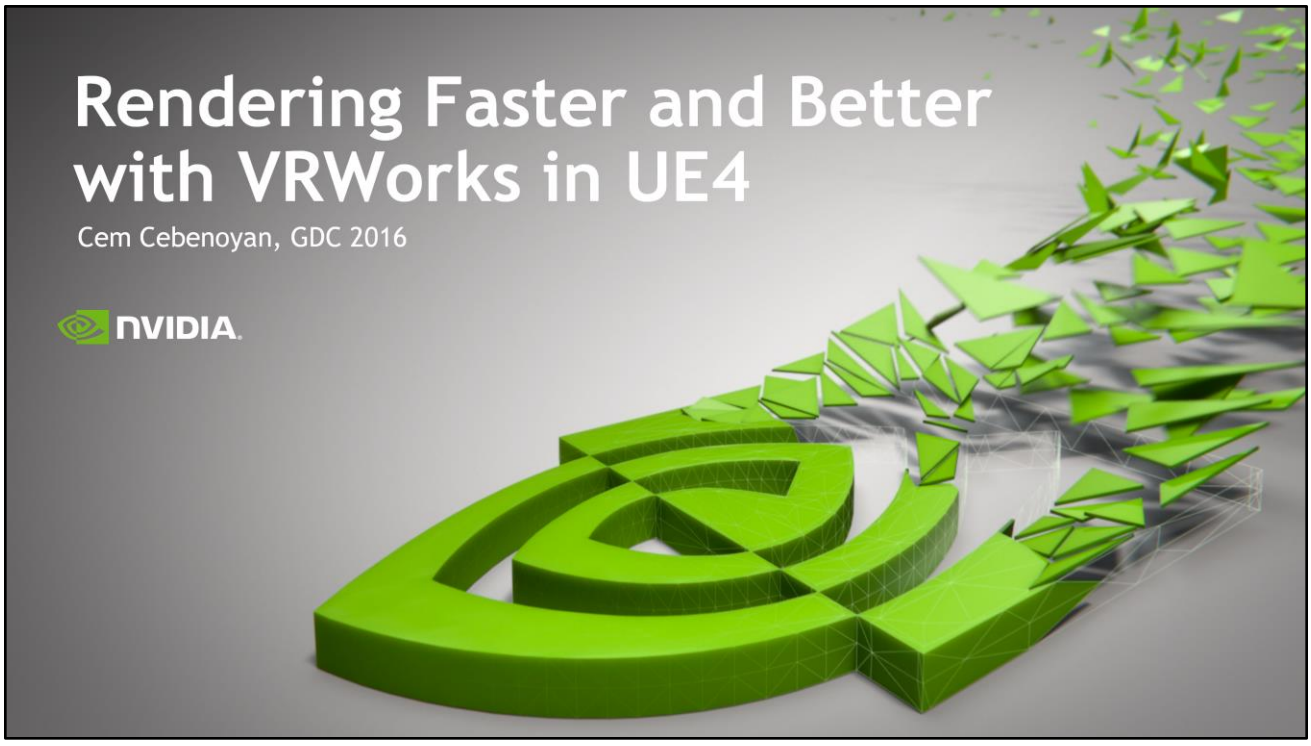

Rendering Faster and Better with VRWorks in UE4
Cem Cebenoyan, GDC 2016
NVIDIA

# Talk Overview

VRWorks Features

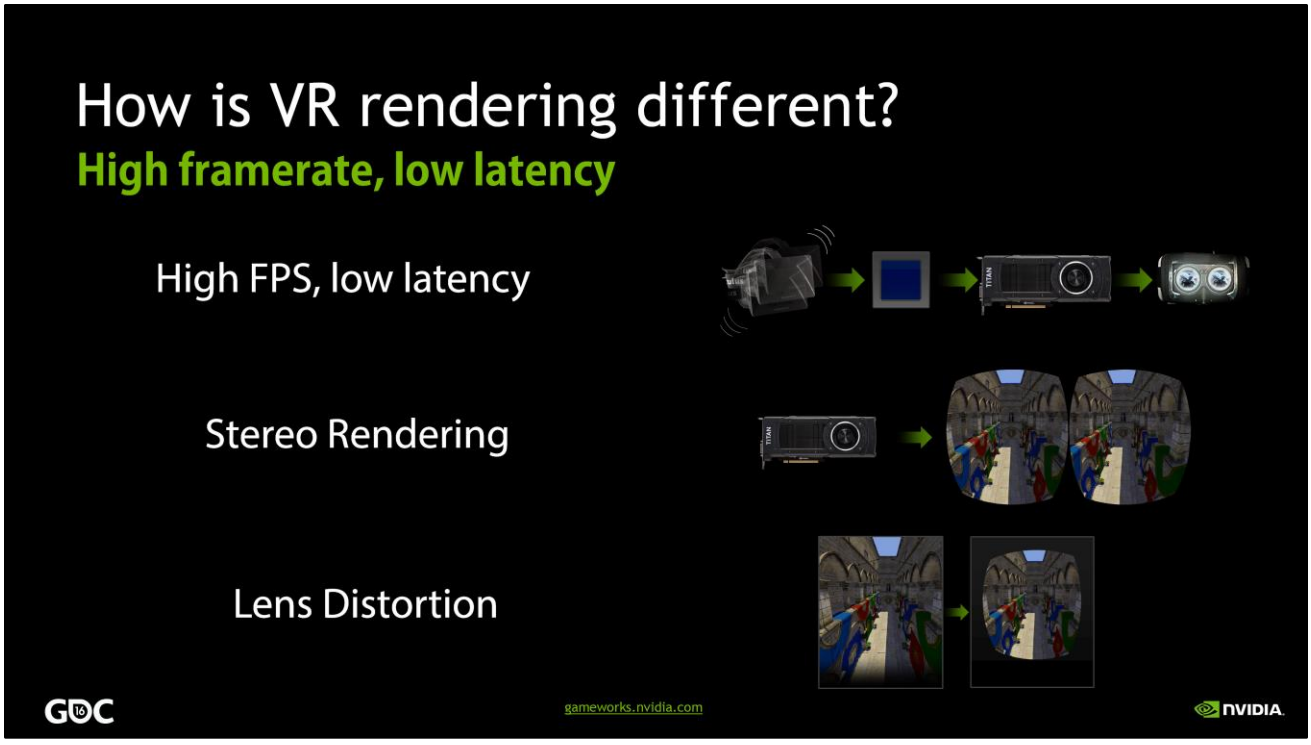
- Multi-Res Shading
- VR SLI

UnrealEngine 4 Integration

# How is VR rendering different?



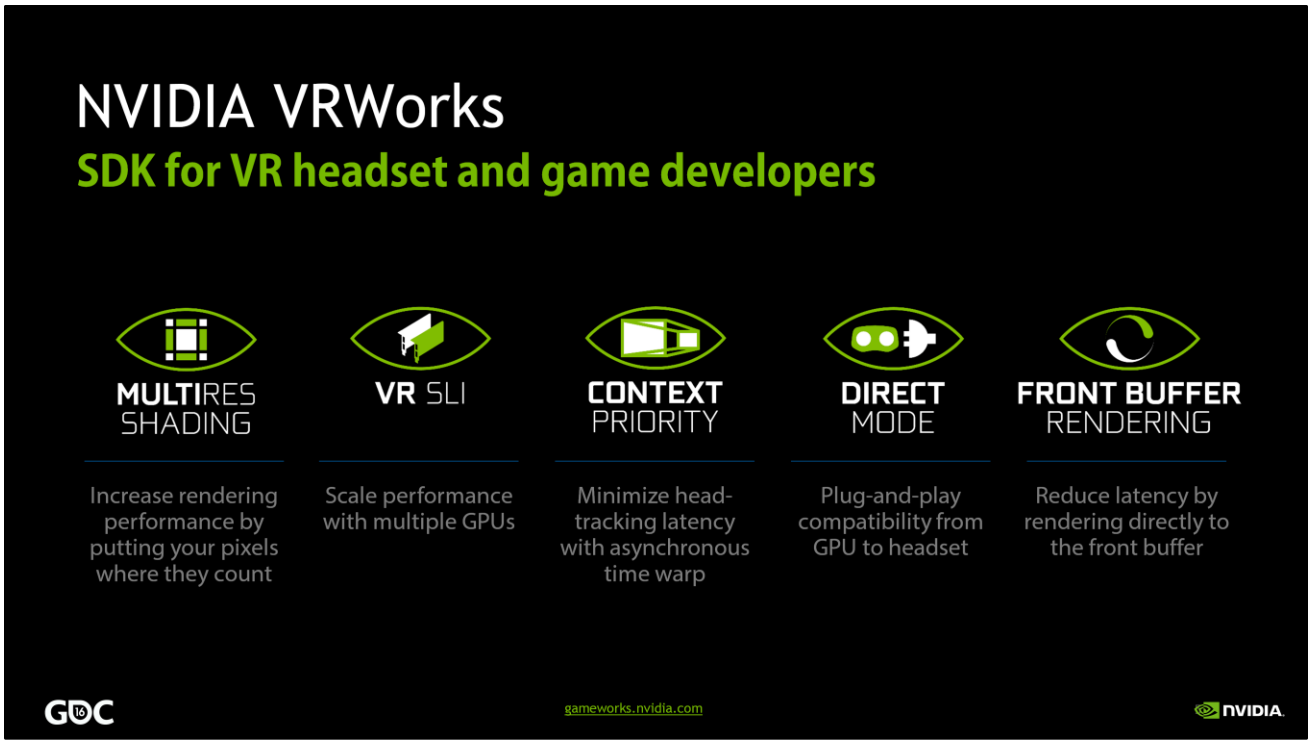
To set the stage, first I want to mention a few ways that virtual reality rendering differs from the more familiar kind of GPU rendering that real-time 3D apps and games have been doing up to now.




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.

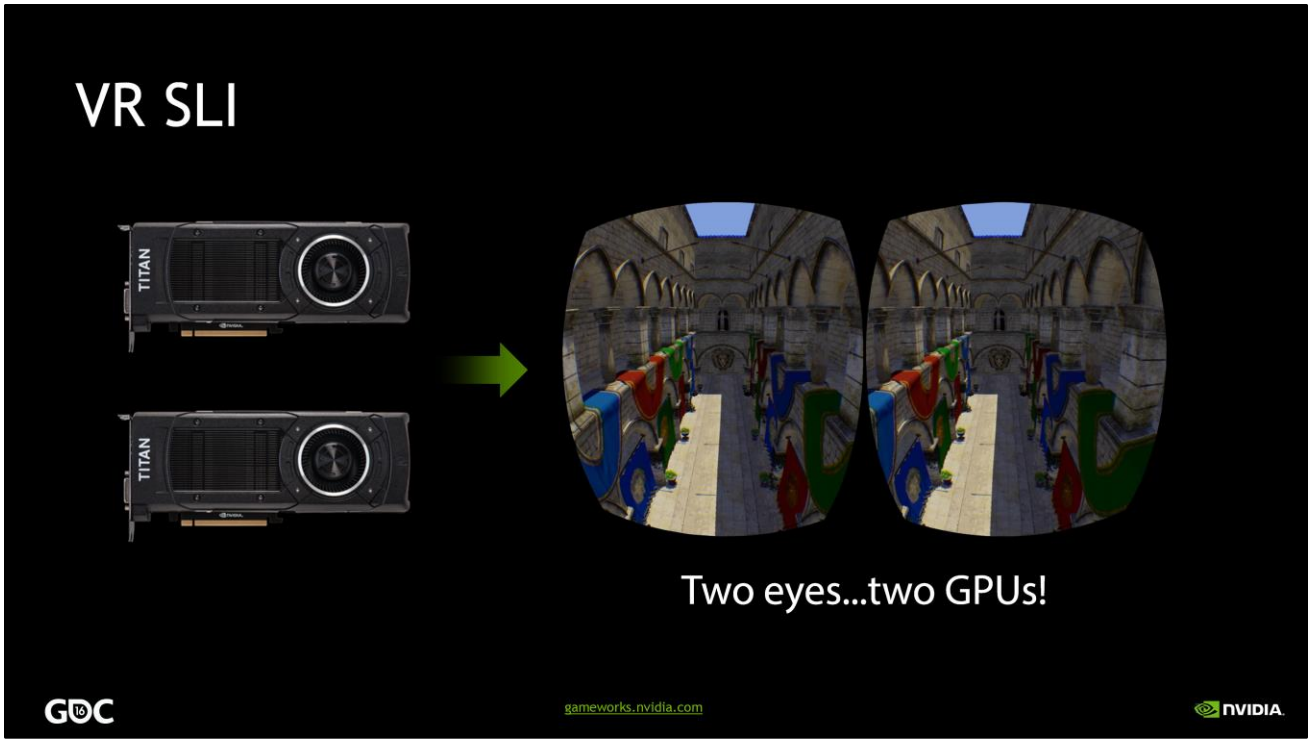
Traditional real-time rendering pipelines have not been optimized to minimize latency, so this goal requires us to change our mindset a little bit.




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—VRWorks. VRWorks 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.

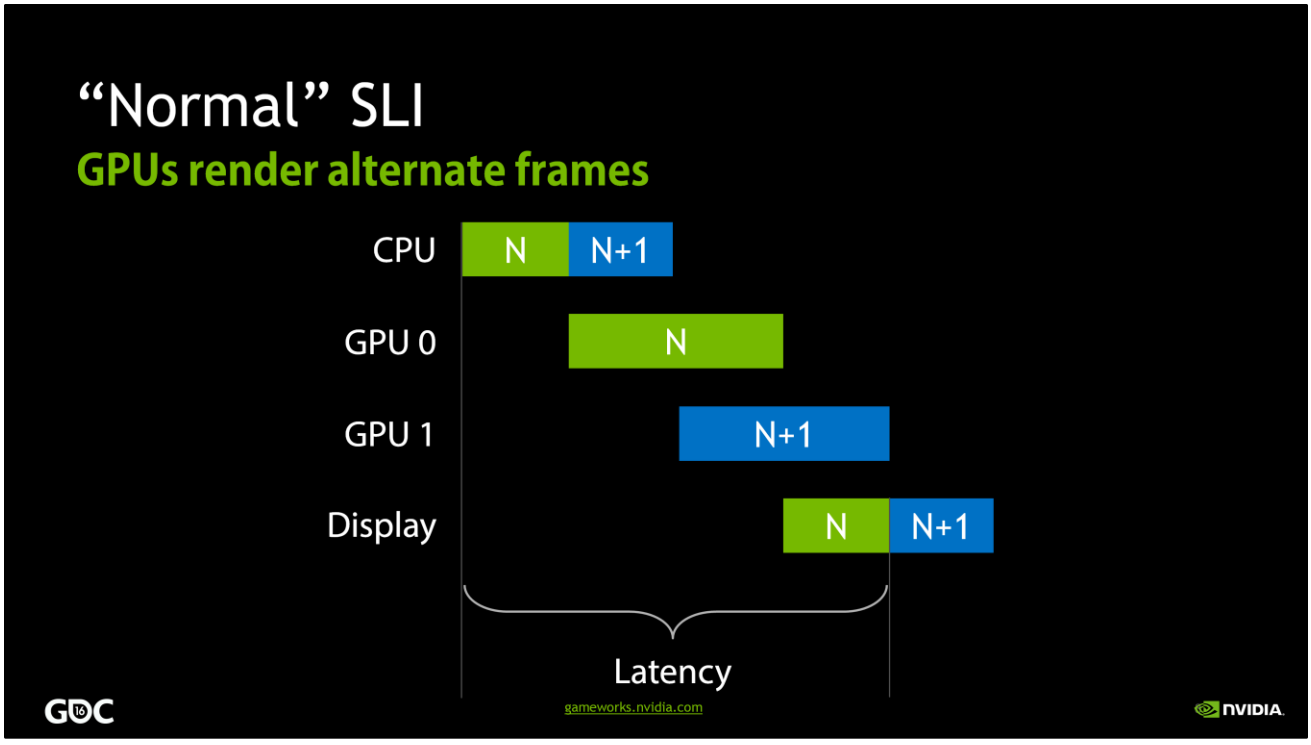
It has several different components, which we'll go through in this talk. The first two features, multi-res shading and VR SLI, are targeted more at game and engine developers. The last three are more low-level features, intended for VR headset developers to use in their software stack.

# VR SLI



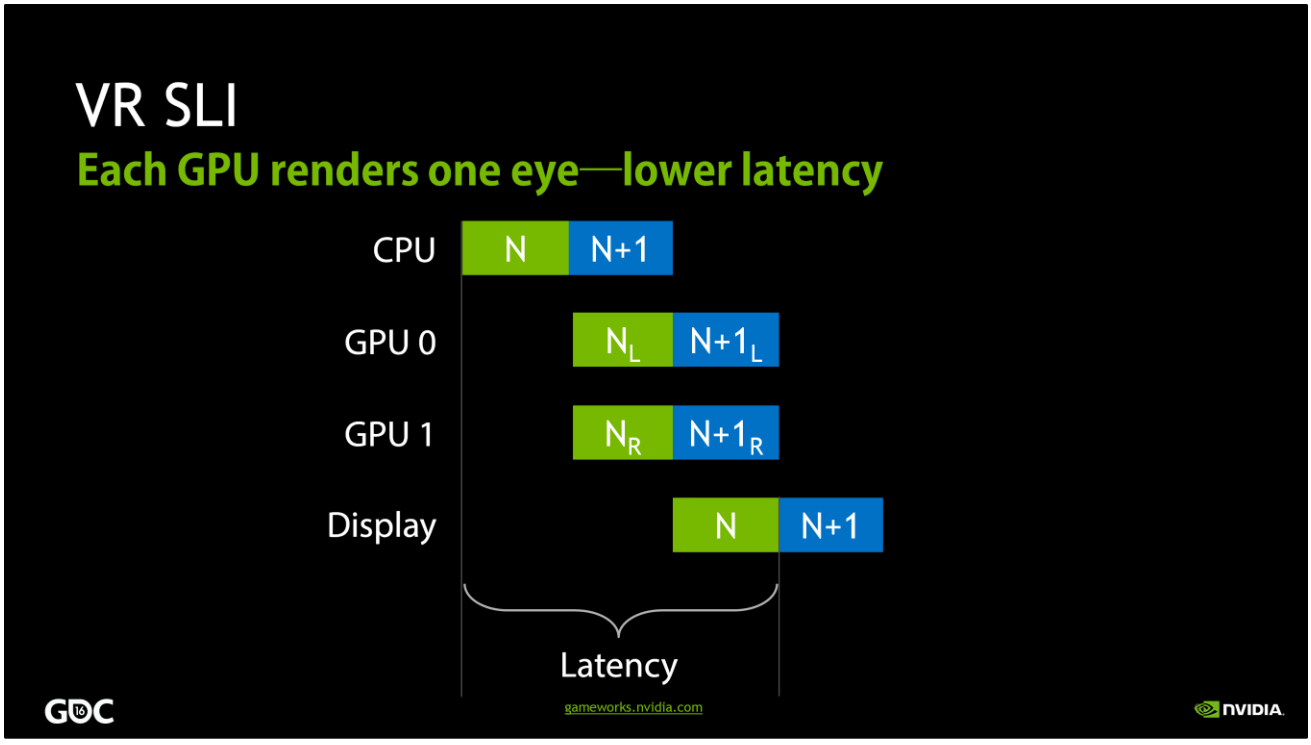

Given that the two stereo views are independent of each other, it's intuitively obvious that you can parallelize the rendering of them across two GPUs to get a massive improvement in performance.

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.

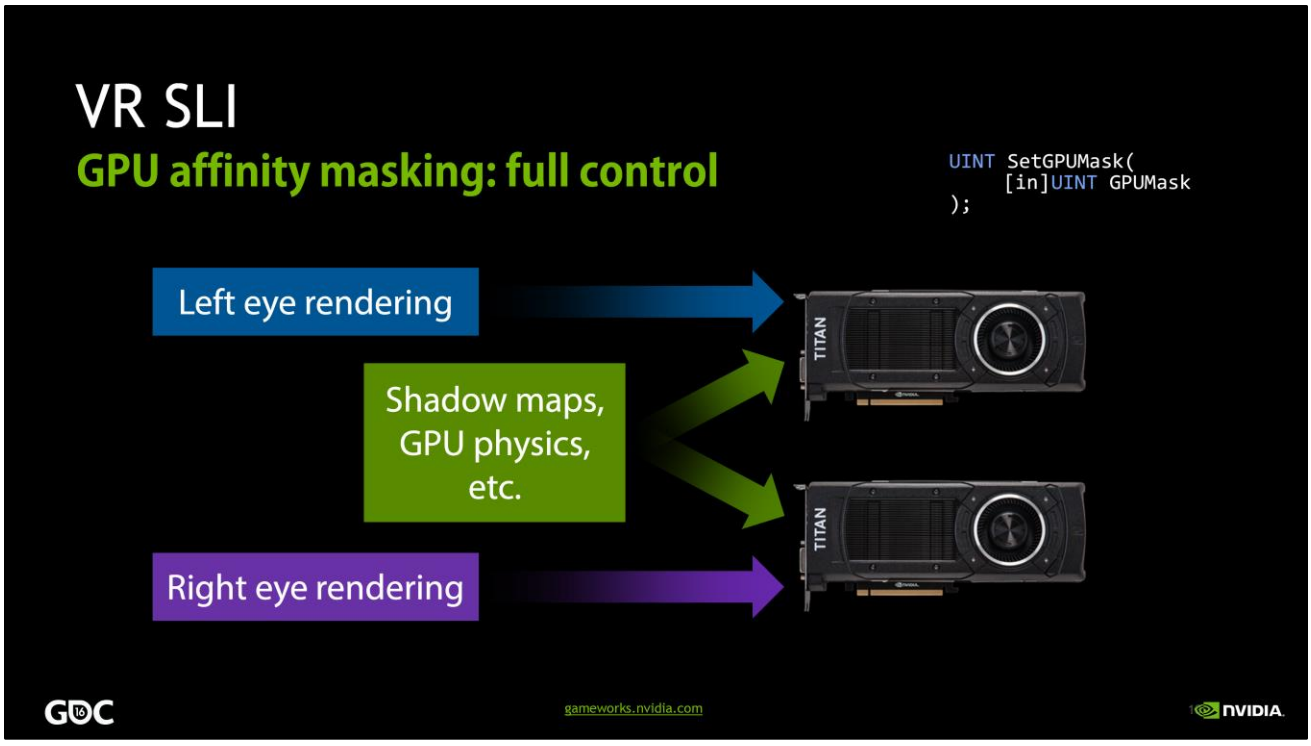



Before we dig into VR SLI, as a quick interlude, let me first explain how "normal", non-VR SLI works. For years, we've had alternate-frame SLI, in which the GPUs trade off frames. In the case of two GPUs, one renders the even frames and the other the odd frames. The GPU start times are staggered half 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.



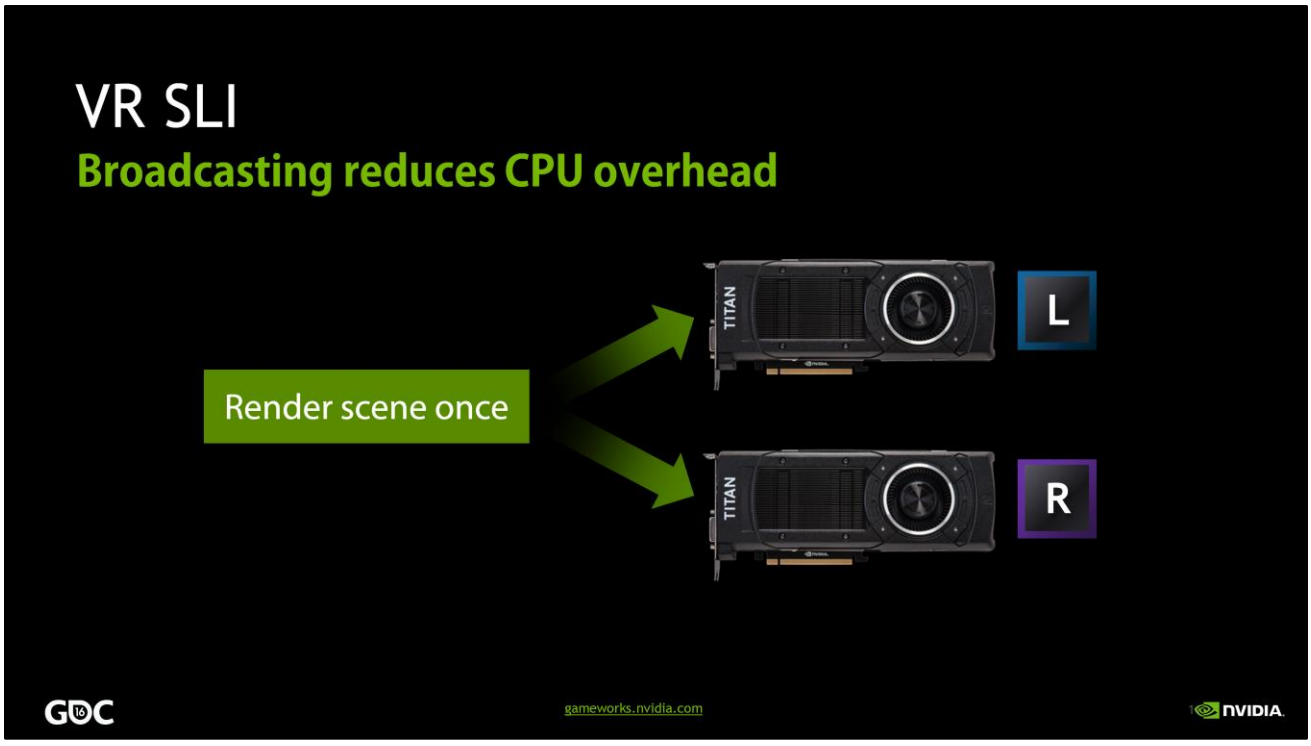

A better way to use two GPUs for VR rendering is to split the work of drawing a single frame across them—namely, by rendering each eye on one GPU. This has the nice property that it improves both framerate *and* latency relative to a single-GPU system.




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.

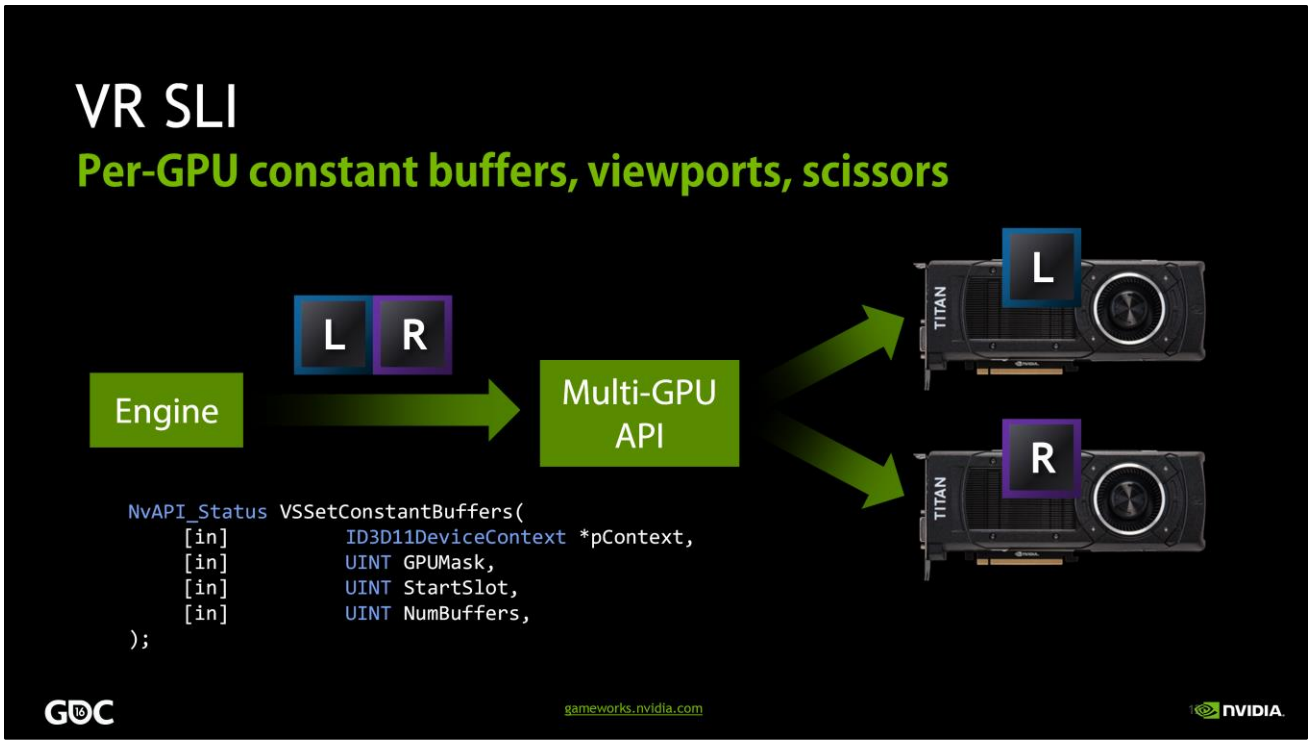
By the way, all of this extends to as many GPUs as you have in your machine, not just two. So you can use affinity masking to explicitly control how work gets divided across 4 or 8 GPUs, as well.




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.



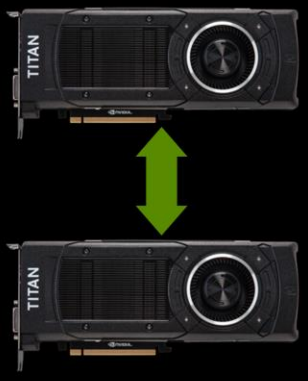

The only difference between the eyes is their view position—just a few numbers in a constant buffer. So, VR SLI lets you send different constant buffers to each GPU, so that each eye view is rendered from its correct position when the draw calls are broadcast.

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
);
```

GDC gameworks.nvidia.com NVIDIA

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.

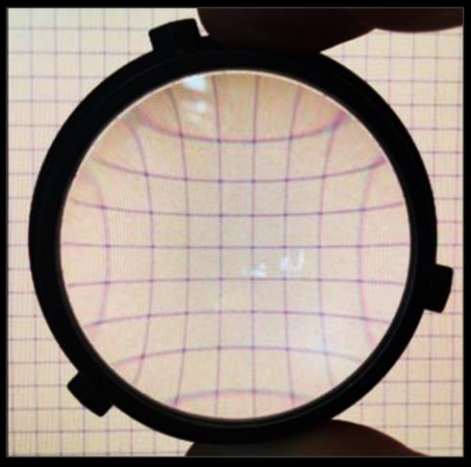
One point worth noting here is PCI Express bus bandwidth. PCIe2.0 x16 gives you 8 GB/sec of bandwidth, which isn't a huge amount, and it means that transferring an eye view will require about a millisecond. That's a significant fraction of your frame time at 90 Hz, so that's something to keep in mind.

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

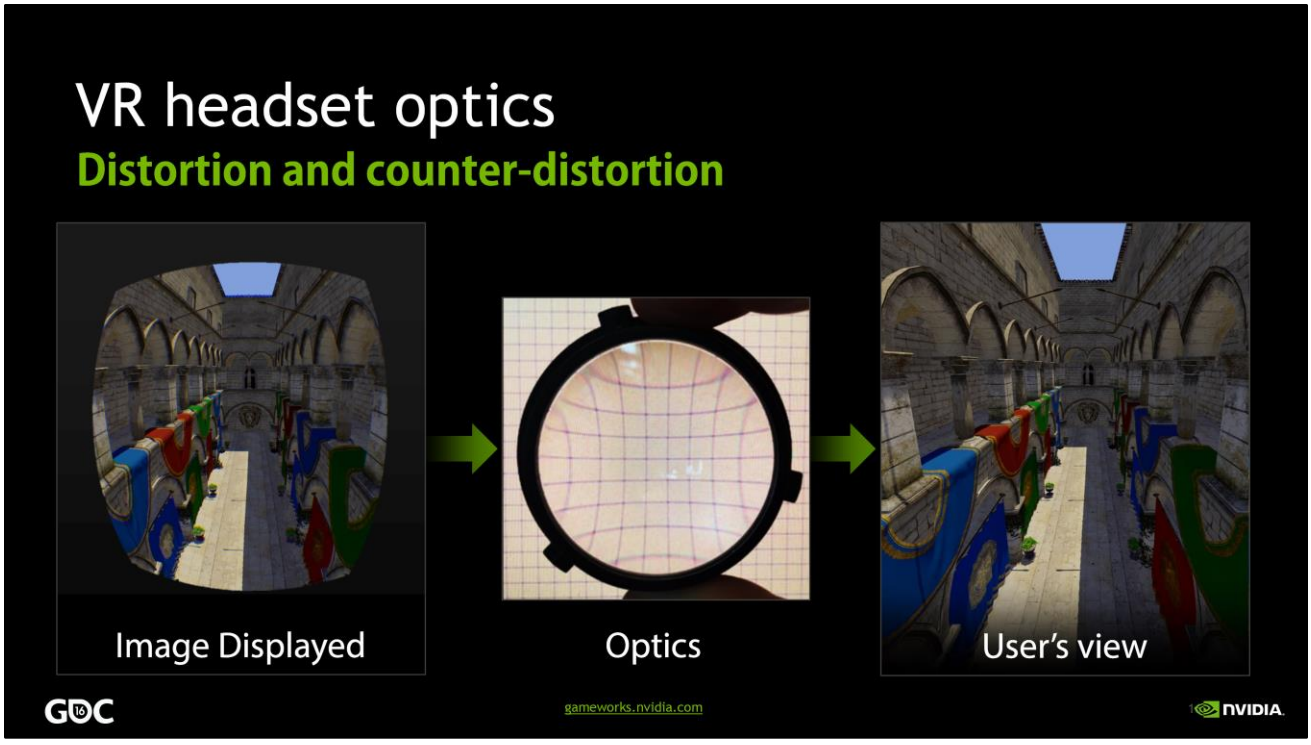
## Distortion and counter-distortion





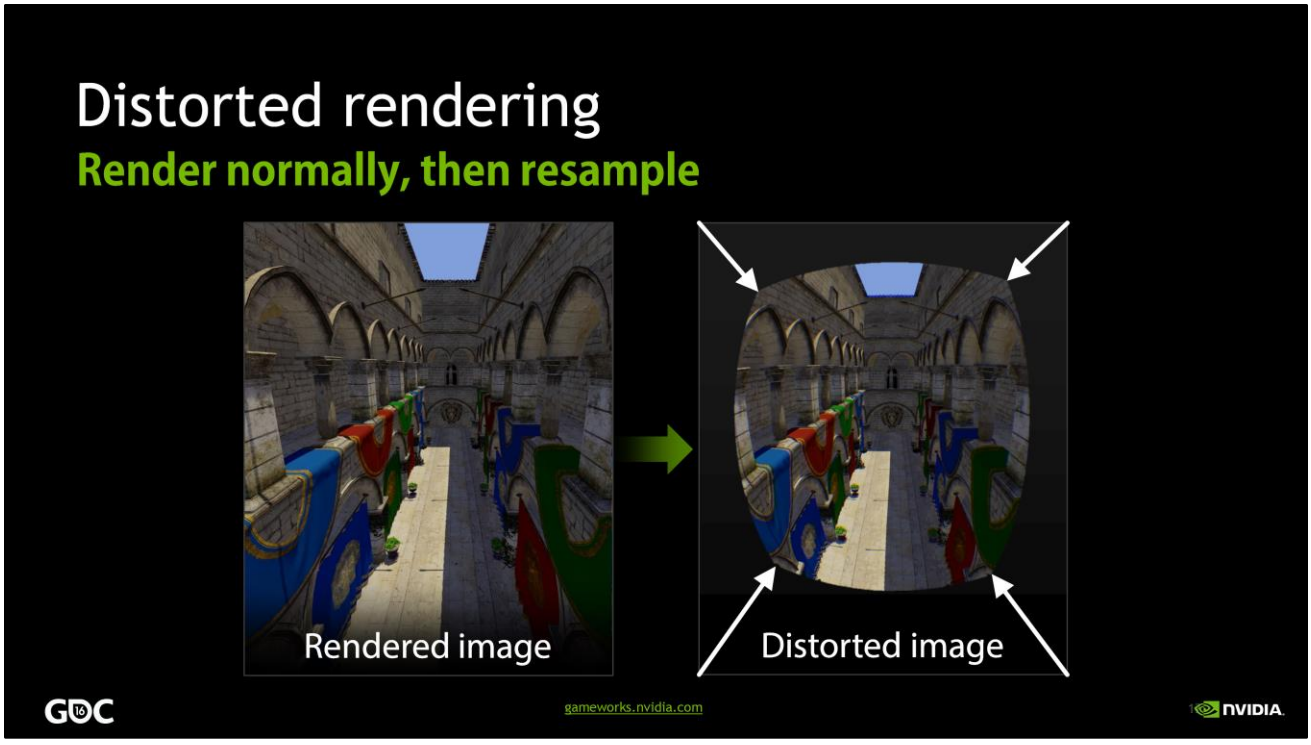
First, the basic facts about how the optics in a VR headset work.

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.




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.

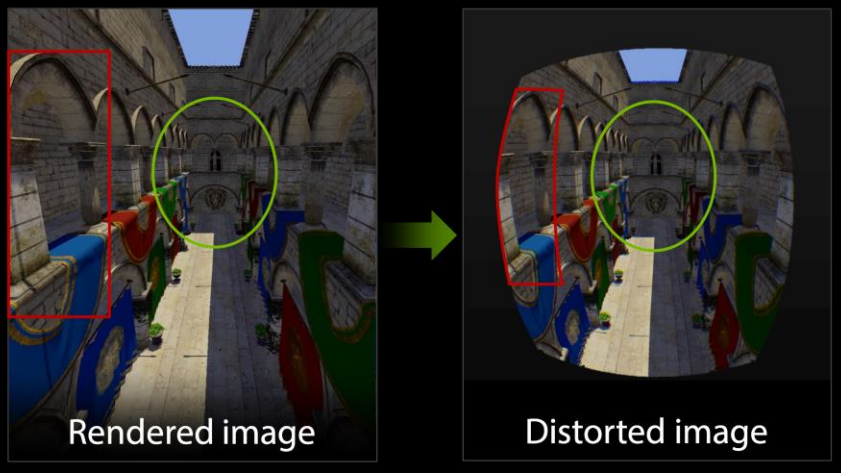



The trouble is that GPUs can't natively render into a nonlinearly distorted view like this—their rasterization hardware is designed around the assumption of linear perspective projections. Current VR software solves this problem by first rendering a normal perspective projection (left), then resampling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





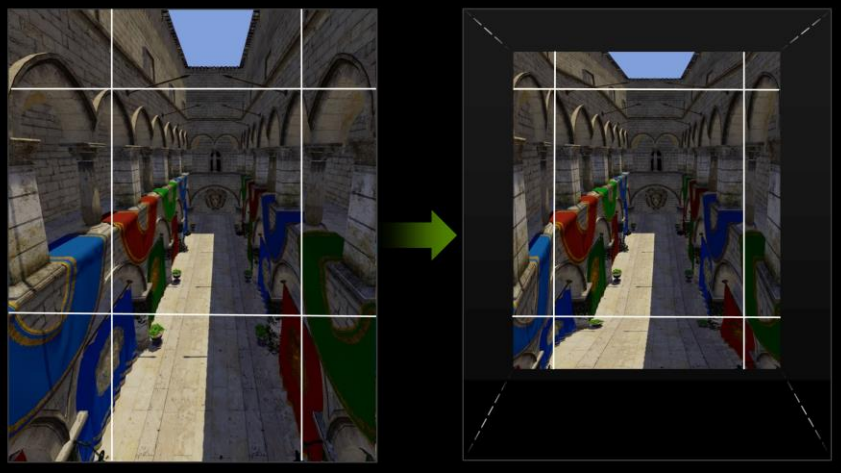

The reason for this is that if you look at what happens during the distortion pass, you find that while the center of the image stays the same, the outskirts are getting squashed quite a bit.

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

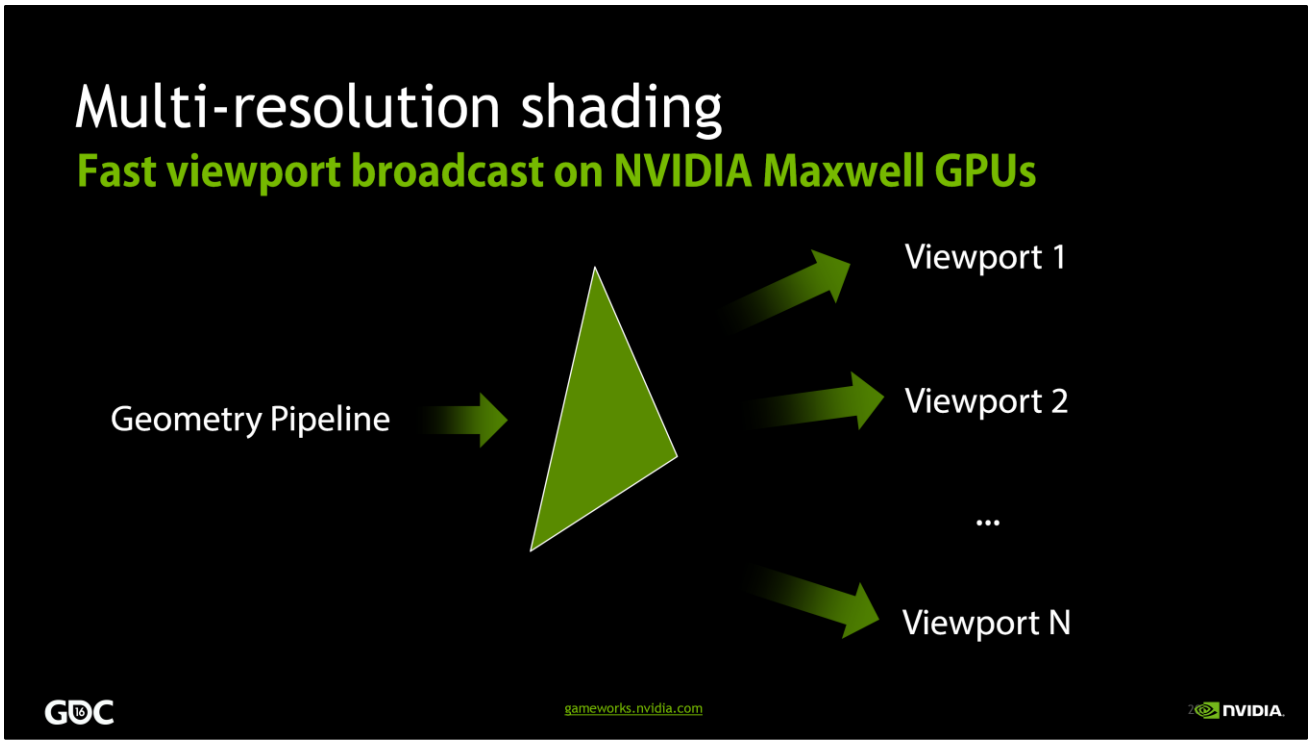




That brings us to multi-resolution shading. The idea is to subdivide the image into a set of adjoining viewports—here, a 3x3 grid of them. We keep the center viewport the same size, but scale down all the ones around the outside. All the left, right, top and bottom edges are scaled in, effectively reducing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




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

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, 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.

# Multi-resolution shading

## Public, open-source SDK

https://developer.nvidia.com/vrworks

NVAPI for Maxwell viewport broadcast in DX11/12

Helper code for configuring viewports, shader math, etc

DX11 sample app

Guide to integrating in an existing engine



Our SDK for multi-res shading is available on our website – that's the link there. It's an open-source SDK that provides two main things: a set of NVAPIs that allow developers to access the Maxwell viewport broadcast features, plus some open-source helper code for configuring the multi-res viewport layout, mapping back and forth between multi-res and linear UV spaces, and such.

There's also a DX11 sample app, and a programming guide that explains everything in detail, including notes on how to integrate this technique in an existing rendering engine.

# Multi-Res in UnrealEngine 4

# Multi-resolution shading

## Unreal Engine integration

We've integrated multi-res in UE 4.10

Currently limited support for post effects with multi-res

Available on GitHub

https://github.com/NvPhysX/UnrealEngine/tree/MultiRes-4.10

UE 4.11 integration in progress



We're also working on integrating multi-res into Unreal Engine. As of now, we have a trial integration in UE 4.10, which is available on Github if you have access to UE4 there. It currently supports only a limited set of post effects, but the ones most commonly used for VR are all there. Meanwhile, we're also making progress on an integration into UE 4.11.

# UE4 integration

**https://github.com/NvPhysX/UnrealEngine/tree/MultiRes-4.10**

Postprocessing passes supported:

- Bloom & lens flare
- Tonemapping
- Temporal AA
- Reflection environment (not screen-space reflection)
- Height fog
- Decals
- Distortion/refraction



Only certain post-passes have had multi-res support added, since it takes a bit of attention to each one to make it work

# Multi-resolution shading
## Performance

UE4 Infiltrator demo: +30% to +40% FPS

- @ approximate VR render res

Best when pixel-bound

- We've seen ~50% perf boosts

# Console variables

r.MultiRes: 0 or 1 toggles multi-res rendering on or off

r.MultiRes.SplitsInset: [0, 0.5]

Splits' distance from edge of the screen, as fraction of screen size

r.MultiRes.DensityScale: [0, 1]

Scale factor for resolution of outer viewports

# UE4 integration

## Main points

Direct engine integration of Multi-res shading

RHI support for multiple viewports & scissors

FastGS plumbing

Post processing shaders



These are the main points where multi-res code was integrated in UE4

# Rendering passes

Geometry passes: add FastGS; render with 9 viewports & scissors

- Base pass, decals, deferred lights, depth prepass, distortion, shadow projection, translucency, velocity

Image-space passes: remap linear/multi-res UVs as needed

- Decals, deferred lights, postprocessing, shadow projection, tiled deferred lights

# Image-space passes

## Two common cases

Reconstruct 3D position from 2D+depth

- Map 2D position to linear before reconstructing

Large filter kernels

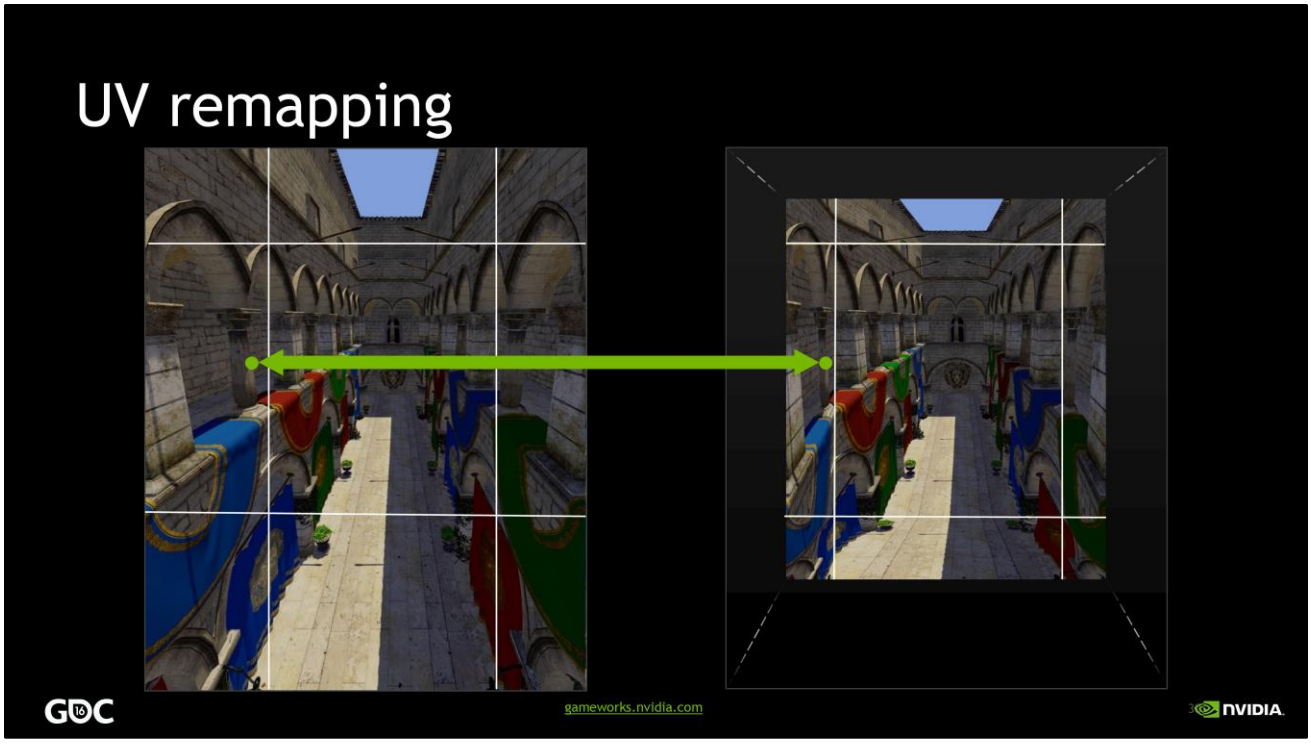
- Map sample point to linear, apply offset, map back to multi-res



For image-space passes, there are two things that commonly show up.

1. Reconstructing world space 3D position from 2D position + the depth buffer. Done in lots of shaders e.g. lighting, reflections, shadows. For these, the reconstruction math needs to be multi-res-aware.

2. Filter kernels like bloom, SSAO, and SSR. For consistent appearance, want the filter size to be consistent across viewports. So, offsets to sample points should be applied in linear space.




Remapping functions in the SDK map between corresponding points in linear UV space (left) and multi-res space (right)

This comes down to checking which viewport you're in (4 compares) and doing a 2D scale-bias (2 MADs). So it's not horribly expensive.

# Shadow projection

## Example of reconstructing 3D position

Before:

```
float2 ScreenUV = float2( SVPos.xy * Frame.BufferSizeAndInvSize.zw );
float SceneW = CalcSceneDepth( ScreenUV );
float2 ScreenPosition = ( ScreenUV.xy - Frame.ScreenPositionScaleBias.wz ) / Frame.ScreenPositionScaleBias.xy;
float4 ShadowPosition = mul(float4(ScreenPosition.xy * SceneW,SceneW,1), ScreenToShadowMatrix);
```



The shadow projection pixel shader is a good example of reconstructing 3D position.

Here, we take the input screen position (SV_POSITION) and use that to get the depth. We then remap the position from multi-res to linear space, and use that with the depth in the screen-space to shadow-space matrix.

# Shadow projection

## Example of reconstructing 3D position

After:

```
float2 ScreenUV = float2( SVPos.xy * Frame.BufferSizeAndInvSize.zw );
float SceneW = CalcSceneDepth( ScreenUV );

// Remap for warped viewports, no-op for regular viewports
float2 LinearScreenUV = MultiResMapRenderTargetMultiResToLinear(ScreenUV);

float2 ScreenPosition = (LinearScreenUV.xy - View.ScreenPositionScaleBias.wz) / View.ScreenPositionScaleBias.xy;
float4 ShadowPosition = mul(float4(ScreenPosition.xy * SceneW,SceneW,1), ScreenToShadowMatrix);
```



The shadow projection pixel shader is a good example of reconstructing 3D position.

Here, we take the input screen position (SV_POSITION) and use that to get the depth. We then remap the position from multi-res to linear space, and use that with the depth in the screen-space to shadow-space matrix.

# Multi-Res in UE4 4.11

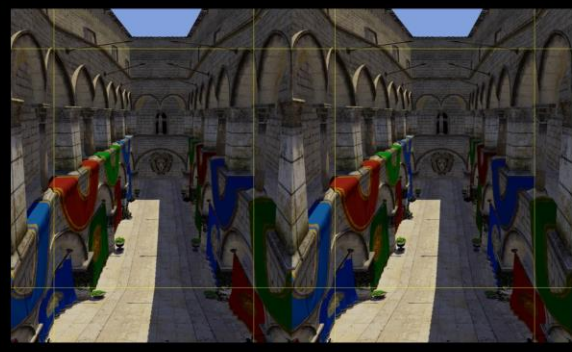
# UE4 4.11 Multires

Refactored to be more native UE code

More post processing effects supported

Instanced stereo support

Coming soon…



# Demo Time

Demo UE4 4.11!


NVIDIA
Questions?
cem@nvidia.com